

# Programming Abstractions

## Lecture 24: MiniScheme F

Stephen Checkoway

# Announcement

Homework 7 is now up on the website

- Use the same groups as before (this time, they should be created already)
- It's due on Dec. 17

Exam 2 is next week

- Monday, Dec. 13: Exam 2 review; come prepared with questions!
- Wednesday, Dec. 15: Exam 2, take home exam

Office hours

- Tomorrow at 13:30–14:30

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 of detail

Applications are parsed into two parts

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

# Evaluating an app-exp

# Evaluating an app-exp

How do we evaluate the app-exp we get from  
`(app-exp parsed-proc list-of-parsed-args)`?

# Evaluating an app-exp

How do we evaluate the app-exp we get from  
`(app-exp parsed-proc list-of-parsed-args)`?

In steps

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

# MiniScheme F: Lambdas

$EXP \rightarrow$  number  
| symbol  
| ( if  $EXP\ EXP\ EXP$  )  
| ( let (  $LET-BINDINGS$  )  $EXP$  )  
| ( lambda (  $PARAMS$  )  $EXP$  )  
| (  $EXP\ EXP^*$  )

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

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

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

parse into lit-exp

parse into var-exp

parse into ite-exp

parse into let-exp

parse into lambda-exp

parse into app-exp



# Implementing lambdas

## Parsing

Parse a lambda expression such as `(lambda (x y z) body)` into a new `lambda-exp` structure

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

# Implementing lambdas

## Evaluating

What should a `lambda-exp` evaluate to?

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

# Closures!

We need a closure data type

- `(closure params body env)`
- `(closure? obj)`
- `(closure-params c)`
- `(closure-body c)`
- `(closure-env c)`

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`

# To recapitulate

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

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

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

# Evaluating calls to closures

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 homework 6

```
(define (apply-proc proc args)
  (cond [(prim-proc? proc)
        (apply-primitive-op (prim-proc-op proc) args)]
        [else (error 'apply-proc "bad procedure: ~s" proc)]))
```

# Evaluating calls to closures

We need to add some code before the `else`

```
(define (apply-proc proc args)
  (cond [(prim-proc? proc)
        (apply-primitive-op (prim-proc-op proc) args)]
        [(closure? proc) ...]
        [else (error 'apply-proc "bad procedure: ~s" proc)]))
```

# How do we evaluate the closure?



# How do we evaluate the closure?

At a high level (don't think about MiniScheme here), given a closure and some arguments, how do we evaluate calling the closure?

# How do we evaluate the closure?

At a high level (don't think about MiniScheme here), 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

# How do we evaluate the closure?

At a high level (don't think about MiniScheme here), 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**

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

## Parsing

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)))
```

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

**Evaluating**

```
(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

The **procedure** evaluates to

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

The **arguments** evaluate to `'(3 5)`

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

## Evaluating

`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  $e[x \mapsto 3, y \mapsto 5]$

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

# Example 2

## Parsing

# Example 2

## Parsing

What is the result of parsing this?

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



# Example 2

## Parsing

What is the result of parsing this?

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

```
(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))))
```

# Example 2

## 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))))
```

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`:

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

# Example 2

## 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  $(\text{var-exp } 'f)$ , getting the closure above and evaluate the arguments getting  $'(6)$

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