Programming Abstractions

Week 7-2: MiniScheme Parser

Stephen Checkoway

Structure of MiniScheme

Environment

`env.rkt`

- Contains the environment data type
  `(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) ((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

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*

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

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

```
(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)`

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 y) z))`

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`

Evaluating literals

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)]))
```

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
```

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

A. `15`

B. the result of `(number 15)`

C. the result of `(lit-exp 15)`

D. the result of `(lit-exp "15")`

E. It's an error of some sort

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

A. `15`

B. the result of `(value 15)`

C. the result of `(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. the result of `(value 15)`

C. the result of `(lit-exp 15)`

D. It's an error of some sort

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))`

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
```

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

- `(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
```

What can MiniScheme do at this point?

MiniScheme B has constant numbers

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

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)`

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 ...)]))
```

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