WESTERN UNIVERSITY

Computer Science 3342b - Winter 2019
Organization of Programming Languages

Assignment 4 - Functional Programming with Haskell
Due March 27, 2019 at 11:59 PM

1 Setup

On the class server, cs3342b.gaul.csd.uwo.ca, clone your repository (if you haven't already) and change to it:

```
git clone https://repo.csd.uwo.ca/scm/compsci3342\_w2019/USERNAME.git
cd USERNAME
```

If your repository was already cloned, change to it and pull down any pending commits:

```
cd USERNAME
git pull
```

From the **root** of your repository, run the setup script to create your assignment 4 directory and file structure, and to download the provided code:

```
curl -L https://bit.ly/2u4hjW4 | bash
```

This will create the following structure:

```
$ find asn4 -type f
asn4/q1/q1.hs
asn4/q2/q2.hs
asn4/q3/q3.hs
asn4/q4/q4.hs
asn4/q5/q5.hs
asn4/q6/q6.hs
asn4/q7/q7.hs
asn4/q8/q8.hs
asn4/q9/Customer.java
asn4/q9/CustomerManager.java
asn4/q9/CustomerTest.java
```

You should now commit and push your initial code:

```
git add .
git commit -m "Pushing initial assignment 4 code"
git push
```

You are expected to put the code from this assignment in the provided files. Failure to adhere to the directory structure and/or naming convention given will result in a **deduction of 10%** of your assignment grade. There will be no exceptions to this rule. This is because using non-standard directory or filenames hinders the automated tests that we use to ensure we get your assignments returned to you as quickly as possible.

2 Questions

1. *(4 marks)* In `asn4/q1/q1.hs`, write a function `mult` that multiplies two numbers **using only the addition operator**. Recall that multiplication can be expressed as follows:

$$a \cdot b = \begin{cases} 0 & \text{if } b = 0, \\ a + a \cdot (b - 1) & \text{if } b > 0 \end{cases}$$

   For full marks, your function must use guarded expressions and no unnecessary control structures (e.g., `if-else` statements) or helper functions. Marks are also allotted to the succinctness and elegance of your function.

   Sample invocation:

```
*Main> mult 4 3
12
```

2. *(4 marks)* In `asn4/q2/q2.hs`, write a function `pow` that, given arguments `a` and `n`, computes the result a^n **without using any of the built-in power operators or any library functions**. Recall the expression a^n can be expressed as follows:

$$a^n = \begin{cases} 1 & \text{if } n = 0, \\ a \cdot a^{n-1} & \text{if } n > 0 \end{cases}$$

For full marks, your function must use guarded expressions and no unnecessary control structures (e.g., `if-else` statements) or helper functions. Marks are also allotted to the succinctness and elegance of your function.

Sample invocation:

```
*Main> pow 2 10
1024
```

3. *(4 marks)* In `asn4/q3/q3.hs`, write a function `pow'` that computes the same result as the `pow` function above, but uses pattern matching instead of guards to compute its result. *Hint: recall that when* `_` *is used in place of a parameter name in a function signature, it matches any value.*

Sample invocation:

```
*Main> pow' 2 10
1024
```

4. *(8 marks) List comprehensions* are a powerful way to quickly build lists. In this question, we'll explore how we can build both finite and infinite lists using list comprehensions. Put the following in `asn4/q4/q4.hs`:

   (a) Define a variable `odds` storing a list of all odd numbers from `1` to `10`. Use a list comprehension to build this list.

   (b) Define a variable `positiveEvens` storing an *infinite list* of all positive, even numbers. Use a list comprehension to build this list.

   (c) Define a variable `powersOfTwo` storing an *infinite list* of the numbers $2^0, 2^1, 2^2, 2^3, \ldots$. Use a list comprehension to build this list.

   (d) Define a function `firstNPrimes` that takes an argument `n`. The function should:

   - Build an infinite list of primes using a list comprehension (*hint: look up* `where` *or* `let` *expressions*)
   - Return the first `n` primes (*hint: look up the* `take` *function*)
   - You may paste in and use the following (inefficient) function definition to determine whether a number is prime:

```
isPrime x
  |  x <= 1    = False
  |  otherwise = null [k | k <- [2..x-1], x `mod` k == 0]
```

Sample invocation:

```
*Main> odds
[1,3,5,7,9]
*Main> take 10 positiveEvens
[2,4,6,8,10,12,14,16,18,20]
*Main> take 10 powersOfTwo
[1,2,4,8,16,32,64,128,256,512]
*Main> firstNPrimes 10
[2,3,5,7,11,13,17,19,23,29]
```

5. *(10 marks) Insertion Sort* can be recursively expressed as follows: to sort $A[1..n]$, first recursively sort $A[1..n-1]$, then insert $A[n]$ into the sorted array $A[1..n-1]$. In this question, we will implement Insertion Sort in Haskell in `asn4/q5/q5.hs`.

   - Write a function `insert` that takes a number x and a sorted list of numbers ℓ. This function should return a list of numbers with x inserted at the correct location in ℓ, preserving its sorting.

     Sample invocation:

     ```
     *Main> insert 4 [1,2,3,5]
     [1,2,3,4,5]
     ```

   - Write a function `isort` that takes a list of numbers ℓ. Using recursion and the `insert` function, this function should return a list of numbers ℓ' sorted in ascending order using the recursive definition of insertion sort given above.

     Sample invocation:

     ```
     *Main> isort [20, 1, 22, 6, 17, 3, 13, 15, 10, 7]
     [1,3,6,7,10,13,15,17,20,22]
     ```

6. *(20 marks)* Insertion Sort has a worst-case time complexity of $O(n^2)$. We can do better. The pseudocode for a recursive *Merge Sort* is given below. Merge Sort has a worst-case time complexity of $O(n \cdot \log n)$.

   MERGE-SORT(A)

   ```
   // An array of length 0 or 1 is already sorted
   if A.length < 2
       return A

   // Find the middle of the array and recursively sort its two halves
   middle = ⌊A.length / 2⌋
   firstHalf = MERGE-SORT(A[0 .. middle − 1])
   lastHalf = MERGE-SORT(A[middle .. A.length − 1])

   // Merge the two sorted halves
   return MERGE(firstHalf, lastHalf)
   ```

   MERGE(A, B)

   ```
   if A.length == 0
       return B
   elseif B.length == 0
       return A

   x = A[0]
   y = B[0]

   // Compare the head of each list
   if x < y
       // If A has a smaller head, return its head concatenated with
       // the result of recursively merging the rest of A with B
       return [x] + MERGE(A[1 .. A.length − 1], B)
   else
       // If B has a smaller head, return its head concatenated with
       // the result of recursively merging the rest of B with A
       return [y] + MERGE(A, B[1 .. B.length − 1])
   ```

   This code is given in *imperative* pseudocode. Your job is to translate it to succinct, *functional*, Haskell code. Put the following in `asn4/q6/q6.hs`:

   (a) Translate MERGE to the Haskell function `merge`.

- This function takes a sorted array of size n, a sorted array of size m, and recursively merges them into a sorted array of size $n + m$.
- Favour a functional programming style that employs pattern matching, guards, etc. You should not need any `if` statements in your code
- *Sample invocation:*

```
*Main> merge [1,2,3,7,8,9] [2,3,4,5]
[1,2,2,3,3,4,5,7,8,9]
```

(b) Translate MERGE-SORT to the Haskell function `msort`.

- This function divides a list into halves, recursively merge sorts the two halves, and then merges them back together into a sorted list
- Favour a functional programming style that employs pattern matching, guards, etc. You should not need any `if` statements in your code
- You may find `where` or `let` expressions useful and elegant – look them up!
- You may find the `take` / `drop` or `splitAt` functions useful – look them up!
- Note: Haskell distinguishes between floating-point division (`/`) and integer division (`div`). You'll want the latter.
- *Sample invocation:*

```
*Main> msort [20, 1, 22, 6, 17, 3, 13, 15, 10, 7]
[1,3,6,7,10,13,15,17,20,22]
```

7. *(20 marks)* In `asn4/q7/q7.hs`, write a function `l33t` that converts strings to l33t-speak. For our purposes, l33t-speak will be defined as:

   - All consonants are converted to (or left as) uppercase
   - `e` and `E` are converted to the number `3`
   - `i` and `I` are converted to the number `1`
   - All other vowels are converted to (or left as) lowercase
   - An exclamation mark is translated to `!!!111oneone`
   - All other characters are left unchanged

   A sample call is shown below:

```
*Main> l33t "The QUICK brOwN fox JUMPS over the LAZY dog!"
"TH3 Qu1CK BRoWN FoX JuMPS oV3R TH3 LaZY DoG!!!111oneone"
```

   For this question, you may import and use the `ord` and `chr` functions, which are used to convert a character to an ASCII symbol and back:

```
*Main> import Data.Char (ord, chr)
*Main Data.Char> ord 'A'
65
*Main Data.Char> chr 65
'A'
```

   You may use no other built-in or third-party functions. It is recommended that you create helper functions (such as `isVowel`, `isUpper`, etc.) to ensure the succinctness of your `l33t` function.

8. *(20 marks)* Recall the digits of the hexadecimal number system (base 16):

Hexadecimal Digit	Decimal Equivalent
0	0
1	1
⋮	⋮
9	9
A	10
B	11
C	12
D	13
E	14
F	15

For an n-digit hexadecimal number $N_{16} = a_{n-1}a_{n-2}\ldots a_1a_0$, we can express its decimal equivalent N_{10} as follows:

$$\begin{aligned} N_{10} &= a_{n-1} \cdot 16^{n-1} + a_{n-2} \cdot 16^{n-2} + \ldots + a_1 \cdot 16^1 + a_0 \cdot 16^0 \\ &= \sum_{i=0}^{n-1} a_i \cdot 16^i \end{aligned}$$

For instance, converting the hexadecimal number $4E1F$, we have:

$$\begin{aligned} 4E3F_{16} &= 4 \cdot 16^3 + E \cdot 16^2 + 3 \cdot 16^1 + F \cdot 16^0 \\ &= 4 \cdot 16^3 + 14 \cdot 16^2 + 3 \cdot 16^1 + 15 \cdot 16^0 \\ &= 4 \cdot 4096 + 14 \cdot 256 + 3 \cdot 16 + 15 \cdot 1 \\ &= 16384 + 3584 + 48 + 15 \\ &= 20031_{10} \end{aligned}$$

In `asn4/q8/q8.hs`, write a recursive function `hexStrToDec` that takes a hexadecimal string and returns the equivalent value as a decimal integer. Your function must satisfy the following constraints:

- For full marks, your function must be as succinct as possible
- You may write one or more helper functions that are called by your function, if needed
- You may import and use any of the following functions if you find them useful:

  ```
  import Data.Char (ord, toUpper, toLower)
  ```

  You may not use any other functions in `Data.Char`
- Your function should handle:
  - Lowercase hexadecimal digits: `ffe`
  - Uppercase hexadecimal digits: `FFE`
  - A mix of lowercase/uppercase digits: `C0ffEE`
  - An optional `0x` prefix: `0xC0ffEE`
  - An optional `0X` prefix: `0XC0ffEE`
  - Leading zeroes: `00FFE`, `0x00FFE`, etc.
  - Invalid hex digits: `HELLO` (*throw the error* `"Non-hexadecimal digits present"` *using the* `error` *function*)

Sample invocation:

```
*Main> hexStrToDec "4e3F"
20031
*Main> hexStrToDec "ffe"
4094
*Main> hexStrToDec "FFE"
4094
*Main> hexStrToDec "C0ffEE"
12648430
*Main> hexStrToDec "0xC0ffEE"
12648430
*Main> hexStrToDec "0XC0ffEE"
12648430
*Main> hexStrToDec "00FFE"
4094
*Main> hexStrToDec "0x00FFE"
4094
*Main> hexStrToDec "Hello World"
*** Exception: Non-hexadecimal digits present
```

9. *(10 marks)* While you may decide never to use a functional programming language again, it is neverthless important to understand functional programming concepts and constructs, as they frequently appear in imperative languages. For instance, Python provides *list comprehensions* for succinct creation of lists, while the popular Underscore.js library provides many functions (e.g., `map`, `filter`, etc.) from the functional programming world.

   Java 8 similarly introduced a number of functional programming constructs, including *method references* – which allow methods to be treated as *first-class citizens* – and lambda expressions, allowing short, simple functions to be defined on the fly and passed around as data.

   In this part, we'll get our feet wet with lambda expressions in Java.

   - Look up a tutorial on lambda expressions and the arrow operator in Java
   - Look up a tutorial on the `Predicate` interface in Java
   - Add the following methods to `asn4/q9/CustomerManager.java`

     ```
     // Return a new list containing only the customers that match the given predicate
     public List<Customer> getCustomersBy(Predicate<Customer> predicate)

     // Return a new list containing only customers 65 or older
     public List<Customer> getSeniors()

     // Return a new list containing only customers under 18
     public List<Customer> getChildren()

     // Return a new list containing only customers in the given country
     public List<Customer> getCustomersFrom(String country)

     // Return a new list containing only customers having a last name starting with the given prefix
     public List<Customer> getCustomersByLastNamePrefix(String prefix)
     ```

   - The last four methods should compute their result by passing lambda expressions to `getCustomersBy`
   - Your methods must **not** print anything. They are returning lists
   - You may **not** modify the method signatures given above, nor may you modify the `Customer` class
   - Test your code by uncommenting lines in the provided `asn4/q9/CustomerTest.java`

3 Marking

The assignment will be marked using a combination of automated and manual checks. Code will be assessed according to criteria including but not limited to:

- Correctness
- Adherence to the given specifications
- Succinctness and style
- Readability, including
  - Appropriate use of whitespace
  - Reasonable comments to allow someone to follow your algorithms

Each file should have a header comment as follows:

```
--
-- Assignment : 4
-- Author     : JOE USER
-- Email      : username@uwo.ca
--
-- Brief description of the contents of the file
--
```

4 Submitting Your Assignment

When you are ready to submit your assignment, you **must** perform the following steps:

- Commit and push your code:

  ```
  git add .
  git commit -m "Final assignment 4 submission"
  git push
  ```

- Tag the last commit with the tag `asn4` and push the tag:

  ```
  git tag asn4
  git push --tags
  ```

  Important: the tag **must** be named `asn4` (not `Asn4`, `a4`, etc.) and **must** be pushed. **Your assignment is not submitted until this is done.**