Reviewing Primitive Procedures

We spent most of Tuesday's class working on primitive procedures. We particularly focused on arithmetic procedures, but the lab also asks you to implement some list procedures.

Question: What does * parse to?

A. '*
B. ('var-ref *)
C. ('prim-proc *)
D. parsing * gives an error

Answer B: ('var-ref *')   * is a symbol; like all symbols it parses to a var-ref.

Question: What does * evaluate to?

A. '*
B. ('var-ref *)
C. ('prim-proc *)
D. An error

Answer C: (prim-proc *) We set up the environment to bind all primitive procedures to prim-proc versions of themselves. That's how we can tell what kind of critter a function is: we evaluate it. Primitive procedures evaluate to prim-procs, lambda expressions evaluate to closures.

Question: What does (* 4 5) parse to?

A. 20
B. ('lit-exp 20)
C. ('app-exp ('var-ref *) (('lit-exp 4) ('lit-exp 5)))
D. ('app-exp ('prim-proc *) (('lit-exp 4) ('lit-exp 5)))

Answer C: ('app-exp ('var-ref *) (('lit-exp 4) ('lit-exp 5)))

We parse an application by building an app-exp with two fields. The first field is the parsed procedure; the second field is the list of parsed arguments.

What does (* 4 5) evaluate to?

A. 20
B. ('lit-exp 20)
C. ('app-exp * (4 5))
D. An error

If this makes any sense at all, (* 4 5) had better evaluate to 20.

How does that happen?

1. (* 4 5) parses to ('app-exp ('var-ref *) (('lit-exp 4) ('lit-exp 5)))
2. That evaluates to (apply-proc ('prim-proc *) (4 5))
3. That evaluates to (apply-primitive-op * (4 5))
4. That evaluates to 20.

Question: What does (list 3 5 2) parse to?

Answer: (app-exp ('var-ref 'list)   (('lit-exp 3) ('list-exp 5) ('lit-exp 2)))

OK; how is that evaluated?

Answer: We call

```
        (apply-proc  ('prim-proc 'list)  (3 5 2))
```

which calls

```
        (apply-primitive-op  'list  (3 5 2))
```

which ought to evaluate to (3 5 2)

We said that function (apply-primitive-op p args) has the following line for +

```
 [(eq? p '+)  (+ (car args) (cadr args))]
```

What is the corresponding line for list?

```
[(eq? p 'list) ????]
```

Answer: [(eq? p 'list) args]

Just one more of these. The MiniScheme function *first* is just like *car*; it gives the first element of its argument, which should be a list. How do we parse the expression '(first (list 3 5 2)) ??

('app-exp (var-ref first) (('app-exp ('var-ref list) ((lit-exp 3) (lit-exp 5) (lit-exp 2))))))

How does that get evaluated?

(apply-proc (prim-proc 'first) (foo))
where *foo* is the value of the inner app-exp; we've already seen that this evaluates to (3 5 2)
So we do (apply-proc (prim-proc 'first) ((3 5 2)))

(apply-proc (prim-proc 'first) ((3 5 2))) calls
(apply-primitive-op 'first ((3 5 2)))

What is the line of (apply-primitive-op p args) for first?

[(eq? p 'first) (car **(car args)**)]

Note that the quote operator ' is not part of MiniScheme.
(first (list 1 2 3)) is a valid MiniScheme expression but
(first '(1 2 3)) is not.