Project 1: A Lisp Interpreter

A tiny Lisp-subset interpreter written in C is available in the file `~cs61c/lib/lisp.c`. Your job is to extend the interpreter. Part of the object of this exercise is for you to be sure to learn any C details needed for the project that you haven't learned already, so ask questions if necessary.

1. Add a symbol table so that `eval` can look up the values of symbols. (Since there are no user-defined procedures in this Lisp, all symbols will be global.) Use an appropriate searchable data structure from 61B; it needn't be the fastest possible.

2. Modify `eval` so that the operator (first) subexpression of a compound expression is evaluated recursively, as the argument subexpressions already are. Invent a primitive-procedure data type in addition to the data types symbol, number, pair, and niltype; put the primitives in the initial symbol table. [The interpreter as it exists now doesn't look up primitives in the symbol table; it has their names built in, as if they were special forms. But you'll change this, so that procedures are first-class values, as in Scheme.] When you've done this, the interpreter should be able to evaluate an expression such as

```
((car (cdr (cons 3 (cons + 'foo)))) 4 5)
```

Hint: Your representation for a primitive might include the number of arguments expected as well as a pointer to the C procedure that carries it out.

3. Add the `define` special form to add an entry to the symbol table, e.g.,

```
(define x (+ 2 3))
```

would add an entry with the name `x` and the value 5. It should return the name of the defined symbol (`x` in this example).

4. Add a vector (array) data type. For this data type, the `struct thing` should include a pointer to a dynamically-allocated block of storage just large enough for the array, and a number indicating the length of the array. The elements of the array should be pointers to `thing`s. The reader should recognize the notation `#(4 5 foo baz)` to represent a vector, and should allocate (for this example) an array of length four. The printer should know how to print vectors, too. The elements of a vector can be of any type, of course, including lists and vectors. Also implement the following primitives:

`(make-vector length)`	returns a vector of `length` empty lists
`(vector-ref vector index)`	returns the `index`th element
`(vector-set! vector index value)`	changes the element's value

5. **Optional — for fun, not for credit! Don't do this until the rest of the project is completed.** Add user-defined procedures. To do this, create a new `USERPROC` type that includes the procedure's text and its defining environment. Add a `lambda` special form to create user procedures. The text of the procedure is just the `lambda` expression (or its cdr, if you prefer, leaving out the word `lambda` itself). An environment is a list of frames, where each frame is a symbol table like the one you created earlier. Modify the symbol

lookup routine to handle a list of symbol tables. Then you have to write `apply` so that for user-defined procedures it extends the procedure's defining environment with a new frame in which the procedure's formal parameters are bound to the calling expression's actual argument values, and you have to write `eval-sequence` that evaluates the procedure's body. (Actually, since we have no primitives with side effects — no mutation, for example — you could get away with only allowing one expression in the body.) Basically you are redoing the metacircular evaluator from chapter 4 of *SICP*, but writing in C instead of in Scheme. This really isn't hard if you remember 61A!