Programming Abstractions

Lecture 19: MiniScheme B and C

Stephen Checkoway

What can MiniScheme do at this point?

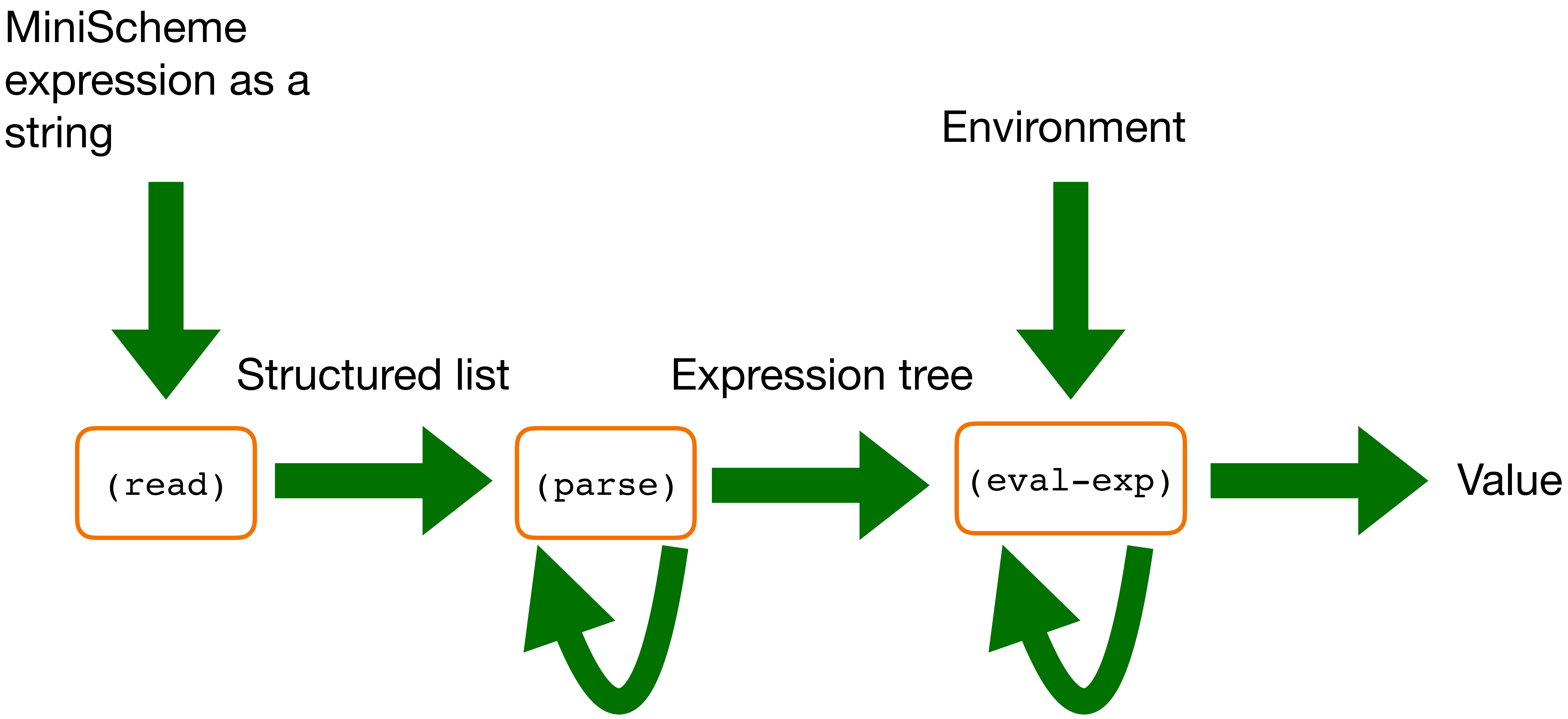
MiniScheme A has constant numbers

Recall

`(parse input)` — Parses the input, at this point only numbers, and returns a `(lit-exp num)`

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

Interpreter flow



Let's add some variables!

MiniScheme B

Grammar

EXP → number	parse into `lit-exp`
| symbol	parse into `var-exp`

Data type for a variable reference expression

```
(struct var-exp (symbol) #:transparent)
```

- `(var-exp symbol)`
- `(var-exp? exp)`
- `(var-exp-symbol exp)`

Parsing symbols

MiniScheme B

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

When I run `(parse 'foo)`, I get

```
(var-exp 'foo)
```

Interpreting symbols

MiniScheme B

```
(define (eval-exp tree e)
  (cond [(lit-exp? tree) (lit-exp-num tree)]
        [(var-exp? tree)
         (env-lookup e (var-exp-symbol tree))]
        [else (error 'eval-exp "Invalid tree: ~s" tree)]))
```

You'll need a working `env-lookup`

```
> (env-lookup init-env 'x)
23
> (eval-exp (var-exp 'x) init-env)
23
```

Assuming that `x` is bound to 10 and `y` to 25 in `init-env`, what does `(parse 'x)` return (assuming the implementation discussed so far)?

A. 10

B. 25

C. `(lit-exp 10)`

D. `(var-exp 'x)`

E. It's an error of some sort

Assuming that `x` is bound to 10 and `y` to 25 in `init-env`, what does `(eval-exp (parse 'x) init-env)` return (assuming the implementation discussed so far)?

A. 10

B. 25

C. `(lit-exp 10)`

D. `(var-exp 'x)`

E. It's an error of some sort

What can MiniScheme do at this point?

MiniScheme B has constant numbers

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

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

Data type procedures

- `(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?

What is the result of `(parse '(foo x y z))`?

A. `(app-exp 'foo '(x y z))`

B. `(app-exp (var-exp 'foo) '(x y z))`

C.
```
(app-exp (var-exp 'foo)
         (list (var-exp 'x) (var-exp 'y) (var-exp 'z)))
```

D.
```
(app-exp 'foo
         (list (var-exp 'x) (var-exp 'y) (var-exp 'z)))
```

E. It's an error because the variables `foo`, `x`, `y`, and `z` aren't defined

What is the result of `(parse '(foo (add1 x)))`?

```
A. (app-exp (var-exp 'foo)
            (app-exp (var-exp 'add1) (var-exp 'x)))

B. (app-exp (var-exp 'foo)
            (list (app-exp (var-exp 'add1) (var-exp 'x))))

C. (app-exp (var-exp 'foo)
            (list (app-exp (var-exp 'add1)
                           (list (var-exp 'x))))))
```

D. It's an error