Programming Abstractions

Lecture 19: 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 only numbers, and returns a `(lit-exp num)`

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

MiniScheme B grammar

MiniScheme B

Grammar

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

Data types constructed by `parse`

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

MiniScheme B `parse`

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

MiniScheme B eval-exp

```
(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`

What does `(parse 275)` return?

A. `275`

B. `(lit-exp 275)`

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

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

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.

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

`(foo x 8 y)` is an application with procedure `foo` and arguments `x`, `8`, and y

Which rule should we add to our grammar to support procedure calls like `(+ 10 15)` and `(car lst)`?

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

A. (*PROC ARGS*)

B. (*PROC ARGS**)

C. (symbol *EXP**)

D. (*EXP**)

E. (*EXP EXP**)

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

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

```
(struct app-exp (proc args) #:transparent)
```

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

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 should the result of evaluating the procedure part be?

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

- `(struct prim-proc (symbol) #:transparent)`

We're going create a bunch of these

- `(prim-proc '+)`
- `(prim-proc '-)`
- `(prim-proc 'car)`
- `(prim-proc 'cdr)`
- `(prim-proc 'null?)`
- ...

prim-proc

A `prim-proc` is a **value** that will be returned by `eval-exp`, just like numbers are in MiniScheme now

A `(prim-proc 'car)` is to the MiniScheme interpreter exactly the same thing `#<procedure:car>` is to DrRacket

Since prim-proc is **only** used to interpret expressions, where should this data type be defined?

Binding variables to prim-proc

In DrRacket, + is bound to `#<procedure:+>`

In MiniScheme, + needs to be bound to `(prim-proc '+)` in our initial environment, `init-env`

And similarly for `-`, `*`, `/`, `car`, `cdr`, `null?` etc.

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

Evaluating the arguments

In parse, we could simply map parse over the arguments to get a list of trees corresponding to our arguments

We cannot simply use `(map eval-exp (app-exp-args tree))` to evaluate them, why?

What should we map instead?

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., closures produced by evaluating 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 '*) (list (lit-exp 2) (lit-exp 3)))`

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

E. `(app-exp (var-exp '*) (list (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 '*) (list (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

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) (apply car args)]
[(eq? op 'cdr) (apply cdr args)]
[(eq? op 'list) (apply list args)]
```

What can MiniScheme C do?

Numbers

Pre-defined variables

Procedure calls to built-in procedures