Interpreter, if/let

It's time to start actually implementing the interpreter portion of the project!

If you were previous working with a partner, this is the time where you fork off your work from each other. Starting with this assignment and going forward, you will be submitting your own work. That said, you are definitely allowed to continue talking to your old partner about your work, and you may share ideas (as you can with anyone in the class). If you have a bug you can't find, your old partner can look at that and can help you find it, but may not walk away with a record of the code that you used. The code that you submit must be your own.

Here's a simple way to explain it: it's ok to look at someone else's code if you're helping them solve a problem. It's not ok to copy code or to transmit code by paper, electronic, or other means. I'm happy to clarify if need be, please ask.

1. Get started

Download this 05-iflet.zip file, and extract the zip file to your ProgrammingLanguages folder that you've been working in.

2. Functionality

For this phase of the project, you must support the correct evaluation of the following types of Scheme expressions:

1. Boolean, integer, real, and string literals. (They evaluate to themselves.)
2. `(if test trueExpr falseExpr)`
3. `(let list-of-bindings body)`

You will need to implement the creation of new frames. Much more on this in a bit.

4. Variables. Now that you've implemented `let` and frames, you should be able to evaluate a bound symbol.

When you're done with this part of the project, you should be able to evaluate very simple programs like this:

```
(let ((x #t)) (if x 3 5))
```

This should evaluate to 3. Your programs won't yet be able to handle functions; that's coming later.

3. Core functionality

At the core of your interpreter, you will need to implement a function to evaluate Scheme code:

```
SchemeVal *eval(SchemeVal *tree, Frame *frame) { ... }
```

Given an expression tree and a frame in which to evaluate that expression, `eval` returns the value of the expression. I'll explain the "frame" in a moment. Here's a skeleton of my `eval` function, to give you an idea of where to start:

```
SchemeVal *eval(SchemeVal *tree, Frame *frame) {
   switch (tree->type)  {
     case INT_TYPE: {
        ...
        break;
     }
     case ......: {
        ...
        break;
     }
     case SYMBOL_TYPE: {
        return lookUpSymbol(tree, frame);
        break;
     }
     case CONS_TYPE: {
        SchemeVal *first = car(tree);
        SchemeVal *args = cdr(tree);

        // Sanity and error checking on first...

        if (!strcmp(first->s,"if")) {
            result = evalIf(args,frame);
        }

        // .. other special forms here...

        else {
           // not a recognized special form
           evalationError();
        }
        break;
     }

      ....
    }
    ....
}
```

4. Frames, and bindings

We will be discussing this, but here is a brief overview of how to evaluate Scheme code.

The `eval` function mentioned above takes a pointer to a "frame"; what is a frame? A frame is a collection of bindings. OK, what's a binding? A binding is a mapping from a variable name (i.e. a symbol) to a value. Frames are created whenever we introduce new variable names. For example, in the program

```
(let ((x 5) (y "hello")) (if #t x y))
```

the bindings for `x` and `y` are stored in a single frame. You will have to construct a frame whenever `eval` encounters a `let`. This frame should be passed in when calling `eval` on the body of the let expression. The frame is used to *resolve* (find the value of) each variable encountered during evaluation. When `eval` tries to evaluate `x`, it looks in the current *frame* to find a value for `x`.

There's one other crucial detail here: frames can be chained together to create a larger environment consisting of multiple frames. For example, in the program

```
(let ((x "hello")) (let ((y "goodbye")) (if #t x y)))
```

each of the two let expressions creates a separate frame. When `eval` evaluates `x`, it first checks the innermost let's frame; since that frame only contains a binding for `y`, it then checks the outer let's frame, where it finds a binding for `x` with value `"hello"`.

To summarize, evaluating a let expression of the form:

```
(let ((var1 expr1) (var2 expr2) ... (varn exprn)) body)
```

consists of the following steps:

1. Let `e` be a pointer to the current frame. Create a new frame `f` whose parent frame is `e`.
2. For i = 1 through n:
   - Let `vali` be the result of evaluating `expri` in frame `e`.
   - Add a binding from `vari` to `vali` to `f`.
3. Evaluate `body` using frame `f` and return the result.

You will need to implement data structures for frames and bindings. The easiest approach is to use linked lists. The linked list of frames essentially forms a stack: you push on a new frame just before evaluating the body of the let expression, and pop the frame off before returning (although the "pop" really happens automatically when you return from `eval`). Within each frame you should store a list of variable/object pairs (i.e. bindings), using another linked list. When you need to resolve a variable, check the current frame first, then its parent if that variable is not in the current frame, then its parent, and so on until you reach a frame with no parent.

5. Special cases

For `(if test ...)`, any value for `test` (even a non-Boolean value) is considered true, unless it is `#f`.

For the specific form `(if test conseq)` where the alternative is left out..

- If `test` is `#f`, then error.
- If `test` is otherwise, then return the result of evaluating `conseq` as expected.

The form `(if test)` with no other parameters should also produce an error.

For `(let (...) )` where let contains a list of bindings but is missing a body, produce an error.

6. Evaluation errors

There are going to be many cases in which evaluation is not possible. For example:

- When an if expression has too many arguments
- When the `list-of-bindings` for let contains something that isn't a nested list
- When you encounter a variable that is not bound in the current frame or any of its ancestors
- etc…

In each of these cases you may immediately quit, printing "evaluation error". Make sure to clean up memory on your way out; use `texit` for this.

In any of these cases, you should print "Evaluation error" at the start of your error message so that the tests will pass. You might enhance these messages by adding more text to it, like "Evaluation error: if has too many arguments" or the like.

7. Multiple S-expressions

As discussed in the parser assignment, a Scheme program is a list of S-expressions. You should call `eval` on each S-expression. This is what the function `interpret`, that you need to write, should do: it is a thin wrapper that calls `eval` for each top-level S-expression in the program. You should print out any necessary results before moving on to the next S-expression.

This is also true for `let` itself. If the body of `let` has multiple expressions, you should evaluate each one of them, and return the result of the last one. See the last test below for an example of this.

8. Sample output

```
$ cat test-in-01.scm
3
5
(if #f 7 12)
$ ./interpreter < test-in-01.scm
3
5
12

$ cat test-in-03.scm
(let ((foo "hello")
      (bar 7)
      (baz #f))
  (if baz foo bar))
$ ./interpreter < test-in-03.scm
7

$ cat test-in-04.scm
(let ((x 3)) 2 4 x)
$ ./interpreter < test-in-04.scm
3
```

9. Your repository and your files

Your repository will follow the same structure that it has for the previous assignments. The key changes are in `schemeval.h`, which now includes a struct for a frame, and `interpreter.h`, which is the header file for the new functionality. You'll need to add `interpreter.c`, which implements the functions specified in `interpreter.h`.

Again as usual, you may use my binaries if you wish instead of using your own previous code, and the directions on how to do so are the same. I continue to encourage you enthusiastically to use your own code!

Building and testing your code will work precisely the same as on the previous assignment.

10. What to submit

Follow the instructions for the last assignment: submitting should be the same.

Have fun interpreting!

11. Preparing for celebrations of knowledge

The celebration of knowledge may ask you to write on paper C code to solve problems similar to the above. It might have you write the code from scratch; it might have you fill in some blanks in partially written code; or it might use some other method for having you demonstrate that you can write code. To prepare, you should practice writing C code on paper that can solve any of the above functions, or others like it.

If you're looking for more examples to practice:

- Implement the Scheme function `display` (described further down in this section of Dybvig). Don't worry about all of the edge cases, but here are two typical examples:

```
(display x)  ;; displays value of a particular variable
(display "hello world") ;; displays a particular string without the quot
```

- Implement the Scheme functions `when` and `unless` (described further down in this section of Dybvig).

This assignment was originally created by David Liben-Nowell and has since been updated by Dave Musicant, Jed Yang, and Laura Effinger-Dean.