# Interpreter, Last portion

This assignment is the final phase of your interpreter! We'll be implementing a few more pieces of Scheme aimed at hopefully succeeding at the devious "Knuth test" (test-files-e/test61.scm).

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.

Under college policy, I may not accept this assignment if it has been submitted after the last day of classes under exceptional circumstances.

Also by college policy, I may not grade any work under any conditions that has been submitted beyond the end of the last final exam time. Any requests for an extension past then must be submitted via an application to the Dean of Students office. That application states that personal extensions such as these will only be granted for "personal reasons such as illness or unusual circumstances beyond the student's control."

## 1. Get started

Download this 09-final.zip file, and extract the zip file to your ProgrammingLanguages folder that you've been working in.

## 2. Functionality

Required additions to your interpreter are as follows:

## 2.1. Some missing special forms

Implement the following special forms: `letrec` and `set!`.

- `letrec`: letrec allows recursion in functions defined in a let. Take a look at the specification in Dybvig. The key difference from let is that the bound variables can be accessed from other variables bound in the same let, but unlike let*, the values can't be accessed until all expressions have been evaluated. For instance, consider:

```
(letrec ((x 3) (y x))
    x)
```

This violates the description in Dybvig that "The order of evaluation of the expressions expr ... is unspecified, so a program must not evaluate a reference to any of the variables bound by the letrec expression before all of the values have been computed." The specification says that if this restriction is violated, an exception (i.e., evaluation error) should be raised. However, Guile doesn't handle it correctly! It prints 3. Your interpreter should handle this correctly by raising an evaluation error.

Similarly, consider:

```
(define y 5)
(letrec ((x y) (y x))
  x)
```

Again, the value of a variable may be accessed before all variables are bound, so your interpreter should raise an evaluation error. This differs from Guile, which appears to print nothing in response to this.

Still not quite sure how to interpret `letrec`? Here's a rough guide to evaluating `(letrec ((x1 e1) ... (xn en)) body1 ... bodyn)`, although it doesn't include all needed error checking:

1. Create a new frame env' with parent env.
2. Create each of the bindings, and set them to `UNSPECIFIED_TYPE` (this is in `schemeval.h`).
3. Evaluate each of e1, …, en in environment env'. If any of those evaluations use anything with `UNSPECIFIED_TYPE`, that should result in an error.
4. After all of these evaluations are complete, replace bindings for each xi with the evaluated result of ei (from step 2) in environment env'.
5. Evaluate body1, …, bodyn sequentially in env', returning the result of evaluating bodyn.

- `set!`: Read the specification in Dybvig. This should be similar to `define`, except that it does not create a new variable; rather, it modifies an existing one.

### 2.2. One more primitive function

Implement `<` (i.e., the less-than function).

## 3. 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 following additional primitives and/or special forms: `let*`, `and`, `or`, `cond`, `list`, `append`, and `equal?`. `equal?` only needs to work for numbers, strings, and symbols; it does not need to perform any sort of deep equality test on a nested structure.

## 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. (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!

---

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