Interpreter, Define / Lambda

For this portion of the project, you'll implement both `define` and `lambda`. When done, your interpreter will run functions that you create yourself, in Scheme!

This assignment is to be done individually. You can talk to other people in the class, me (Dave), and any of the course staff (graders, lab assistants, teaching assistants, prefects) for ideas and to gain assistance. You can help each other debug programs, if you wish. The code that you must be your own, however, and you are not permitted to share your code with others, not even your partner from other portions of the course. See the course syllabus for more details or just ask me if I can clarify.

1. Get started

Download this 07-lambda.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:

2.1. `(define var expr)`

Scheme actually has a number of different variations on `define` ; you only need to implement the first one, which is how we've used it. Unlike let, define does not create a new environment frame; rather, it modifies the current environment frame. You will therefore need to have a "top" or "global" frame which contains bindings of variables created using `define`. You can put bindings for primitive functions in this top-level frame too; you'll read more about that later.

You should not print anything after a define statement. You should go about this by having `define` return a special `SchemeVal` with type `VOID_TYPE`, and only print the result of an expression if it is not `VOID_TYPE`. There's probably other ways of doing this too.

`define` can be used at any level of nesting, and it simply modifies whichever frame is currently active.

The Scheme standard is somewhat messy regarding whether you can redefine a variable that you have already defined. The details of exactly what works and when is somewhat tricky, and so we're going to simply that specification for our purposes. For our interpreter, `define` should always fail with an evaluation error if you try to redefine a symbol that has already been defined in the current frame. It is perfectly acceptable to redefine a symbol in the current frame if it has only been defined in parent frames.

2.2. `(lambda params body)`

You will need to implement closures. Specifically, for purposes of this project a closure is just another type of object, containing everything needed to execute a user-defined function: (1) a list of formal parameter names; (2) a pointer to the function body; (3) a pointer to the environment frame in which the function was created. Use a `CLOSURE_TYPE` within `schemeval.h`. Look inside `schemeval.h` to see how it is defined.

2.3. Function application: `(e1 ... en)`

You should recursively evaluate `e1` through `en` and then apply the value of `e1` to the remaining values. The section below on function application has more details.

As usual, your program should never crash or segfault on bad input; just print "Evaluation error" and exit gracefully. You can supplement with more detailed error information if you wish.

Once you've finished these components, your interpreter should be able to evaluate user-defined functions, even those which are recursive. Here's one non-recursive example for you to try:

```
$ cat test-in-01.scm
(define not
  (lambda (bool)
    (if bool #f #t)))

(define testit
  (lambda (cond conseq alt)
    (let ((nconseq (not conseq)) (nalt (not alt)))
      (if cond nconseq nalt))))

(testit #t #f #t)
(testit #f #f #t)
$ ./interpreter < test-in-01.scm
#t
#f
```

3. Function application

In addition to `eval` as described above, you will need to implement function application. To do this, create a function called `apply`:

```
SchemeVal *apply(SchemeVal *function, SchemeVal *args);
```

You should call this function once you've evaluated the function and the arguments. Here, `function` is a user-defined function (i.e. a closure). You should take the following steps in order to execute the function:

- Construct a new frame whose parent frame is the environment stored *in the closure*.
- Add bindings to the new frame mapping each formal parameter (found in the closure) to the corresponding actual parameter (found in `args`).
- Evaluate the function body (found in the closure) with the new frame as its environment, and return the result of the call to `eval`.

Here's my new `eval` skeleton, modified to support function application:

```
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(expr);
        SchemeVal *args = cdr(expr);

       // Sanity and error checking on first...

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

        // .. other special forms here...

        else {

            // If not a special form, evaluate the first, evaluate the args, t
            // apply the first to the args.
            SchemeVal *evaledOperator = eval(first, frame);
            SchemeVal *evaledArgs = evalEach(args, frame);
            return apply(evaledOperator,evaledArgs);
        }
        break;

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

If a function is evaluated but not applied, you should just output `<#procedure>`. Here is an example:

```
$ cat test-in-02.scm
(lambda (x) x)
$ ./interpreter < test-in-02.scm
#<procedure>
```

4. Your repository and your files

Your repository will follow the same structure that it has for the previous assignments. There should be no changes at all in the files that I provide; you'll then need to add in the files from your previous version, and/or switch over to using my binaries. But note that the binaries I provide only go through part 4 of the project; I am not providing a working if/let interpreter. (The project gets too intermingled at this point for me to be able to supply complete partial work.)

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

5. What to submit

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

Have fun interpreting!

6. 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.

Additionally, this assignment heavily addressed the structure of frames and closures. Therefore, you might also be asked questions regarding when frames and/or closures are created, how they point to each other, and how that changes over the course of the program. You should be able to draw or interpret memory diagrams that utilize frames and closures, and be able to properly describe how they all fit together in the context of function creation and evaluation.

If you're looking for coding examples to practice:

- Implement the third version of define shown in Dybvig, which is the one that lets you use `define` alone as shorthand for both `define` and `lambda`:

```
(define (myfunction x y z)
   (+ x y z))
```

The above works, but is really shorthand for

```
(define myfunction
    (lambda (x y z)
       (+ x y z)))
```

- Implement a variation of lambda that takes a single parameter, not in parentheses, that gets bound to a list of parameters. This is described in this section of Dybvig, in the second bullet describing how *formals* works. Here's an example:

```
$ cat test-in-02.scm
(define tryit
  (lambda x
    x))

(tryit 1 2 3)
$ ./interpreter < test-in-02.scm
(1 2 3)
```

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