

Programming Abstractions

Lecture 20: MiniScheme C continued

Stephen Checkoway

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)
```

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)))`


Evaluating an app-exp

To evaluate an app-exp

- Evaluate the procedure
- Evaluate the arguments
- Apply the procedure to 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)) ]
```



Evaluating the procedure yields a value

New type whose instances represent primitive procedure values

- `(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?)`
- ...

Later, we'll support closures too!

We added 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))
```

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

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?

After evaluating proc and args, need to apply

To evaluate an app-exp

- Evaluate the procedure ✓
- Evaluate the arguments ✓
- Apply the procedure to 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 (map ... (app-exp-args tree))])
    (apply-proc proc args)) ]
```

Applying a procedure

The `apply-proc` procedure takes an evaluated procedure (a value of some sort) and a list of evaluated arguments (a list of values)

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 "...)]))
```

When implementing cdr, what should we add to apply-primitive-op?

```
(define (apply-primitive-op op args)
  (cond ...
    [(eq? op 'cdr) ???]
    ...
    [else (error ...)]))
```

A. (cdr args)

B. (rest args)

C. (cdr (first args))

D. (apply cdr args)

E. More than one of the above works correctly

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 is the result of `(eval-exp (parse '(* 4 5)) empty-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. An error of some sort

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. An error of some sort

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

What can MiniScheme C do?

Numbers

Pre-defined variables

Procedure calls to built-in (primitive) procedures

Testing

You'll need to test your implementation

Make sure you test as you go!

One test file for each MiniScheme module

- `env-tests.rkt`
- `parse-tests.rkt`
- `interp-tests.rkt`

Parser tests

Test that you can parse numbers, symbols, and applications (so far)

```
; Test that (var-exp? (parse 'x)) returns #t
(test-pred "Variable"
           var-exp?
           (parse 'x))
```

```
; Test that (parse 'y) returns (var-exp 'y)
(test-equal? "Variable equality"
             (parse 'y)
             (var-exp 'y))
```

Parser tests

```
; Test that (parse '()) raises exception
(test-exn "Invalid syntax ()"
  exn:fail?
  (λ () (parse '())))
```

```
; Test that (parse "string") raises exception
(test-exn "Invalid syntax \"string\""
  exn:fail?
  (λ () (parse "string")))
```

Interpreter tests

; Construct a test environment

```
(define test-env  
  (env '(foo bar) '(10 23) init-env))
```

; Test evaluating literals

```
(test-equal? "Literal"  
              (eval-exp (lit-exp 5) test-env)  
              5)
```

; Test evaluating variables

```
(test-equal? "Variable"  
              (eval-exp (var-exp 'foo) test-env)  
              10)
```

Interpreter tests

```
; Test primitive procedures
(test-equal? "Primitive cons"
              (eval-exp (var-exp 'cons) test-env)
              (prim-proc 'cons)))
```


WARNING

; Do NOT do this if you can help it

```
(test-equal? "Apply (- 23 3)"  
             (eval-exp (parse '(- 23 3)) test-env)  
             20)
```

Two reasons

1. You'll want to test the interpreter separately from the parser
2. It's *extremely* easy to make a mistake:

```
(test-equal? "Apply (- 23 3)"  
             (eval-exp (parse (- 23 3)) test-env)  
             20)
```

Tests can be run independently or all at once

```
(run-tests env-tests)
```

```
(run-tests parse-tests)
```

```
(run-tests interp-tests)
```

Running the tests.rkt file will run all tests at once via

```
(run-tests all-tests)
```

Or you can get a gui via

```
(test/gui all-tests)
```

