Homework 3
Project 1 Part 1
CS 360
Summer 2021-2022

Geoffrey Mainland   Patrick Brinich

Due: 11:59:59pm EDT
Friday, July 15, 2022

Assignment Link

Please accept the assigment on GitHub Classroom Here: `https://classroom.github.com/a/cEOu4TyW`.

Homework Instructions:

You may work in groups of up to three for this assignment.

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 lecture material and text. Understanding the interpreter will make this assignment much easier and potentially shorten the amount of code you need to write.

You should complete all problems by modifying the `README.md` and `mceval.rkt`. files in your repository. Do not create additional copies of `mceval.rkt` for the different parts of the assignment. You may modify other files, including the test files, but we will grade only your `README.md` and `mceval.rkt` files.

Problems 1–4 require you to modify the applicative-order interpreter in `mceval.rkt`.

This assignment is worth 100 points. There are 120+10 possible points.

Working with the interpreter

There are three ways to test your evaluator:

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

2. From DrRacket, call `top-mceval` function with a *quoted* scheme expression, e.g. `(top-meceval '(+ 2 3))`.

3. From DrRacket, call `(main)`, type the expression you want to evaluate into the prompt, and hit enter.

Methods 2 and 3 allow you to use the GUI debugger to set breakpoints and step through code. Read the Graphical Debugging Interface section of the Racket manual for more information. If you're not using the GUI, debugging with `trace` as described here can be helpful. Also, `display` (and `newline`) can be used to print expressions if needed.

Running the test suite

To run the tests we provide using the GUI, open the file `run-tests-gui.rkt` in DrRacket and click the "Run" button. As with all other assignments, you may also run the tests via `make test`.

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.

Make sure you read Dr. Mainland's "Course Notes on the Metacircular Evaluator!" They tell you exactly how to implement a primitive, a top-level definition, and a special form.

It can be very useful to define helper functions that can be tested independently of the rest evaluator, especially when implementing a special form as a derived expression.

Problem 1: Adding Primitives (20 Points)

Add the following primitives `+`, `*`, `-`, `/`, `<`, `<=`, `=`, `>=`, `>` and `error`.

Your error primitive should take no arguments and abort the interpreter with the message “Metacircular Interpreter Aborted” (without the quotes). You should use Racket’s error function to raise an exception like this:

```
(error "Metacircular Interpreter Aborted")
```

Problem 2: Implementing `and` and `or` (30 Points)

Add support for `and` and `or` to your interpreter. Be sure your implementation adheres to the R5RS standard. Pay careful attention to how the arguments to and and or are evaluated and the value of the and or or expression—the language standard describes a recursive procedure for evaluating and and or expressions. Also make sure you evaluate each subexpression at most once.

Problem 3: Implementing `let` as its own special form (30 Points)

Implement let expressions. Your implementation must *explicitly create an environment* and evaluate the body of the let in this new environment. Do not use the technique that implements a derived expression using the rewrite-as-a-lambda-application technique. Your solution should not need to call either make-procedure or mcapply Pay careful attention to the following details of the semantics of let:

1. The expressions that determine the values of the variables bound by the let must be evaluated.

2. The body of a let is a *sequence of expressions*.

Please check these additional resources about let if you’re unsure:

1. Week 02 Lecture Slides

2. The R5RS standard

3. Section 1.3.2 of SICP

Problem 4: Implementing `force` and `delay` (40 Points)

Add support for `force` and `delay` to your interpreter, where delayed expressions are only evaluated once when forced.

For full credit, a delayed expression must be evaluated at most once. If your implementation evaluates a delayed expression every time it is forced but is otherwise correct, you will receive half credit. Do not submit two implementation of force and delay.

It is highly recommended that you implement the non-memoizing version of force and delay first. If you get stuck on the memoizing version without having implemented the non-memoizing version, stop what you're doing and get the non-memoizing version working first.

The implementations of both the non-memoizing and memoizing versions of force and delay were discussed in lecture. The easiest path to success will follow that discussion. Although force and delay can be implemented using thunks, that approach is difficult to get right.

Your solution must demonstrate that you understand the underlying mechanisms for implementing lazy evaluation. Therefore, you may not use Racket's force and delay or equivalent syntax macros in your solution to this problem. The homework template is set up to prevent you from accidentally using Racket's force and delay.

If you have successfully implemented force and delay using the technique from lecture, then your evaluator should evaluate `((delay 5))` to 5. However, force and delay are not identity functions! Although `(force (delay e))` will have the same value as the expression `e`, implementing both force and delay as the identity function is incorrect. Similarly, if `(delay (error))` throws an error, your implementation is not correct; delay must delay evaluation.

Hint: Write a function `delay->lambda` that rewrites a delay expression as described in lecture, leading to an implementation of delay as a derived expression. Test your `delay->lambda` independently to make sure it is performing the rewrite correctly, e.g., `(delay->lambda '(delay 3))` should evaluate to `(lambda () 3)`.

Problem 5: Homework Statistics (10 Point Extra Credit)

How long did spend on each problem? Please tell us in your `README.md`.

You may enter time using any format described here. 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 do not put any comments on the same line as the time report. Please do not otherwise edit, move, or reformat the lines reserved for time statistics reports. Here is the reporting format you should use:

```
Problem 1:  15m

Here's a description of my experience solving this problem.
```

If you did not attempt one or more problems, you can still receive full marks by telling us you spent 0 minutes on these problems. You will not receive points for this problem if you leave any of the time reports blank.

Course Software on TUX

1. Append the following line to your `.bash_profile` file (in your home directory) on TUX to use the correct versions of DrRacket and GHC:

   ```
   source ~mainland/courses/software/cs360.sh
   ```

2. Log out of your tux account and then log back in.
3. Use the `which -a` command to verify everything is set up:
   - `which -a racket` should output something like `/home/mainland/courses/software/bin/racket`
   - `which -a ghc` should output something like `/home/mainland/courses/software/bin/ghc`

Development outside of TUX

You can find the version of racket used for the course here: `https://download.racket-lang.org/releases/8.3/` You're free to develop your solutions in any environment, but please make sure your solutions run on TUX.

Docker containers for VS Code

Every homework repository is set up to work with development containers if you use Visual Studio Code. If you install Docker, then the "Remote - Containers" extension can automatically launch an appropriate Docker container and re-open your repository in the container, giving you a Linux environment in which to work no matter what computer you are using. See the Development Containers in Education post on the Visual Studio Code blog for more details. To use development containers:

1. Install Docker Desktop: `https://www.docker.com/products/docker-desktop`
2. Install Visual Studio code.
3. Install the "Remote - Containers" extension in VS Code.
4. Clone your repository as usual.
5. Open your cloned repository in Visual Studio Code. It should ask you if you want to re-open it in a container—click "Reopen in Container." At any time, you can also open the command palette in VS Code and run the command "Remote-Containers: Open Workspace in Container" to re-open your workspace in a container.