Programming Abstractions

Lecture 20: MiniScheme C continued

Stephen Checkoway

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