HOMEWORK 3: CS 4450

DUE: 3:00PM, 11/15/17

Directions

* Name your submission file as follows. For groups, submission files must be named `{groupname}_homework3.hs`. For example, a group named `reynolds` would submit `reynolds_homework3.hs`. For individuals, submission files must be named `{pawprint}_homework3.hs`. For example, an individual with a pawprint `tnrn9b` would submit `tnrn9b_homework3.hs`.
* Your submission *must* load and typecheck in the Haskell Platform to get any points.
* Every function you define must have a type signature in order to get full credit for the problem.
* Functions must be documented. Examples provided in the assignment file.
* Name all functions and data types exactly as they appear in the assignment. You cannot import any additional libraries or enable any additional language features unless granted explicit permission from the instructor.
* The code you submit must be your own. *Exceptions*: you may (of course) use the code we provide however you like, including examples from the slides.
* No late submissions! Don't submit the assignment hours after it is due and claim that Canvas wasn't working. If Canvas truly isn't working, you should submit the assignment via email minutes after it is due (3:00 PM) – not hours after it is due.
* Each problem must have at least one test (written by you) using the `Hspec` package. Make sure you have `Hspec` installed! You must name them `test_prob1`, `test_prob2`, `test_prob3`, and `test_prob4`, respectively; test individual problems, and `test_probs` to test all problems.

Background

Reverse Polish Notation (**RPN**) and Polish Notation (**PN**) are alternatives to the more commonly seen infix notation for arithmetic. Unlike infix notation, **RPN** and **PN** do not require conventions regarding the order of operations. Instead, the order of evaluation is dictated by the syntax. With **RPN**, operands are followed by their operators and evaluated accordingly. In this assignment, we will implement an **RPN** interpreter similar to what you might see in an HP calculator.

Here is the declaration for the language you will write the interpreter for:

```
data Op    = Val Int | Plus | Minus | Mul | IntDiv deriving (Show, Eq)
type PExp  = [Op]
```

Our operators (i.e., `Plus`, `Minus`, `Mul`, and `IntDiv`) and operands are both represented by the `Op` type. Whole arithmetic expressions in **RPN** are represented as lists of operations. Evaluation for **RPN** works by reading a stream of inputs from front to back. If a number (i.e., Val 5) is read, it (i.e., 5) is pushed onto a stack. If any operator is read, its operands are popped off of the stack, the operation is performed with them and the result is pushed back onto the stack. The topmost value on the stack becomes the rightmost argument to an operator. For example,the input "2 5 -" should evaluate to "-3". Correct computations in **RPN** result in an empty input and a stack with only one number value. Stacks with more than one value in them at the end of evaluation result from malformed input.

Problems

1. Write a function called `prob1`, that parses a `String` and returns a `PExp`. Input strings for this function are tokens that are either numbers or operators separated by whitespace. Numbers can be an arbitrarily long strings of digits.

Note: You do *not* need to validate that the input is a well-formed, **RPN** expression.

Hint: Converting String to other data types (like `Int`) can be done using the `read` function in Haskell. Review the documentation and examples of this function online to see how it works. Here are examples of how `prob1` should work:

```
Homework3*> prob1 "200 + - * /"
[Val 200,Plus,Minus,Mul,IntDiv]
Homework3*> prob1 "+ - * / 200"
[Plus,Minus,Mul,IntDiv,Val 200]
```

2. Write a function called `prob2`, that evaluates an **RPN** expression. This function should be typed as `PExp -> Int`. Cases of bad input and evaluation should result in a call to the Haskell `error` function.

Hint: You may need to define a helper function that takes a stack (represented using a list) and a `PExp` and returns an `Int`. Then, use the helper function to define `prob2`. Here are examples of how `prob2` should work:

```
Homework3*> prob2 [Val 4, Val 2, IntDiv]
2
Homework3*> prob2 [Mul]
*** Exception: Bad Input.
Homework3*> prob2 [Val 4,Val 0,IntDiv]
*** Exception: Cannot divide by zero!
Homework3*>
```

3. The evaluator crashes in cases where there are bad inputs and division by zero. This isn't particularly useful for recovery purposes. We can refactor the evaluator by using the data type, `Result a b`, to allow us to return a valid result or a failure indicator. Note the following code (from `RPNAST.hs`):

```
data RPNError    = DivByZero | BadSyntax deriving (Show, Eq)
data Result a b  = Failure a | Success b deriving (Show, Eq)
type RPNResult   = Result RPNError Int
```

By convention, the data type, `Result a b`, is either the failure case (i.e., `Failure a`) or the success case (i.e., `Success b`). Write a function called `prob3`, that has the type `PExp -> RPNResult`.

Hint: you may need to define a helper function that takes a stack and a `PExp` and returns an `RPNResult`. Use this helper function to define `prob3`.

Note: This problem will not simply encapsulate your work from Problem 2, the `error` function in Haskell (a hack of sorts) crashes the program and isn't catchable in a pure function. Here are examples of how `prob3` should work:

```
Homework3*> prob3 [Val 5, Val 0, IntDiv]
Failure DivByZero
Homework3*> prob3 [IntDiv, Plus, Val 0]
Failure InvalidInput
Homework3*> prob3 [Val 5, Val 1, Val 1, Plus, Mul]
Success 10
```

4. Write a function called `prob4` that takes a `PExp` and (given correct input) returns a `String` of an equivalent arithmetic expression in INFIX NOTATION, with the correct corresponding order of operations enforced using parentheses. This translation process is still prone to failure on bad inputs, so we should use a similar `Result a b` configuration (as in `prob3`), but instead of creating a special type to represent it, we will return a `String` in the failure case as well. Formulating the correct type signature of `prob4` is left to the student. Here are examples of how `prob4` should work:

```
Homework3*> prob4 [Val 1, Val 1, Plus]
Success "(1 + 1)"
Homework3*> prob4 [Val 2, Val 4, Plus, Val 3, IntDiv]
Success "((2 + 4) / 3)"
Homework3*> prob4 [Val 2]
Success "2"
Homework3*> prob4 [Plus]
Failure "Bad Input."
```

GRADING

Function	Points
prob1:	10
prob2:	10
prob3:	10
prob4:	10
Hspec Tests:	10
Total	**50**