Programming Languages and Computation

Week 3: Primitive Operators

1. In this exercise, you will extend Brischeme with a new primitive operator for less-than-or-equal.

   (a) Add a new nullary constructor Leq to the type primops in the file *lib/ast.ml*.

   (b) Explain that this operator looks like "<=" as a string, for the purposes of printing (e.g. in debug messages), by adding an extra case to the function string_of_primop in file *lib/ast.ml*.

   (c) Find and understand the function lex_bool from the file *lib/lexer.ml*. Using this for inspiration, write a function lex_le_or_leq which, when invoked in a state where peek () = '<', returns either PrimOp Leq or PrimOp Less depending on whether the next character in stream is '=' or not. Replace the right-hand-side of the '<' case in the function lex_init by a call to this function.

   (d) Find the code for evaluating a call of the primitive operators, in the function step_sexp of the file lib/eval.ml. Add a case for the less-than-or-equal primitive operator, taking the existing code for the less-than operator as inspiration.

2. In this exercise, you will extend Brischeme with pairs (tuples of length 2). The idea is that pairs will be constructed using a new primitive binary operator *cons*, and new primitive unary operators *car* and *cdr* can be used to project out the first and second components, as in:

```
> (define x (cons 2 6))
> (car x)
2
> (cdr x)
6
```

   (a) Add new nullary constructors Cons, Car and Cdr to the type primops in the file *lib/ast.ml*, and explain what they look like as strings (for output purposes) in string_of_primop.

   (b) Modify the function lex_kw_or_id to return TkPrimOp Cons, TkPrimOp Car and TkPrimOp Cdr when the lexeme is "cons", "car" and "cdr" respectively.

   (c) Explain that Call (Cons, [v1; v2]) is already a value (and therefore does not make any execution step) whenever v1 and v2 are values, by adding a case Call (Cons, [v1; v2]) **when** is_value v1 && is_value v2 -> **true** to the function is_value of file *lib/eval.ml*. This will require that is_value is now defined as a recursive function, using **let rec**.

(d) Explain how calls `Call (Car, [Call (Cons, [v1; _])])` and `Call (Cdr, [Call (Cons, [_; v2])])` are evaluated, by modifying the code for primitive operations in `step_sexp`.