# Programming Abstractions

## Week 8-2: MiniScheme C

Stephen Checkoway

# What can MiniScheme do at this point?

MiniScheme B has constant numbers

MiniScheme B has pre-bound symbols that are in the `init-env`

# Recall

(`parse input`) — Parses the input, at this point either a number or a variable, and returns a (`lit-exp num`) or (`var-exp sym`)

(`eval-exp tree e`) — Evaluates the parse `tree` in the environment `e`, returning a value

What does `(parse 15)` return?

A. 15

B. `'(lit-exp 15)`

C. It's an error

What does `(parse 'z)` return?

A. `'(lit-exp z)`

B. `'(var-exp z)`

C. It's an error

What does `(eval-exp (var-exp 'z) environment)` do?

A. Returns what `z` is bound to in `environment`

B. It's an error

C. It looks up with `z` is bound to, returning the result or causing an error if `z` is not bound

D. Something else

What does `(eval-exp (lit-exp 108) environment)` do?

A. Returns what `108` is bound to in `environment`

B. It's an error

C. It looks up with `108` is bound to, returning the result or causing an error if `108` is not bound

D. Returns 108

E. Something else

# Homeworks 6 and 7

Multiple steps, each adding parts to the MiniScheme interpreter

For each new type of expression
- ‣ Add a new data type
  - `ift-exp`
  - `let-exp`
  - etc.
- ‣ Add constructors, recognizers and accessors
- ‣ Modify `parse` to produce those
- ‣ Modify `eval-exp` to interpret them

```
EXP → number
    | symbol
    | ( if EXP EXP EXP )
    | ( let ( LET-BINDINGS ) EXP )
    | ( letrec ( LET-BINDINGS ) EXP )
    | ( lambda ( PARAMS ) EXP )
    | ( set! symbol EXP )
    | ( begin EXP* )
    | ( EXP EXP* )
LET-BINDINGS → LET-BINDING*
LET-BINDING → [ symbol EXP ]
PARAMS → symbol*
```

# Let's add arithmetic and some list procedures
**MiniScheme C**

Let's add +, -, *, /, `car`, `cdr`, `cons`, etc.

Students find this to be the hardest part of the project
‣ It's the first complex part
‣ It contains some things that make more sense later, once we add lambda expressions

# Many ways to call procedures

```
(+ 2 3)

((lambda (x y) (+ x y)) 2 3)

(let ([f +]) (f 2 3))
```

The parser can't identify primitive procedures like + because symbols like f may be bound to primitive procedures
‣ It can't tell because the parser **does not have access to the environment**

All that the parser can do is recognize a procedure application and parse
‣ the procedure; and
‣ the arguments

# Enter lists

So far, the input to MiniScheme A and B has just been a number or a symbol

If the input is a list, then the kind of expression it represents depends on the first element
- If the first element is `'lambda`, it's a lambda expression
- If the first element is `'let`, it's a let expression
- If the first element is `'if`, it's an if-then-else expression
- etc.

Applications don't have keywords, so any nonempty list for which the first element is not one of our supported keywords is an application

# Procedure applications
## MiniScheme C

*EXP* → number          parse into `lit-exp`
     | symbol          parse into `var-exp`
     | ( *EXP EXP*$^{*}$ )      parse into `app-exp`

An `app-exp` is a new data type that stores
▸ The parse tree for a procedure
▸ A list of parse trees for the arguments

Procedures to implement
▸ `(app-exp proc args)`
▸ `(app-exp? exp)`
▸ `(app-exp-proc exp)`
▸ `(app-exp-args exp)`

# Recursive implementation
## Parsing

Expressions are recursive: *EXP → ( EXP EXP\* )*

When parsing an application expression, you want to parse the sub expressions using parse

```
(define (parse input)
  (cond [(number? input) (lit-exp input)]
        [(symbol? input) (var-exp input)]
        [(list? input)
         (cond [(empty? input) (error ...)]
               [else (app-exp (parse (first input))
                              (...))])]
        [else (error 'parse "Invalid syntax ~s" input)]))
```

Parse the procedure

Parse the arguments

# How should you parse the arguments?

Consider `input` that looks like

`((lambda (x y) x) 2 3)` or

`(f 4 5 6)`

The procedure part can be parsed with `(parse (first input))`

How should you parse the arguments?

# Evaluating an `app-exp`

Evaluate the procedure part

Evaluate each of the arguments

If the procedure part evaluates to a primitive procedure, call a procedure you'll write that will perform the operation on the arguments
‣ E.g., if the primitive procedure is *, then you'll want to call * on the arguments

The tricky part is what does it mean to evaluate the procedure part?

# Evaluating the procedure part of an `app-exp`

Consider the input `'(+ 2 3 4)`

The procedure part is `'+` which will be parsed as `'(var-exp +)`

Variable reference expressions are evaluated by looking the symbol up in the current environment

Therefore, we need our initial environment to contain a binding for the symbol `'+` (and all the other primitive procedures we want to support)

# prim-proc data type

We can create a new data type prim-proc
‣ `(prim-proc symbol)`
‣ `(prim-proc? value)`
‣ `(prim-proc-symbol value)`

The prim-proc is *only* used to interpret expressions so where should this data type and its procedures be defined?

# Adding primitives to our initial environment

```
(define primitive-operators
  '(+ - * /))

(define prim-env
  (env primitive-operators
       (map prim-proc primitive-operators)
       empty-env))

(define init-env
  (env '(x y) '(23 42) prim-env))
```

# **Evaluating an** `app-exp`

Recall: `app-exp` stores the parse tree for the procedure and a list of parse trees
for the arguments

We need to evaluate all of those; add something like the following to `eval-exp`

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

eval-exp's environment
parameter

# Applying a procedure

The `apply-proc` procedure takes an evaluated procedure and a list of evaluated arguments

It can look at the procedure and determine if it's a primitive procedure
‣ If so, it will call apply-primitive-op
‣ If not, it's an error for now; later, we'll add code to deal with non-primitive procedure (i.e., normal lambdas)

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

# Applying primitive operations
## (apply-primitive-op op args)

apply-primitive-op takes a symbol (such as `'+` or `'*`) and a list of arguments

You probably want something like

```
(define (apply-primitive-op op args)
  (cond [(eq? op '+) (apply + args)]
        [(eq? op '*) (apply * args)]
        ...
        [else (error ...)]))
```

What is returned by `(parse '(* 2 3))`?

A. `'((prim-proc *) 2 3)`

B. `'((prim-proc *) (lit-exp 2) (lit-exp 3))`

C. `'(app-exp (prim-proc *) ((lit-exp 2) (lit-exp 3)))`

D. `'(var-exp * (lit-exp 2) (lit-exp 3))`

E. `'(app-exp (var-exp *) ((lit-exp 2) (lit-exp 3)))`

When evaluating an `app-exp`, the procedure and each of the arguments are evaluated. For example, when evaluating the result of `(parse '(- 20 5))`, there will be three recursive calls to `eval-exp`, the first of which is evaluating `(var-exp '-)`.

What is the result of evaluating `(var-exp '-)`?

A. `#<procedure:->` (i.e., the procedure – itself)

B. `'(app-exp -)`

C. `'(prim-proc -)`

D. It's an error because – requires arguments

What is the result of `(eval-exp (parse '(* 4 5)) init-env)`?


A. `20`

B. `'(app-exp (var-exp *) ((lit-exp 4) (lit-exp 5)))`

C. `'(prim-proc * 4 5)`

D. `'(prim-proc (var-exp *) (lit-exp 4) (lit-exp 5))`

E. `'(app-exp (prim-proc *) 4 5)`

# Why go to all that trouble?

In a later version of MiniScheme, we'll implement lambda

We'll deal with this by adding a line to `apply-proc` that will apply closures

# Adding other primitive procedures

In addition (pardon the pun) to `+`, `-`, `*`, and `/`, you'll add several other primitive procedures

‣ `add1`

‣ `sub1`

‣ `negate`

‣ `list`

‣ `cons`

‣ `car`

‣ `cdr`

And you'll add a new variable `null` bound to the empty list

**What does** `(car (list 3 5 2))` **parse to?**

# What does `(car (list 3 5 2))` parse to?

```
'(app-exp (var-exp car)
         ((app-exp (var-exp list)
                  ((lit-exp 3)
                   (lit-exp 5)
                   (lit-exp 2)))))
```

# Adding additional primitive procedures

1. Add the procedure name to `primitive-operators`

2. Add a corresponding line to the `cond` in `apply-primitive-op`

E.g.,
```
[(eq? op 'car) (car (first args))]
[(eq? op 'list) args]
```

# What can MiniScheme C do?

Numbers

Pre-defined variables

Procedure calls to built-in procedures