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

Homework 2: Working with the Metacircular Interpreter

Due Friday, February 8, 11:59:59PM EST.

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

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`, `mceval.rkt`, and `lazy-mceval.rkt` files in your repository. Please **do not** create additional copies of `mceval.rkt` or `lazy-mceval.rkt` for the different parts of the assignment.

Problems 1 and 2 require you to modify the applicative-order interpreter in `mceval.rkt`.

Problems 2 and 3 require you to modify the lazy interpreter, `lazy-mceval.rkt`.

This assignment is worth 50 points. There are 65 possible points.

Problem 1: Implementing `force` and `delay` (20 points total)

Add support for `force` and `delay` to your interpreter, where `delay` 'ed expressions are only evaluated once when `force` 'd.

For full credit, you must support call-by-need evaluation, which only evaluates `delay` 'ed expressions once.

If your implementation is call-by-name but otherwise correct, you will receive 10 points. You do not need to submit both a call-by-name and a call-by-need implementation for full credit; just the call-by-need version will do.

It is *highly* recommended that you implement the call-by-name version first. If you get stuck on the call-by-need version without having implemented the call-by-name version, stop what you're doing and get the call-by-name version working first.

We recommend you use memoization to implement call-by-need `force` and `delay` .

The implementations of both the call-by-name and call-by-need versions of force and delay were discussed in lecture. The easiest path to success will follow that implementation. If you have successfully implemented `force` and `delay` using the technique from lecture, then your evaluator should evaluate `((delay 5))` to `5` (why?).

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 Problem 1. The homework template is set up to prevent you from accidentally using Racket's `force` and `delay` .

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 and error, your implementation is not correct and will score 0; `delay` **must* delay evaluation.

Problem 2: Implementing streams (10 points total)

Add support for the following stream functions to your interpreter:

1. `stream-cons` (4 points)
2. `empty-stream` (1 points)
3. `stream-empty?` (1 points)
4. `stream-first` (2 points)
5. `stream-rest` (2 points)

You should be able to complete this problem with either the call-by-name or call-by-need implementation of `force` and `delay` . That is, you can receive full credit for this problem even if you did not receive full credit for Problem 5.

Your streams should be strict in the head of the stream and lazy in the tail. Note that Racket streams are lazy in **both** the head and the tail.

Your implementation must use your `force` and `delay` from Problem 5. You may not use Racket's `force` and `delay` or equivalent syntax macros in your solution to Problem 6.

Problem 3: Extending the lazy evaluator (10 points total)

For this problem, you will modify the **lazy** evaluator in `lazy-mceval.rkt` .

Add your primitives (Problem 1) and `and` and `or` (Problem 2) to the lazy evaluator. You should *almost* be able to copy the code over as-is. Be careful with `and` and `or` (why?).

Problem 4: A Hybrid Evaluator (20 points total)

For this problem, you will modify the **lazy** evaluator in `lazy-mceval.rkt`.

The original metacircular evaluator in `mceval.rkt` implements *applicative-order evaluation*—arguments are evaluated before a function is called. The lazy evaluator in `lazy-mceval.rkt` implements *normal-order evaluation*—arguments are evaluated when they are needed. For this problem, you will modify the lazy evaluator to implement a **hybrid** approach: the *definition* of each function will specify which arguments are delayed and which are evaluated immediately. To illustrate, consider the following function we saw in lecture:

```
(define (try a b)
  (if (= a 0) 1 b))
```

Evaluating `(try 0 (/ 1 0))` will produce an error in the applicative-order interpreter. The problem is that the applicative-order interpreter evaluates all function arguments, even when they are not needed. Imagine we could write this instead (this **is not** standard Scheme!)

```
(define (try a (delayed b))
  (if (= a 0) 1 b))
```

Where `(delayed b)` is an annotation to change the way compound procedures are applied as follows:

1. First evaluate the operator.
2. For each argument, if the corresponding parameter is annotated with `delayed`, delay its evaluation.
3. Otherwise, evaluate it.

In the case of `try`, this would mean that `a` is evaluated and `b` is delayed.

Modifying the interpreter to support this extension takes very little code, but it requires that you understand the metacircular interpreter and lazy evaluation.

Problem 5: Homework Statistics (5 points total)

How long did spend on problems 1–4? Please tell us in your `README.md`. You may enter time using any format described here (https://github.com/wroberts/pytimeparse).