Assignment 1: Intro to Racket & Haskell, Recursion, and Pattern Matching

CSC324H1, Fall 2024

Due Date: Friday, September 27, 2024 (11 pm)

In this assignment, you will be introduced to the two languages used in this course: Racket and Haskell. You will practice two concepts which are fundamental to most functional programming languages: Recursion and Pattern Matching.

One of the goals of this course is for you to become comfortable with looking up documentation independently for the basic syntax and built-in functions of new languages. To complete this assignment, in addition to the course materials, you may consult the documentation of the two languages and other online references.

In each of the tasks (except for `subst`), you will implement the same function in both Racket and Haskell. The logic behind both implementations will be the same, so after you implement each function in one language, implementing it in the other will mainly serve as a practice for the syntax. We recommend that you implement each function in Racket first and Haskell second.

Make sure to read the comments in the starter code for examples, details, and hints. You should complete the files `a1.rkt` and `A1.hs`. Run the tests provided to you to check your work. The Racket tests are in same file, but the Haskell tests are in `A1_Student_Tests.hs`. Note that your code will be graded based on additional hidden tests.

Task 1: Intro to Racket & Haskell — 20 pts

(a) `celsius-to-farenheit` (Racket and Haskell) — 10 pts

The function `celsius-to-farenheit` (in Haskell: `celsiusToFarenheit`) takes a temperature in degrees Celsius and converts it to farenheit, rounded to the nearest integer.

(b) `remove-second` (Racket and Haskell) — 10 pts

The function `remove-second` (in Haskell: `removeSecond`) takes a list, removes the second element of the list, and returns the resulting list. If the input list has less than two elements, it returns the list unmodified.

Task 2: Recursion — 45 pts

(a) `collatz` (Racket and Haskell) — 10 pts

Consider the following operation on a positive integer n:

- If n is even, divide it by 2. Specifically, return $n/2$.
- If n is odd, triple it and add 1. Specifically, return $3n + 1$.

The *collatz conjecture* is a well-known unsolved problem which claims that if we start with any positive integer n and apply the above operation repeatedly, we will eventually reach 1. For example, starting with 12, we get the sequence 12, 6, 3, 10, 5, 16, 8, 4, 2, 1. This conjecture has been verified for all integers up to 10^{20}, but not proven in general.

You will implement a function `collatz` which takes a positive integer n and returns the sequence described above, starting at n and ending at 1. The sequence starting at the input n is guaranteed to lead to a 1.

(b) `better-fibonacci` (Racket and Haskell) — 20 pts

The *Fibonacci sequence* is defined as follows:

$$f_n = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ f_{n-1} + f_{n-2} & \text{otherwise} \end{cases}$$

The function `fibonacci` takes a non-negative integer `n` and returns the `n`-th element of the fibonacci sequence. You are given a simple implementation of `fibonacci` in the starter code.

1. Asnwer the short-answer question in the *Racket file* labelled "QUESTION 1":

   `QUESTION 1: When calling (fibonacci 5), how many times is 'fibonacci' called (including the initial call and all recursive calls)?`

   Assign your answer to the variable `fibonacci-saq` as a number. See the Racket starter file for an example and a hint.

2. Implement `fibonacci` more efficiently, as `better-fibonacci` (`betterFibonacci` in Haskell). The key idea is to use a helper `fibonacci-helper`.

   The function `fibonacci-helper` (`fibonacciHelper` in Haskell) takes a non-negative integer `n` and returns a *pair*: the `(n-1)`-th and `n`-th elements of the fibonacci sequence. In other words, given input n, the helper returns (f_{n-1}, f_n).

   Note that for simplicity, please return $(0, 1)$ for $n = 0$.

3. Answer the short-answer question in the *Racket file* labelled "QUESTION 2":

   `QUESTION 2: When calling (better-fibonacci 5), how many times is 'fibonacci-helper' called?`

   Assign your answer to the variable `better-fibonacci-saq` as a number. See the Racket starter file for a hint.

Note that the short answer questions only need to be answered in the Racket file.

(c) `factorial-tail` (Racket and Haskell) 15 pts

The *factorial* of a positive integer n is defined as the product of all integers from 1 to n (or equivalently, from n to 1). You are given a simple implementation of `factorial` in the starter code. This implementation is *not* a tail recursion.

Your goal is to implement `factorial-tail` which computes the factorial of a given positive integer `n`. As indicated by the name, your implementation of `factorial-tail` *must* use *tail recursion*. See the starter code for a hint.

Task 3: Pattern Matching 35 pts

(a) `area-or-volume` (Racket and Haskell) 20 pts

We will define some syntax to describe a few basic "shapes". We will use different notations in the Racket and Haskell exercises.

Notation for Racket:

```
shape = (list 'circle <radius>)
  | (list 'triangle <base> <height>)
  | (list 'square <side>)
  | (list 'rectangle <width> <height>)
  | (list 'sphere <radius>)
  | (list 'cube <side>)
  | (list 'prism <base> <height>)
```

Notation for Haskell:

```
Shape = Circle <radius>
  | Triangle <base> <height>
  | Square <side>
  | Rectangle <width> <height>
  | Sphere <radius>
  | Cube <side>
  | Prism <base> <height>
```

The function `area-or-volume` will take a `shape` as input. If the `shape` is 2D (circle, triangle, square, or rectangle), it will return its area, and if it is 3D (sphere, cube or prism), it will return its volume. See below for area and volume formulas. **Assume $\pi = 3$.**

Circle with radius r:	$A = \pi r^2 \approx 3r^2$
Triangle with base b and height h:	$A = \frac{1}{2}bh$
Square with size a:	$A = a^2$
Rectangle with sides w and h:	$A = wh$
Sphere with radius r:	$V = \frac{4}{3}\pi r^3 \approx 4r^3$
Cube with side a:	$V = a^3$
Prism with height h and a base with area A:	$V = hA$

Note that the base of a prism can be any 2D shape (in our case: circle, triangle, square, or rectangle). The volume of a prism equals its height multiplied by the area of its base.

The Haskell exercise makes use of *algebraic data types* which will be covered in more detail later. For now, you only need to understand how to pattern match on these kinds of data types. See the function `shapeToText` in the Haskell starter code for an example.

(b) `subst` (Racket Only) **15 pts**

Consider the language described by the following syntax:

```
expr = ('λ (<id>) <body-expr>)
    | (<func-expr> <arg-expr>)
    | ('+ <expr1> <expr2>)
    | <id>
    | <int-literal>
```

The function `subst` will take an `expr`, an `id`, and a `val` (which is itself an expr) as input. It will substitute all free occurrences of `id` in `expr` with `val`, however it will leave bound occurrences of `id` in `expr` unchanged.

An occurrence of `id` in `expr` is bound if it is within the body of a λ-expr whose argument identifier is `id`. An occurrence of `id` in `expr` is free if it is not bound. In the following examples, free occurrences are red and bound occurrences are blue.

- `(λ (x) x)`
- `(λ (x) y)`
- `(λ (x) (+ x y))`
- `((λ (x) x) x)`
- `((λ (x) (λ (y) (+ x y))) (+ x y))`

Submission and Instructions

Submit the files `a1.rkt` and `A1.hs` to Markus. Make sure to complete all sections labeled "Complete me" in `a1.rkt` and "undefined" in `A1.hs`.

For all assignments:

- You are responsible for making sure that your code has no syntax errors or compile errors. If your file cannot be imported in another file, you may receive a grade of zero.
- If you do not intend to complete one of the functions, do not remove the function signature. If you are including a partial solution, make sure it doesn't cause a compile error. If your partial solution causes compiles errors, it's better to comment out your solution (but not the signature).
- Do not modify the function signatures provided in the starter code unless instructed. Changing the signature may cause your code to fail the tests.

- In Racket, you may not use any iterative or mutating functionality unless explicitly allowed. Iterative functions in Racket are functions like `loop` and `for`. Mutating functions in racket have an `!` in their names, for example `set!`. If you use the materials discussed in class and do not go out of your way to find these functions, your code should be okay.
- Do not modify the `(provide ...)` (in Racket) and `module (...) where` (in Haskell) lines of your code. These lines are crucial for your code to pass the tests.
- Do not modify existing imports or add new imports. Doing so may cause your grade to be deducted.