Programming Abstractions

Lecture 18: MiniScheme A

Stephen Checkoway

Structure of MiniScheme

Environment

`env.rkt`

- Contains the environment data type with constructor
  `(env list-of-symbols list-of-values previous-env)`
- Contains other procedures to recognize and access the symbols, values, and previous environment
- Your task is to implement `(env-lookup environment symbol)`

Structure of MiniScheme

Parser

`parse.rkt`

- Contains data types for let expressions, lambda expressions, if-then-else expressions, procedure-application expressions and so on
- Builds a parse tree out of these data types from an expression

```
> (parse '(let ([f (lambda (x) (+ x 1))]) (f 5)))
(let-exp '(f) (list (lam-exp '(x) ...)) (app-exp ...))
```

- You get to implement all of this, bit by bit

Structure of MiniScheme

Interpreter

`interp.rkt`

- Contains data types for closures and primitive procedures (i.e., built-in procedures)
- Takes an expression tree and an environment and returns a value

  ```
  > (eval-exp exp-tree environment)
  ```
- You get to implement all of this, bit by bit, at the same time you're implementing the parser

What, exactly, is the input to parse?

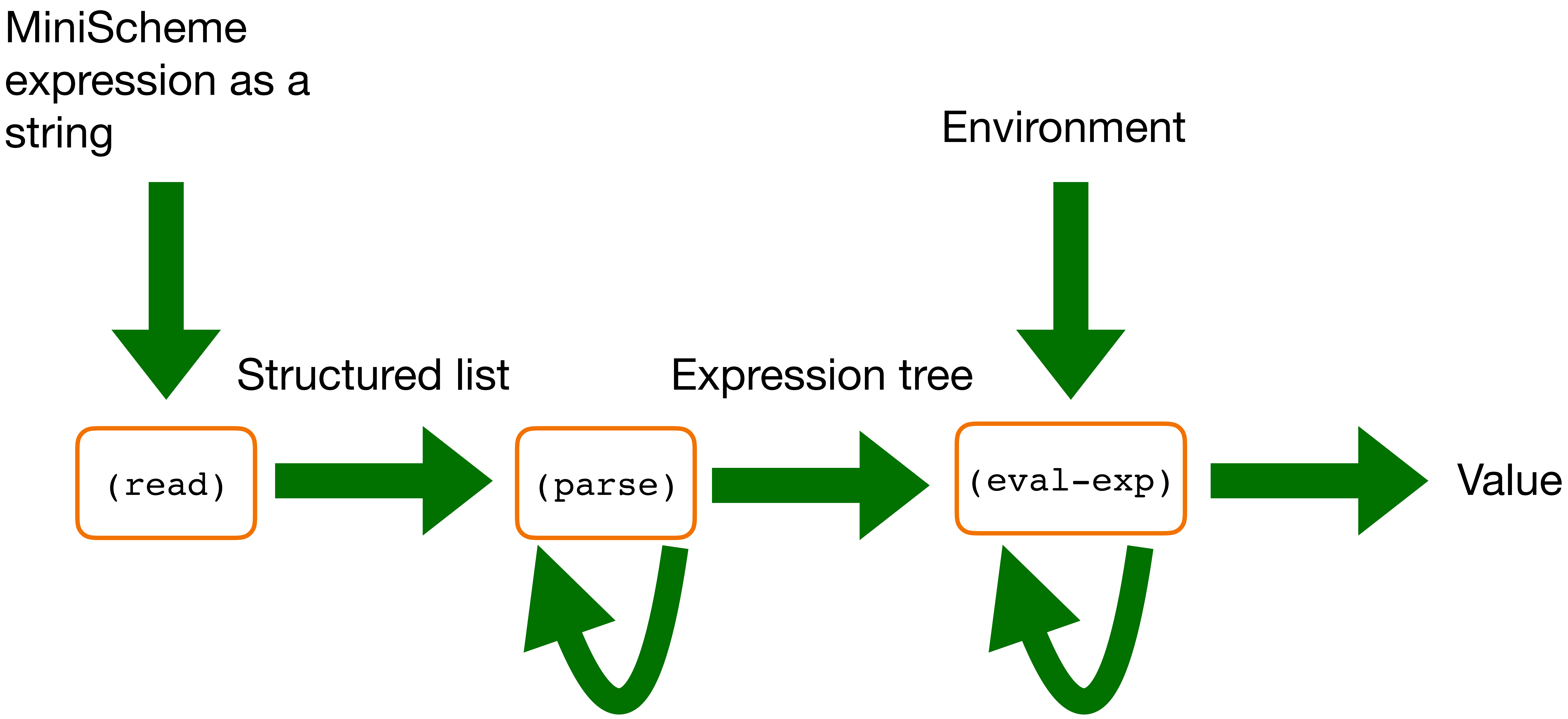
Scheme (and thus Racket) has a procedure `(read)` that reads input and returns a structured list or an atom

The interpreter project flow

1. `read` returns a structured list which is passed to `parse` as the input parameter
2. `parse` produces a parse tree containing nodes like `lit-exp`, `let-exp`, and `app-exp` which is passed, along with `init-env` to `eval-exp`
3. `eval-exp` takes a parse tree and an environment and evaluates the expression, returning the result

Do a demo with `(let ([x 100] [z 25]) (+ (- x 10) z))`

Interpreter flow



Programs are just structured lists

Parsing

Consider the program

```
(let ([x 10]
      [y 20])
  (+ x y))
```

This is just a structured list containing the symbols `let`, `f`, `x`, `y`, and + and the numbers 10 and 20

Your first task is going to be to build some new data types to represent programs by parsing these structured lists

A full grammar for Minischeme

EXP → number

| symbol

| (`if` *EXP EXP EXP*)

| (`let` (*LET-BINDINGS*) *EXP*)

| (`letrec` (*LET-BINDINGS*) *EXP*)

| (`lambda` (*PARAMS*) *EXP*)

| (`set!` symbol *EXP*)

| (`begin` *EXP**)

| (*EXP*$^+$)

LET-BINDINGS → *LET-BINDING**

LET-BINDING → [symbol *EXP*]

PARAMS → symbol*

Start simple: only numbers

EXP → number   parse into `lit-exp`

We're going to need a data type to represent literal expression (and the only type of literals we have are numbers)

We're going to want something like

```
(struct lit-exp (num) #:transparent)
```

which gives

```
(lit-exp num) ; constructor
(lit-exp? exp) ; recognizer
(lit-exp-num exp) ; accessor
```

Parsing numbers

Our first parser: MiniScheme A

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

This and the definition of the `lit-exp` data type belong in `parse.rkt`

You don't need to implement it exactly the way I do

That said, when I run `(parse 52)`, I get
`(lit-exp 52)`

Provide the definitions

`(provide proc1 proc2 data1 data2 ...)`

We want `parse.rkt` to be just one module in our program so make sure to provide the procedures

- `(provide parse)`
- Also the procedures for creating and manipulating the `lit-exp`

What does `(parse 15)` return (assuming the implementation we've discussed so far)?

A. `15`

B. `(number 15)`

C. `(lit-exp 15)`

D. `(lit-exp "15")`

E. It's an error of some sort

Evaluating literals (interp.rkt)

Our first interpreter: MiniScheme A

We'll need to require `env.rkt` and `parse.rkt` to get access to those modules' procedures

The main procedure in `interp.rkt` is `eval-exp`

Extracts the number from the `lit-exp`

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

What does `(eval-exp 15 empty-env)` return (assuming the implementation we've discussed so far)?

A. `15`

B. `(value 15)`

C. `(lit-exp 15)`

D. It's an error of some sort

What does `(eval-exp (lit-exp 15) empty-env)` return (assuming the implementation we've discussed so far)?

A. `15`

B. `(value 15)`

C. `(lit-exp 15)`

D. It's an error of some sort

Putting them together

```
> (parse 107)
(lit-exp 107)

> (lit-exp 107)
(lit-exp 107)

> (eval-exp (lit-exp 107) empty-env)
107

> (eval-exp (parse 107) empty-env)
107
```

Read-eval-print loop

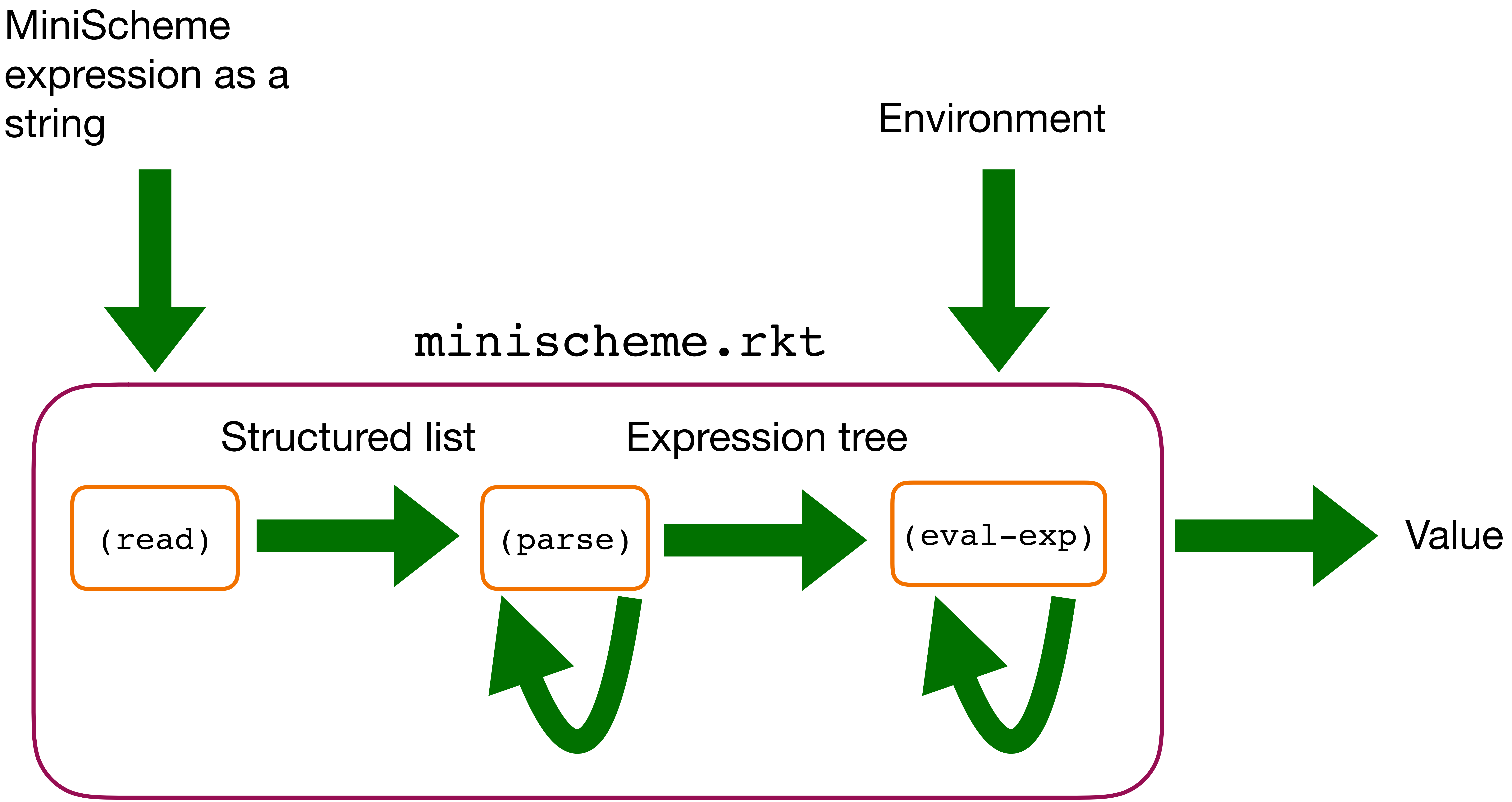
Having to call `parse` and then `eval-exp` over and over is a hassle

It'd be better if we could run a read-eval-print loop that would read in an expression from the user, parse it, and evaluate it in an environment

`minischeme.rkt` will do this for you but it needs several things (`provide`)

- `parse.rkt`
  - A `(parse input)` procedure
- `interp.rkt`
  - An `(eval-exp tree environment)` procedure
  - An initial environment `init-env`
    Something like
    `(define init-env (env '(x y) '(23 42) empty-env))`

minischeme.rkt



Running the read-eval-print loop

Open `minischeme.rkt` in DrRacket, click Run

Enter expressions in the box (only numbers are supported right now)

Enter `exit` to exit MiniScheme

```
Welcome to DrRacket, version 7.7 [3m].
Language: racket, with debugging; memory limit: 128 MB.
MS> 105
105
MS> 23
23
MS> exit
returning to Scheme proper
```

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.
- 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*

Interpreter flow