

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

$EXP \rightarrow$ number	parse into <code>lit-exp</code>
symbol	parse into <code>var-exp</code>

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)?

$EXP \rightarrow$	number	parse into lit-exp
	symbol	parse into var-exp
	???	

A. (*PROC ARGS*)

B. (*PROC ARG**)

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

$EXP \rightarrow$ number	parse into <code>lit-exp</code>
symbol	parse into <code>var-exp</code>
(<i>EXP EXP*</i>)	parse into <i>app-exp</i>

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: $EXP \rightarrow (EXP EXP^*)$

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
- In MiniScheme C, primitive procedures are the only supported type of procedures; this will change later

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))
```