Assignment 2: Higher-Order Functions, Currying, and Evaluation

CSC324H1, Fall 2024

Due Date: Friday, October 11, 2024 (11 pm)

Make sure to read the comments in the starter code for examples, details, and hints. You should complete the files `a2.rkt` and `A2.hs`. Run the tests provided to you to check your work. The Racket tests are in same file, but Haskell tests are in `A2_Student_Tests.hs`. Note that your code will be graded based on additional hidden tests.

Task 1: Higher-Order Functions (Racket Only) 30 pts

A minimal arithmetic expression language is defined by the following grammar:

```
arith-expr =
    <number>
  | (Neg <arith-expr>)
  | (Plus <arith-expr> <arith-expr>)
  | (Times <arith-expr> <arith-expr>)
```

In this task, you are to define and use some higher-order functions for expressions of this minimal language.

(a) `map-expr` 10 pts

Implement `map-expr` for expressions, which takes the following as input:

- a function `f` that takes a number as input and returns a number, and
- an expression `expr`.

Function `map-expr` applies `f` to each number appearing in the expression and returns the resulting expression, which should have the same structure as the original one.

(b) `fold-expr` 20 pts

You are given an implementation of function `fold-expr`, which takes the following as input:

- a function `f-num` that folds a number,
- a function `f-neg` that folds `(Neg c)`,
- a function `f-plus` that folds `(Plus l r)`,

- a function `f-times` that folds `(Times l r)`, and
- an expression `expr`.

Function `fold-expr` folds the expression as described by the input functions and returns the result. Notice that the result type of `fold-expr` depends on what those input higher-order functions return. Read the implementation of `fold-expr` in the starter code to understand how this is done.

Implement two functions `expr-to-number` and `expr-to-list` that respectively fold expressions to numbers or their list representations using `fold-expr`:

- Function `expr-to-number` takes an expression as input and evaluates the arithmetic operations, then returns the resulting number.
- Function `expr-to-list` takes an expression as input and converts it to its list representation (or a number if it is just a number). See the examples in the starter code for the exact notation.

Note that these implementations should be very short and only call `fold-expr`. You are only allowed to fill in the appropriate inputs of `f-num`, `f-neg`, `f-plus`, and `f-times` in the calls to `fold-expr` to implement the two functions required. We will check this in your implementation.

Task 2: Currying (Haskell Only) 40 pts

(a) `curry`, `uncurry`, `curry3`, and `uncurry3` 20 pts

We can think of a function `f` with two inputs in two ways:

- **Uncurried**: `f` takes a tuple as input and returns a value. For example:

```
ucPlus ::  (Int, Int) -> Int
ucPlus = \(x, y) -> x + y     -- same as:  ucPlus (x, y) = x + y
```

- **Curried**: `f` is a higher-order function which takes a value as input (the "first" input) and returns a function which itself takes a value as input (the "second" input) and returns a value. For example:

```
cPlus ::  Int -> (Int -> Int)
cPlus = \x -> (\y -> x + y)
```

  This is how Haskell functions are defined by default, so we can write the example above more simply as:

```
cPlus ::  Int -> Int -> Int
cPlus x y = x + y
```

Implement four functions `curry`, `uncurry`, `curry3`, and `uncurry3`:

- `curry` takes as input an uncurried function and returns an equivalent curried function. For instance, `curry ucPlus` is functionally equivalent to `cPlus`.

  `(curry ucPlus) 2 3 == 5.`

- `uncurry` takes as input a curried function and returns an equivalent uncurried function. For instance, `uncurry cPlus` is functionally equivalent to `ucPlus`.

  `(uncurry cPlus) (2, 3) == 5.`

- `curry3` and `uncurry3` work similarly to `curry` and `uncurry` respectively, but for functions with three inputs.

(b) `zip`, `unzip`, and `zipWith` 20 pts

Implement two functions `zip` and `unzip` which convert a pair of lists to a list of pairs and vice versa. Additionally, implement `zipWith` which combines two lists into one list using a given function.

- `zip` takes two lists `xs` and `ys`, and returns a list of pairs `zs`. If `x` and `y` are the k-th elements of `xs` and `ys` respectively, the k-th element of `zs` is a pair `(x, y)`. If one input list is longer than the other, then the remaining elements in the longer list are ignored. For example:

```
zip [1, 2, 3] [‘a’, ‘b’, ‘c’] = [(1, ‘a’), (2, ‘b’), (3, ‘c’)]
zip [1, 2, 3] [‘a’, ‘b’] = [(1, ‘a’), (2, ‘b’)]
```

- `unzip` takes a list of pairs `zs` and returns a pair of lists `(xs, ys)`. If `(x, y)` is the k-th element of `zs`, then `x` and `y` are the k-th elements of `xs` and `ys` respectively. Moreover, `xs` and `ys` have the same length as the input list. For example:

  ```
  unzip [(1, 'a'), (2, 'b'), (3, 'c')] == ([1, 2, 3], ['a', 'b', 'c'])
  ```

- `zipWith` takes two lists `xs` and `ys`, and a function `f` which can combine elements of `xs` and `ys`, and returns a list `zs`. If `x` and `y` are the k-th elements of `xs` and `ys` respectively, the k-th element of `zs` is the result of calling `f x y`. If one input list is longer than the other, then the remaining elements in the longer list are ignored. For example:

  ```
  zipWith (+) [1, 2, 3] [2, 4, 6] == [3, 6, 9]
  ```

Task 3: Evaluation (Haskell Only) 30 pts

Recall the language from Assignment 1 defined by the following syntax:

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

In class, we learned about evaluation models based on **substitution** and **closures** and their implementations in Racket. In this task, you are to write functions that evaluate expressions of this language in Haskell based on two evaluation models.

See the starter file for detailed semantics of each function for each subtask.

(a) `evalEagerSubst` 15 pts

Implement `evalEagerSubst` in Haskell. For this subtask, we have provided you the data type definitions of expressions of this language and an implementation of the substitution function `subst` in Haskell. (We do not require capture-avoiding substitution.)

(b) `evalEagerEnv` 15 pts

Implement `evalEagerEnv` in Haskell. For this subtask, we have provided you the data type definitions of values of this language and an implementation of a simple map in Haskell.

Submission and Instructions

Submit the files `a2.rkt` and `A2.hs` to MarkUs. Make sure to complete all sections labeled "Complete me" in `a2.rkt` and "undefined" in `A2.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're not intending 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).
- 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 a `!` 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.