Assignment 2, COMP3400, Semester 1 2020

Calculator

We're going to implement a calculator. This calculator will take in a string, representing an integer arithmetic expression, and give back an integer.

Grammar The calculator will allow:

- Addition
- Subtraction
- Multiplication
- Division
- Parenthesis for grouping

BNF style grammar:

```
Operator ::= '+' | '-' | '*' | '/'
Term ::= Number | '(' Term ')' | Term Whitespace Operator Whitespace Term
Number ::= ['-'] Digit+
Digit ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
Whitespace = ' '+
```

Arithmetic rules Use the BODMAS/PEMDAS order of arithmetic operators when calculating the evaluated answer. In our case the order is:

1. Parenthesis
2. Multiplication and Division
3. Addition and Subtraction

When operators have the same precedence and are not parenthesised, they are evaluated left-to-right. For example, the following expressions will produce the same result, regardless of precedence:

```
1 + 6 - 3 = 4
10 * 20 / 5 = 40
```

However, some expressions will produce different results, depending on associativity. This is due to both the order of operations, and the fact that we are doing integer division.

For example, the following expression will produce different results when evaluated left-to-right or right-to-left, because we are doing integer division:

```
10 * 20 / 30
```

Under otherwise normal arithmetic, the correct answer is 6.666… regardless of the order of operations. However, under integer arithmetic with left-to-right evaluation, the answer is 6 and for right-to-left evaluation, the answer is 0. For this assignment, we are doing left-to-right evaluation, and so the correct answer is 6.

Some expressions would produce different answers simply because the operation is not associative. Specifically, subtraction and division.

```
100 / 10 / 2
10 - 5 - 2
```

Since we evaluate these left-to-right, the correct answers are 5 and 3 respectively.

Parsing Produce a function:

```
parseExpression :: Parser Expression
```

This function should produce the `Expression` data type *with the correct association* of operators. You'll probably need to *reassociate* the syntax tree according to the order of precedence. This would require a function with the type:

```
reassociate :: Expression -> Expression
```

Evaluation If there's a divide by zero, the program's behaviour is undefined. For example, it could give back 0, 42 or let Haskell's division operator crash the program.

The interactive calculator program should keep asking for input and printing the results. Allow quitting the calculator when 'q' is entered.

Example inputs

```
ghci> runCalculator
>>> 3 / 2
1
>>> 1 + 2 + 3
6
>>> 1 + 2 - 3
0
>>> 1 - 2 + 3
2
>>> 1 + 6 / 2
4
>>> (1 + 2) * 3
9
>>> 1 + 2 * 3
7
>>> 1 - (2 + 3)
-4
>>> 10 * 20 / 30
6
>>> 100 / 10 / 2
5
>>> 10 - 5 - 2
```

```
3
>>> 10 * 20 / 5
40
>>> 1 ^ 2
Parse error: unexpected '^'
>>> q
```

Identity rules There are some simple algebraic laws that we can use to simplify a calculator expression. For example, if we add or subtract 0, we know that we can just use the other number. The following property tests demonstrate the **minimum** required identity rules that should apply to the `Expression`

Use these property tests for the simplification function in your API:

```
testAddRightIdentity :: String -> Bool
testAddRightIdentity s =
  simplify (Op expr Add (Number 0)) == simplify expr

testSubtractRightIdentity :: Expression -> Bool
testSubtractRightIdentity expr =
  simplify (Op expr Subtract (Number 0)) == simplify expr

testMultiplyLeftZero :: Expession -> Bool
testMultiplyLeftZero expr =
  simplify (Op (Number 0) Multiply expr) == Number 0

testMultiplyRightIdentity :: Expression -> Bool
testMultiplyRightIdentity expr =
  simplify (Op expr Multiply (Number 1)) == simplify expr
```

There are a few useful simplifications that are missing from the above list. Add similar simplifications and write property tests for them.

Import the test framework introduced during the lectures. Create a `genExpression :: Gen Expression` to execute the property tests.

Data Types and Functions

Data Types

```
data Operation =
  Add
  | Subtract
  | Divide
  | Multiply
  deriving (Eq, Show)

data Expression =
  Number Int
  | Op Expression Operation Expression
  | Parens Expression
  deriving (Eq, Show)

data ParseResult x =
  ParseError String
  | ParseSuccess x String
  deriving (Eq, Show)

data Parser x =
  Parser (String -> ParseResult x)
```

Functions These functions form the API of the calculator:

```
-- converts a string into an expression, or an error
parseExpression :: Parser Expression
-- removes very simple expressions
simplify :: Expression -> Expression
-- runs the property tests for simplify
runTests :: IO ()
-- takes an expression and reduces it to an answer
calculate :: Expression -> Int
-- runs the calculator interactively
runCalculator :: IO ()
```

Instances To gain some code reuse, add the following type-class instances:

```
instance Functor ParseResult

instance Functor Parser
instance Applicative Parser
instance Monad Parser
```

Assessment

- This assignment contributes to 25% of the total marks for COMP3400.
- The marks for this assignment out of 25 will be distributed as follows:
  - `parseExpression` function **7 marks**
  - `simplify` function with **2 marks**
  - `genExpression` function **3 marks**
  - `runTests` function **2 marks**
  - `calculate` function **2 marks**
  - `runCalculator` function **3 marks**
  - Type-class instances **2 marks**
  - Overall design, including helpful program comments **4 marks**

Submission

Submit your `.hs` file(s) to the assessment folder on Blackboard. If you have a bunch of files you can zip them up and submit them.