CS 360

Winter 2018

Programming Language Concepts

CS 360-001 Tuesday/Thursday 15:30-16:50 (UCross 151)

CS 360-002 Tuesday/Thursday 14:00-15:20 (UCross 151)

CS 360-003 Tuesday 18:30-21:20 (UCross 153)

Instructor: Geoffrey Mainland
mainland@drexel.edu (mailto:mainland@drexel.edu)
Office: University Crossings 106
Office hours: Mondays 4pm–7pm; Thursdays 5pm–6pm.

Teaching Assistant: Xiao Han
CLC office hours: Wednesday 2pm–4pm; Thursday 12pm–2pm

Lab 2: Introduction to the Metacircular Interpreter

Due Friday, February 1, 11:59:59PM EST.

Accept this assignment on GitHub Classroom here (https://classroom.github.com/a/vWJQLvab). The standard homework instructions (..) apply.

You may work with 1 partner on this assignment. You **must both acknowledge your partner** in your README.md file. You should both turn in the assignment by pushing to GitHub. Is is perfectly acceptable to turn in the same or similar code.

In this assignment, you will modify the metacircular interpreter we saw in class. Successfully completing this assignment requires reading and understanding a medium-sized program written by someone else. If you do not have a good understanding of how the interpreter works, please review the material covered in lecture and in the book. Diving straight in to the homework without this understanding is going to make your task much more difficult.

You should complete all problems by modifying the `README.md` and `mceval.rkt` files in your repository. Please **do not** create additional copies of `mceval.rkt` for the different parts of the assignment.

This assignment is worth 50 points. There are 55 possible points.

Working with the interpreter

There are then two ways you can test your evaluator evaluator:

1. Type `make` and run the resulting binary, named `mceval`. The `mceval` program will repeatedly read in a Scheme expression and pass it to your interpreter for evaluation.

2. From DrRacket, call the `top-mceval` function with an expression, like this:

```
(top-mceval '(+ 2 3))
```

I used the second approach. I also made judicious use of `display` and `newline` to debug my implementation.

To run the tests we provide, open the file `test-gui.rkt` in DrRacket and click the “Run” button.

Hints for solving the problems

There are three ways to extend the interpreter:

1. Add a primitive.
2. Add a definition to the global environment.
3. Add a special form.

You should only add a special form when it is absolutely necessary. Most of the time, the standard Scheme evaluation rules are exactly what you want. Solving a problem by adding a definition rather than a new special form is also much easier and avoids cluttering up your `mceval` function.

Use `display` and `newline` to print out intermediate expressions! This is extremely helpful when debugging.

Problem 1: Code Reading Questions (10 points total)

Submit the solutions to this problem in `README.md`.

Problem 1.1: Environment representation (2 points)

What representation does the metacircular evaluator use for environments?

Please be specific. An English description will suffice; however, your answer will be stronger if you also provide examples.

Problem 1.2: Defining the primitives (2 points)

What top-level `define` contains the list of primitives supported by the metacircular interpreter? Please name the variable.

Problem 1.3: Understanding primitives (2 points)

Eva Lu Ator and Louis Reasoner are each experimenting with the metacircular evaluator. Eva types in the definition of `map`, and runs some test programs that use it. They work fine. Louis, in contrast, has installed the system version of `map` as a primitive for the metacircular evaluator. When he tries it, things go terribly wrong. Explain why Louis’s `map` fails even though Eva’s works.

Problem 1.4: Understanding `mceval` (2 points)

Louis Reasoner plans to reorder the `cond` clauses in `mceval` so that the clause for procedure applications appears before the clause for assignments. He argues that this will make the interpreter more efficient: Since programs usually contain more applications than assignments, definitions, and so on, his modified `mceval` will usually check fewer clauses than the original `mceval` before identifying the type of an expression.

What is wrong with Louis's plan? (Hint: What will Louis's evaluator do with the expression `(define x 3)` ?)

Problem 1.5: Extending the environment (2 points)

The function `setup-environment` is used to create the initial global environment used by the metacircular interpreter. For later problems, it will be convenient to add your own definitions to the initial global environment. The most convenient way to do this is to call `eval-definition` with the appropriate arguments from within the function `setup-environment`. If you were to add a definition in this manner, what arguments would you pass to `eval-definition` to add the following top-level `define` to the initial global environment? You may give your answer in the form of a Scheme expression.

```
(define (not x) (if x false true))
```

Problem 2: Adding Primitives (10 points total)

Add the following primitives: `+`, `*`, `-`, `/`, `<`, `<=`, `=`, `>=`, `>`, and `error` to `mceval.rkt`. 1 point each.

The `error` primitive should take no arguments and abort the interpreter with the message "Metacircular Interpreter Aborted" (without the quotes).

Hint: You should use Racket's `error` function to raise an exception.

Problem 3: Implementing `and` and `or` (20 points total)

Add support for `and` and `or` to your interpreter (5 points each) by modifying `mceval.rkt`. Be sure your implementation adheres to the Scheme language standard (see here (http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-7.html#%_sec_4.2.1) for the relevant standard). Pay careful attention to how the arguments to `and` and `or` are evaluated and the value of the `and` or `or` expression.

You will probably want to use the `last-exp?`, `first-exp`, and `rest-exps` helper functions.

Remember that the metacircular interpreter cannot interpret `#t` and `#f`; use `true` and `false` instead.

Problem 4: Implementing `let` (10 points total)

Implement `let` expressions. Your implementation **must** create a new environment and evaluate the body of the let in this new environment. You will receive **zero credit** for a solution that does not create a new environment, i.e., you **may not** implement `let` using the rewrite-as-a-lambda-application technique described in lecture.

Pay careful attention to the following "features" of `let`:

1. You must evaluate the expressions that determine the values of the variables bound by the `let`.
2. The body of a `let` is a *sequence* of expressions.

If you are unclear on the semantics of `let`, you can read the language standard (http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-7.html#%_sec_4.2.2).

Section 1.3.2 of SICP (https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book-Z-H-12.html#%_sec_1.3.2) also contains an explanation of `let.`

WARNING: There are partial solutions to this problem on the web that use the rewrite-as-a-lambda-application technique. DO NOT copy these or any other solution you may find. Doing so will not only make me cry, but will also force me to report a violation of the university academic integrity policy as I am **required** to do both by departmental and university policy.

Problem 5: Homework Statistics (5 points total)

How much time did it take you to solve problems 1–4? Please tell us in your `README.md`. You may enter time using any format described here (https://github.com/wroberts/pytimeparse). Please enter times using the requested format, as that will make it easy for us to automatically collect. We are happy to read additional comments you have, but please put them *after* your recorded time. Here is the reporting format you should use:

```
Problem 1: 5m

Here's why this problem was easy.
```