CSCI 2100
Functional Programming
Spring 2022
Exam 2
[40 points]

Policies

- It is an open-book, closed-person test.
- You are allowed to use any resource (books, tutorials, class notes) to figure out the solutions but you are **not allowed** to -
  - ask for solutions to any person
  - post the question to any group (online or physical) in order to get any kind of help
  - discuss Haskell-related issues/questions with anyone
- If you are stuck at any point, you are allowed to ask for help (email) from the course instructor. She will give you some clues.

- You have to answer all the questions to get full credit.
- You get partial credits for partially correct answers.
- You are not allowed to change any function names.
- Coversheet:
  - Write the following two statements on a paper, put the date and your signature on it, create a pdf/image file and save the file as **<Lastname>.pdf/jpeg/jpg**.
  - Statements:
    - `I neither have taken any help from anyone nor have provided help to anyone with the exam.`
    - `I do not know of anyone who has been involved in the test and has violated the policy.`
  - If either of the statements is not true, inform your professor about the incident.
  - **Remember:** Your submission will not be graded without the signed cover-sheet.
- Place all the relevant and necessary files in a folder. Name the folder as **CSCI2100Exam02Lastname**. Lastname should be replaced by your last name. Zip/compress the folder and upload.
- Expected files in the CSCI2100Exam2Lastname folder: **problem01.hs, problem02.hs, problem02_output.dat, problem03.hs, problem04.hs**, and the **coversheet**.

Problem 01: 10 points

Topic: Modules and **User-defined Types**
Source code: problem01.hs (update the source code as required)

Part a: 3 points

In **problem01.hs** file, we have a module **Problem01** which has one user defined data type, **Parent**.

```
bash-3.2$ ghci problem01.hs
[1 of 1] Compiling Problem01  (problem01.hs, interpreted )
Ok, one module loaded.

*Problem01> :browse
data Parent = Parent {father :: String, mother :: String}
*Problem01> p1 = Parent {father = "super dad", mother = "super mom"}
*Problem01> p2 = Parent {father = "spider man", mother = "super girl"}
*Problem01> p3 = Parent {father = "spider man", mother = "super girl"}
```

Task: Adjust the definition of the Parent datatype so that we can compare between different Parent type data.

Sample Testcases:

```
*Problem01> p1 == p2
False
*Problem01> p3 == p2
True
*Problem01> p4 = read "Parent{father=\"john\", mother = \"jill\"}"::Parent
*Problem01> p4
Parent {father = "john", mother = "jill"}
```

Part b: 4 points

Task: Define another data type, **Student**, in the **Problem01** module. Student has three fields: **name** (of type String), **gpa** (of type Float), and **parent** (of type Parent).

Save the definition and test your code with the following sample test cases:

```
*Problem01> s1 = Student {name = "Karl", gpa = 3.88, parent = Parent{father
= "A", mother = "K"}}

*Problem01> s2 = Student {name = "Mel", gpa = 3.98, parent = Parent{father =
"M", mother = "L"}}
```

Make sure that we can compare different fields of the students.

```
*Problem01> s1 == s2
False
*Problem01> s1 /= s2
```

```
True
*Problem01> gpa(s2) > gpa(s1)
True
*Problem01> parent(s2) == parent(s1)
False
```

Part c: 3 points

Define a function, **sibling**, that takes two Student type values (say, x and y) as input and returns True if the two type parameters (x and y) have at least one common field in the parent data (either the father/mother (or both) is (are) the same for x and y). Otherwise, it returns False.

Sample testcases:

```
*Problem01> s1 = Student {name = "George", gpa = 4.00, parent = Parent
{father = "A", mother = "D"}}

*Problem01> s2 = Student {name = "Melanie", gpa = 4.00, parent = Parent
{father = "A", mother = "C"}}

*Problem01> s3 = Student {name = "Jill", gpa = 4.00, parent = Parent {father
= "B", mother = "E"}}

*Problem01> sibling s1 s2
True
*Problem01> sibling s3 s2
False
```

Problem 02: 10 points

Topic: Files and IO Actions
Solution: problem02.hs (update the source code as required)

The provided function (**generateBalancedParen**) takes a positive integer as input and generates all possible strings of n-balanced parentheses. It also has a **main** block that just prints one line on the standard output stream (meaning, to our screen).

```
bash-3.2$ ghci problem02.hs
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( problem02.hs, interpreted )
Ok, one module loaded.
*Main> generateBalancedParen 0
[""]
*Main> generateBalancedParen 1
["()"]
*Main> generateBalancedParen 2
["()()","(())"]
```

Update the **main** block so that it implements the following algorithm:

```
   1. Ask the user to enter a number (say, n)
```

```
2. If n <= 10 and n >= 0 then
      a. Call the generateBalancedParen with n as input to create a
         list of all possible balanced parentheses.

      b. Open a file (problem02_output.dat) in append mode and
         write the information (n and the output list). The format
         is shown below.

      c. Go to the beginning of the main block.

3. If n is out of range (n<0 or n>10), then stop.
```

Sample input:

```
Enter a number (0~10, any value outside of the range will terminate
the program):
0
Enter a number (0~10, any value outside of the range will terminate
the program):
1
Enter a number (0~10, any value outside of the range will terminate
the program):
2
Enter a number (0~10, any value outside of the range will terminate
the program):
101
```

Sample output/ The expected file:

```
0 >> ""

1 >> "()"

2 >> "()()"
"(())"
```

Hint: You may find the worksheet problems (04/01) useful. Check the posted sample solutions if needed.

Problem 03: 10 points

Topic: Recursion and Higher order functions
Source code: problem03.hs (update the source code as required)

Part a)

Write a recursive function, **sumDigits**, that takes a positive integer as input and returns the summation of the digits from the input.

```
*Main> sumDigits 0
0
```

```
*Main> sumDigits 100
1
*Main> sumDigits 101
2
*Main> sumDigits 92
11
```

Part b)

Write expressions to get expected outputs.

i. Given a list of integers Z, write an expression to generate a list of integers X, where,

$$X = \{ x \mid x = sumDigits(z),\ z \in Z \}$$

Sample IO:

```
*Main> z = [1, 10, 112, 13, 99, 1000]
*Main> x = <expr> z
*Main> x
[1,1,4,4,18,1]
```

ii. Given a list of integers Z, write an expression to find a number t such that $t \in Z$ and $sumDigits(t) \geq sumDigits(z), \forall_{z \in Z}$

Sample IO:

```
*Main> z = [1, 10, 112, 13, 99, 1000]
*Main> t = <expr> z
*Main> t
99
```

Problem 04: 10 points

Topic: Recursion
Source code: problem04.hs (update the source code as required)

Infix, Postfix, and Prefix notations are three different but equivalent ways of writing expressions. In prefix expressions, operators are written before their operands. The benefit of having a prefix (or postfix) expression is that we do not need to use parenthesis for expressing the precedences among the operations we need to perform. Below is a table that gives us some ideas on prefix and infix notations.

Infix	Prefix
B + D	+ B D
A * (B + C) / D	/ * A + B C D
(A + B) / (C + D - E)	/ + A B - + C D E
(A + B) / (C + D - E) + F	+ / + A B - + C D E F

If you are unfamiliar with the notations, here is a source that might give you some more ideas: https://www.cs.man.ac.uk/~pjj/cs212/fix.html

The Haskell compiler (at least the GHC) can evaluate an infix expression. We will write a Haskell program to evaluate a prefix expression in this part. This means that our program will return an evaluated result given a valid prefix expression. For example: + 3 4 should return 7, - 3 4 should return -1.

Before we start, let's define some constraints:

i. We will only stick to binary operators for this task (like, addition, subtract, multiplication, division).
ii. In order to indicate the operators, we will use some codes (Add, Sub, Mul, Div) like the followings:
Add represents an addition,
Sub represents a subtraction,
Mul represents a multiplication,
Div represents a division
iii. The input values and the output will be of type Fractional (or just Double) as all other numeric **types** fall in the class **Fractional.**
iv. The initial input will be a String like this: "Add 3 Sub 4 5".
v. The output will be a number (a Double value).
vi. We may not worry about invalid expressions.
vii. Use the code given as problem04.hs and implement the `postfixExpressionEvaluator` function.

Sample IO

```
bash-3.2$ runhaskell problem04.hs
"Enter a prefix arithmetic expression (like Add 3 4):"
Add 3 4
7.0
"Enter a prefix arithmetic expression (like Add 3 4):"
Sub Add 2 3 Add 7 -9
7.0
"Enter a prefix arithmetic expression (like Add 3 4):"
Mul 3 Add 2 3
15.0
"Enter a prefix arithmetic expression (like Add 3 4):"
Div 1 Add 1 1
0.5
"Enter a prefix arithmetic expression (like Add 3 4):"
Div 2 Add 1 -1
Infinity
"Enter a prefix arithmetic expression (like Add 3 4):"

bash-3.2$
```