MiniScheme C
Primitive Procedures

It is hard to do much in a programming language without arithmetic. MiniScheme C adds arithmetic operators and all of the other primitive procedures (such as the list operators car, cdr and cons). Many students find this the hardest step of the project. That might be because it is the first complex step and it might be because some of the things we do need to mesh with later steps (when we call lambda expressions). Whatever the reason, be aware that this is a trouble point. Make sure you understand what all of the code is supposed to do before you write it.

Consider the many ways procedure calls (or *applications*) can be expressed in Scheme:

```
(+  2  3)
(f  2  3)
(  (lambda (x y) x)  2  3)
(let ([f  +]) (f 2 3))   ; yes, this works
```

That last example should show you that the parser can't identify primitive procedures: in (f 2 3) f might be a variable bound to a primitive procedure. There is no way our parser can recognize that (it doesn't have access to the environment). All that our parser can do with an application is to recognize it as an application and to parse the various parts of it.

So far everything we have parsed has been an atom: either a number or a symbol. If our input is a list rather than an atom the kind of expression it represents depends on its first element. If this is 'lambda we must have a lambda-expression; if it is 'let we have a let-expression, and so forth. Applications don't have keywords. We will assume that any input that is a list in which we don't recognize the first element must be an application.

Our grammar at this point is

```
EXP ::- NUMBER        ; parse into lit-exp
      | SYMBOL        ; parse into var-ref
      | (EXP EXP*)    ; parse into app-exp
```

An app-exp is a datatype that store two elements: the parse tree for a procedure, and a *list* of parse trees for the arguments. It is easy enough to implement that datatype:

```
(define new-app-exp (lambda (proc args)
                      (list 'app-exp proc args)))
```

You can build a recognizer for it. Note that you need two getters: one for the procedure part and one for the list of arguments.

This gives us parser code that looks like this:

```
(define parse (lambda (input)
        (cond
            [(number? input) (new-lit-exp input)]
            [(symbol? input) (new-var-ref input)]
             [(not (pair?  input)) (error 'parse "Bad syntax ~s" input)]
             [else (new-app-exp ..............)])))
```

What are the parts of an app-exp? Consider our example applications:

```
(+  2  3)
(f  2  3)
(  (lambda (x y) x)  2  3)
```

In each of these the procedure part is the car of the input; the argument list is the cdr of the input. We recursively parse the procedure part, and each element of the argument list. The latter is a good place to use *map*:

```
(new-app-exp (parse (car input)) (map parse (cdr input)))
```

So parsing an application into an app-exp tree is easy. Evaluating an app-exp in eval-exp is also easy: we evaluate the procedure part, we evaluate each element of the arg list, and if the procedure is a primitive procedure we call something that tells us how to do it.

But wait -- what does it mean to "evaluate the procedure part"??

Consider the simple application (+ 2 3). The procedure part is the car of this expression: '+ This is a symbol. We know how to parse symbols; they parse into var-refs. So the procedure part of the app-exp for this expression is ('var-ref +). We evaluate a var-ref by looking its symbol up in the environment. This means our initial environment needs to contain a binding for the symbol +. What should we bind it to? An easy solution is to make yet another datatype prim-proc, so when we look up + we get ('prim-proc +). This way when we evaluate (+ 2 3) we will immediately know that + is a primitive procedure.

This is easy to implement. After Lab5B your initial environment is something like

```
(define init-env (extended-env '(x y) '(10 23) (empty-env)))
```

We can extend this to include primitives as follows:

```
(define prims '(+ - * /)) ; and anything else you want to be a primitive
```

Extend init-env to:

```
(define init-env (extended-env '(x  y) '(10 23)
                       (extended-env prims (map new-prim-proc prims)
                       (empty-env))))
```

where new-prim-proc is the constructor for your prim-proc datatype.

Now we are ready to give the eval-exp code for an app-exp. Remember that the app-exp stores a parsed procedure and a list of parsed arguments. We evaluate those and call a procedure called apply-proc. The appropriate line of eval-exp is

```
[(app-exp? tree) (apply-proc (eval-exp (app-proc tree) env)
                             ; oh, how will we evaluate a0
                             ; list of parsed arguments....?)]
```

Note that in that line of code app-proc is the getter for the procedure part of an app-exp node.

The apply-proc function gets an evaluated procedure and evaluated arguments. It can look at the procedure and determine if it is a primitive procedure:

```
(define apply-proc (lambda (p arg-values)
        (cond
                [ (prim-proc? p) (apply-primitive-op (prim-proc-sym p) arg-values)]
                [else (error 'apply-proc "Bad procedure: ~s" p)])))
```

Finally, (apply-primitive-op op vals) gets the actually primitive procedure symbol (such as +) and the actual argument values. It has lines like

[(eq? op '+) (+ (car vals) (cadr vals))]

This might seem like a long way around to implement arithmetic operators but it sets the stage for everything else we will do with applications. After we implement lambda expressions we will only need to add to apply-proc a line that says how to call a closure (which of course is what we get when we evaluate a lambda expression) with arguments. One line of code will allow us to call user-defined functions. That is exciting! We really are getting somewhere.