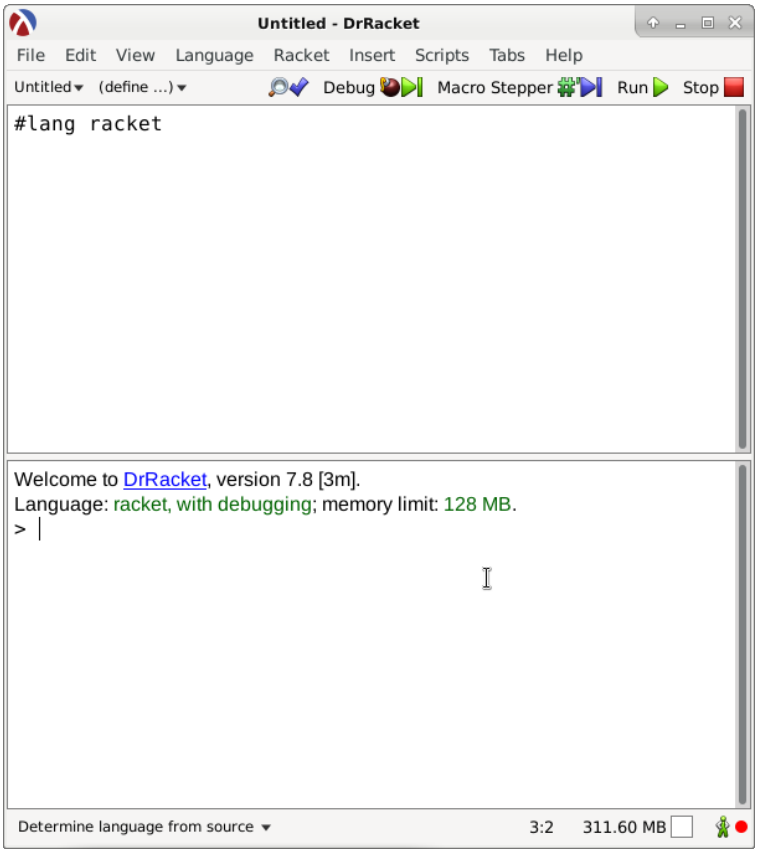
Lab #2 — Scheme Exercises
CS 152 Section 5 — Fall 2020
Michael McThrow
San José State University

Prelude – Installing and Using DrRacket

We will be using the DrRacket IDE in this course for running Scheme programs. DrRacket is available for Windows, macOS, and Linux. You can download it by going to https://racket-lang.org, clicking on the "Download" button on the top right of the page, and choosing the appropriate platform for your computer.

Once you have Racket installed, open DrRacket. Here is what DrRacket looks like on Linux; it should look similar on Windows and macOS:

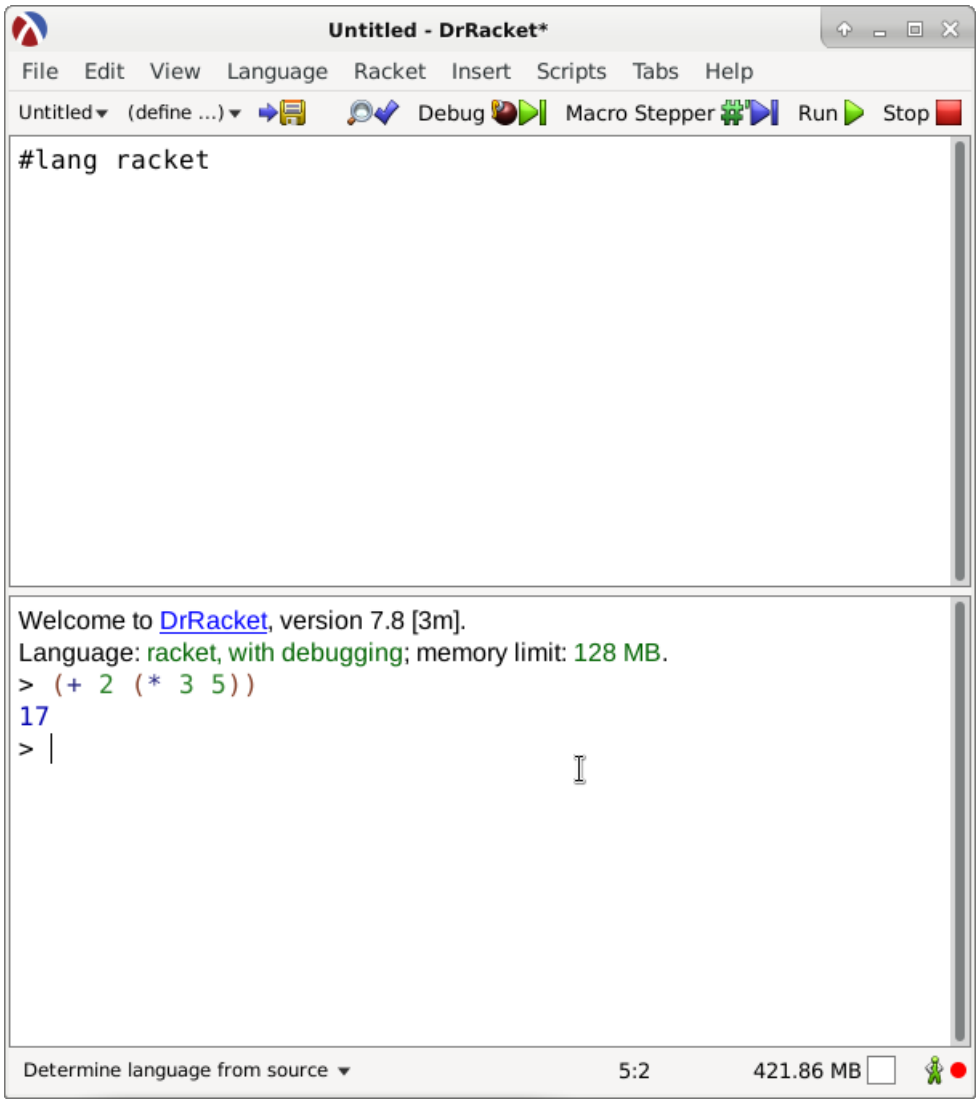


Note that the window has two sections. The first section where you see "`#lang racket`" is the source code file. The second section is the interpreter, which some programmers call the REPL, which stands for the "Read Eval Print Loop," which describes how the user interface for interactive Lisp interpreters is implemented. (This REPL terminology has been now widely applied for other interactive interpreters, such as those for Python and JavaScript.) When you write your programs, you're going to use the upper portion. However, if you just want to evaluate an expression on the fly, you can use the REPL on the lower portion.

Now, notice the "`#lang racket`" in the source code file section of DrRacket. That indicates that the program written below that line will be in the Racket programming language, which is a variant of

Scheme. For this assignment, however, we will be sticking to functions that are included in the R6RS version of Scheme. This enables us to use the REPL (sadly R6RS Scheme doesn't have REPL support implemented in DrScheme) and to make use of DrScheme's excellent debugging facilities.

Let's try out the REPL by evaluating `(+ 2 (* 3 5))`. Type "`(+ 2 (* 3 5))`" at the prompt (with no quotes) and press Enter.



When you write your code, you are going to use the upper portion of the window, and you are going to use the Run button to run your code, either the entirety of the source code or a selection of it.

For a more detailed tutorial of the DrRacket IDE, please visit the DrRacket IDE manual at https://docs.racket-lang.org/drracket/.

Warmup Exercises (40 points)

1. (10 points) Write a function named `fizz-buzz` that accepts no arguments. The fizz-buzz function outputs numbers 1 to 100 (inclusive) and prints them line-by-line as standard output. If a number is divisible by 3, we print "fizz" instead of the number. If a number is divisible by

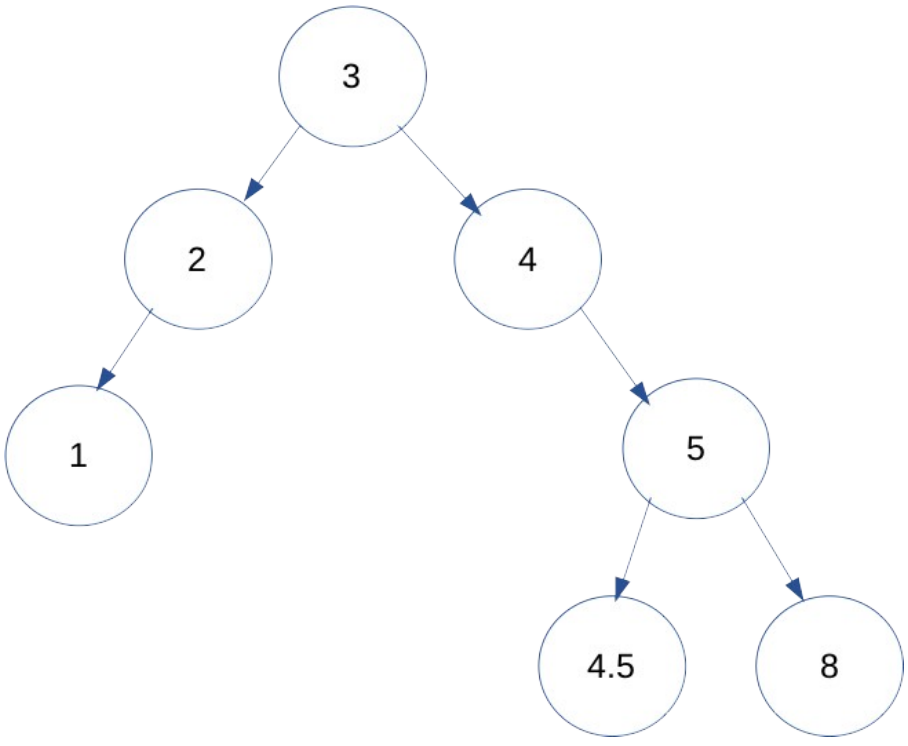
5, we print “buzz” instead of the number. If a number is divisible by both 3 and 5 (e.g., 15), we print “fizzbuzz” instead of the number.

Hint: Scheme provides a remainder function (e.g., `(remainder 3 2)` evaluates to `1`.) The display function accepts both numbers and strings. Note, though, that unlike Java’s System.out.println() function, display does not automatically append a new line.

2. (15 points) Write a function named `gcd` that accepts two arguments m and n. This computes the greatest common denominator of m and n; you may assume these are integers. For example, if m is 16 and n is 36, the greatest common denominator is 4. You may assume that m and n are integers.
3. (5 points) Write a function named `linear-search` that accepts two arguments: `items` and `key`. This function performs a linear search on the list items, determining whether key (an integer) is among those items. The function evaluates to true (`#t`) if the key is found, and false (`#f`) is the key is not found. For this function, when it comes to operating on the list, you are only allowed to use the Scheme functions `first` and `rest`; you may not use Scheme’s built-in search functions, nor are you allowed to take advantage of maps and filters.
4. (10 points) Write a function named `my-remove` that accepts two arguments: `items` (a list) and key. This function returns a new list that does not have any elements that match the key. Once again, you may only use `first` and `rest` for traversing the existing list, and for creating the new list, you are limited to the `cons`, `list`, and `append` functions.

Binary Search Tree (60 points)

We will be writing a binary search tree in Scheme. Recall from CS 146 that a binary search tree is a data structure that allows for the storage of elements in a manner that is amenable to performing searches in O(log n) in the average case. For example, suppose we are adding the elements 3, 4, 2, 5, 8, 1, and 4.5 (in that order) to construct a binary search tree. We will end up with the resulting tree:



In Scheme, we could represent the above binary search tree as a collection of embedded lists:

```
(3 (2 (1)) (4 () (5 (4.5) (8))))
```

Each node is a list with up to three elements. The first element of the list refers to the node's value. If a node is terminal (i.e., has no children), then it is represented as a one-element list. If a list has only two elements, that means the node only has a left child. If a node has only a right child, then it has three elements, but because there is no left child, the second element is the empty list `()`. A node with both a left and right child has three elements: the value, the left child, and the right child.

Let's do each insertion step by step:

```
3 => (3)
4 => (3 () (4))
2 => (3 (2) (4))
5 => (3 (2) (4 () (5)))
8 => (3 (2) (4 () (5 () (8))))
1 => (3 (2 (1)) (4 () (5 () (8))))
4.5 => (3 (2 (1)) (4 () (5 (4.5) (8))))
```

Note that no duplicate elements are allowed in the binary search tree.

Write the following functions as follows (note: you may assume that all elements in the binary search tree are numbers):

1. (22.5 points) `add-to-binary-search-tree` with two arguments `bst` (the binary search tree) and `item`, which is the item to be inserted into the binary search tree. This function returns a new binary search tree in its correct order. Note that if `item` is already in the binary search tree, then this function simply returns the existing binary search tree.
2. (7.5 points) `create-binary-search-tree` with argument `items`, a list of items to be inserted into the binary search tree. This function returns a new binary search tree with all of the items inside, excluding duplicates.
3. (15 points) `search-binary-search-tree` with two arguments `bst` (the binary search tree) and `key`, which is the item to search for in the binary search tree. This function returns `#t` if found and `#f` otherwise.
4. (15 points) `binary-search-tree-to-list` with two arguments `bst` (the binary search tree) and `traversal`, a symbol that indicates the traversal method. If the symbol is `preorder`, then place the elements in a list using preorder traversal. If the symbol is `inorder`, then use inorder traversal. If the symbol is `postorder`, then use postorder traversal.

Rules

- No side effects are allowed with the exception of the `display` function and other functions that print to the screen.
- No mutation is allowed; this includes the use of `set!`
- No `do` loops; either use recursion or map/filter/fold style functions (`for-all` is acceptable).
- Make sure all of your code cleanly runs when turning it in; if I cannot evaluate your function, that function gets a grade of zero.

- No external libraries or Racket-only libraries; use only the R6RS Scheme standard library.

Recommendations and Hints

- I highly recommend writing some test cases for your code. One of the nice things about coding without side effects is that you can just use simple equality functions to test what your functions evaluate to.
- Functional programming can sometimes feel like a mind-bending exercise for beginners. I highly recommend working out your programs on paper and thinking about how you would express with recursion, map, filter, and/or fold what you would normally express with loops in your Java programs. In some ways it's like learning how to code again. But don't feel discouraged; remember the first time you learned how to program, or the first time you learned how to write in assembly language. With practice, though, then functional programming will be second nature for you.
- Helper functions are vital in an immutable world. Remember that you can have as many arguments as you can in your helper functions.
- You are allowed to define your own top-level functions besides the ones that have been specified as common helper functions, but these functions are subject to the same restrictions as the others (for example, these functions can't take advantage of any features that the functions specified cannot take advantage of).

Instructions for Turning In the Assignment

Please place all of your functions in a file called `lab2.rkt`. This is the file that you will be turning in to me via Canvas.