M21274 – MATHFUN
Discrete Mathematics and Functional Programming

Worksheet 4: Tuples, Strings and Lists

Introduction

This worksheet introduces you to tuples, strings, and lists in Haskell. Begin by opening the `Week4.hs` file in your browser and saving it in your Haskell folder. This file defines a type synonym `StudentMark` as well as the functions from the lecture. It also includes the `testData` constant (a list of `StudentMarks`) that we will use to test functions that involve student marks.

Load this script into GHCi and experiment by evaluating some expressions. Then work your way through the examples 1 and 2. Finally, add your solutions to the programming exercises at the end of `Week4.hs`.

Worked example 1

Question: Write a function `sumEvenNumbersBetween` that calculates the sum of even integers between (inclusive) its two arguments:

```
sumEvenNumbersBetween :: Int -> Int -> Int
```

Test	Output
`sumEvenNumbersBetween 5 8`	`14`
`sumEvenNumbersBetween 7 5`	`0`
`sumEvenNumbersBetween 6 6`	`6`

Solution: Let's first define a recursive function `sumNumbersBetween` that returns the sum of all numbers between the two arguments. Add the following recursive definition to `Week4.hs`:

```
sumNumbersBetween :: Int -> Int -> Int
sumNumbersBetween x y
  | x > y = 0
  | otherwise = x + sumNumbersBetween (x + 1) y
```

Now let's rewrite the same function using a list comprehension. Comment out the recursive definition and add the highlighted line:

```
sumNumbersBetween :: Int -> Int -> Int
sumNumbersBetween x y = sum [i | i <- [x .. y]]
-- sumNumbersBetween x y
-- | x > y = 0
-- | otherwise = x + sumNumbersBetween (x + 1) y
```

Read "`[i | i <- [x .. y]]`" as "*list of numbers* `i` *where* `i` *is an element of the list of numbers from* `x` *to* `y`". (We note that this list can be more simply written as: "`[x .. y]`".) We also use the `sum` function to calculate the sum of the values in the list.

The recursive definition of `sumEvenNumbersBetween` below uses the `mod` function to check the remainder of the division of `x` by 2 ("`mod x 2 == 0`" passes when `x` is even):

```
sumEvenNumbersBetween :: Int -> Int -> Int
sumEvenNumbersBetween x y
  | x > y = 0
  | mod x 2 == 0 = x + sumEvenNumbersBetween (x + 2) y
  | otherwise = sumEvenNumbersBetween (x + 1) y
```

Remember that if you surround the `mod` function with backticks (`` ` ``), it can be used as an operator, and thus it can be placed between its operands: `x `mod` 2 == 0`.

Now let's write a version of `sumEvenNumbersBetween` that uses a list comprehension:

```
sumEvenNumbersBetween :: Int -> Int -> Int
sumEvenNumbersBetween x y = sum [i | i <- [x .. y], mod i 2 == 0]
-- sumEvenNumbersBetween x y
-- | x > y = 0
-- | mod x 2 == 0 = x + sumEvenNumbersBetween (x + 2) y
-- | otherwise = sumEvenNumbersBetween (x + 1) y
```

Read "`[i | i <- [x .. y], mod i 2 == 0]`" as "*all numbers i between x and y that pass the evenness check*". You could further simplify the definition of this list, using the `even` function, to "`[i | i <- [x .. y], even i]`".

Worked example 2

Question: Write an `averageMark` function which gives the average mark of the students in a `StudentMark` list (or 0 if the list is empty):

```
averageMark :: [StudentMark] -> Float
```

Test	Output
`averageMark [("Stefan", 56), ("Anya", 73)]`	64.5
`averageMark testData`	48.75
`averageMark []`	0

Solution: The definition of `StudentMark` (shown below) defines this type as a pair (a tuple with 2 elements) where the first element is the name of the student and the second is their mark:

```
type StudentMark = (String, Int)
```

We already have a function `marks` in `Week4.hs` that returns the marks from a list of `StudentMarks`:

```
marks :: [StudentMark] -> [Int]
marks stmks = [mk | (st, mk) <- stmks]
```

The code above uses **tuple unpacking**. It splits every element of `stmks` into a pair `(st, mk)`. We could replace `st` with a wildcard (`_`) as we do not really use the name of the student, so the code becomes `[mk | (_, mk) <- stmks]`.

An alternative approach to tuple unpacking is to get the second element of each pair using the `snd` function:

```
marks :: [StudentMark] -> [Int]
marks stmks = [snd stmk | stmk <- stmks]
```

Let's proceed with the first version here. As we saw in worked example 1, we can use the `sum` function to get the sum of the values in a list:

```
sumMarks :: [StudentMark] -> Int
sumMarks stmks = sum [mk | (_, mk) <- stmks]
```

To get the number of students, we can use the `length` function; for example:

```
numberOfStudents :: [StudentMark] -> Int
numberOfStudents stmks = length stmks
```

Putting everything together, we can now write a definition for `averageMark`:

```
averageMark :: [StudentMark] -> Float
averageMark [] = 0
averageMark stmks = fromIntegral sumMarks / fromIntegral numberOfStudents
  where
    sumMarks = sum [mk | (_ , mk) <- stmks]
    numberOfStudents = length stmks
```

(Recall that we need `fromIntegral` to convert values of type `Int` before dividing them using "/".) Add this definition to `Week4.hs` and test it using `testData` by checking whether the following evaluates to 48.75:

```
averageMark testData
```

Programming exercises

Tuples

1. Write a function `sumDifference` that returns the sum and difference of two numbers as a tuple.

   ```
   sumDifference :: Int -> Int -> (Int,Int)
   ```

   Hint: See the `minAndMax` function in `Week4.hs` which also returns a tuple.

Test	Output
`sumDifference 5 3`	`(8,2)`
`sumDifference 3 5`	`(8,-2)`

2. Students are given 'A' for 70% or higher, 'B' for 60+, 'C' for 50+, 'D' for 40+, and 'F' for everything else. Write a function that grades a given `StudentMark`:

   ```
   grade :: StudentMark -> Char
   ```

Test	Output
`grade ("Sam", 85)`	`'A'`
`grade ("Chris", 39)`	`'F'`

   Your function should return an error if the mark is not between 0 and 100, using Haskell's `error` function. For example, the following function takes a `Char` and uses the functions `isLower` and `isUpper` to check if a character is upper- or lowercase and returns an error otherwise:

   ```
   lowerOrUpperCase :: Char -> String
   lowerOrUpperCase c
     | isLower c = "lower case"
     | isUpper c = "upper case"
     | otherwise = error "Not a letter"
   ```

3. The mark for late assignments or exam retakes is capped at 40%. Write a function with the following signature that caps a student's mark (see the test cases):

```
capMark :: StudentMark -> StudentMark
```

Test	Output
`capMark ("Xiu", 89)`	`("Xiu", 40)`
`capMark ("Ahmed", 36)`	`("Ahmed", 36)`

(You should add the same error check as in the previous exercise.)

Lists and Strings

4. Write `firstNumbers` which returns a list of integers from 1 to the given argument.

```
firstNumbers :: Int -> [Int]
```

Test	Output
`firstNumbers 5`	`[1,2,3,4,5]`
`firstNumbers 0`	`[]`

5. Using `firstNumbers` write `firstSquares` that returns the list of first `n` squares given a positive integer `n`.

```
firstSquares :: Int -> [Int]
```

Test	Output
`firstSquares 5`	`[1,4,9,16,25]`

Haskell has many power operators, use `(^)` for this question. Remember that you can apply an operation on each element when using list comprehension. For example, the constant `testList` defined in the below evaluates to `[3, 6, 9, 12, 15]`:

```
testList = [x * 3 | x <- [1 .. 5]]
```

6. Using a list comprehension, write a function that capitalizes a given string.

```
capitalise :: String -> String
```

Test	Output
`capitalise "Po1 3he"`	`"PO1 3HE"`
`capitalise "Hey!"`	`"HEY!"`

Remember that strings are lists of characters. You can use list comprehension to create them in the same way as you did with numbers. You also need a "case conversion" function from the `Data.Char` module. You have already imported it with the following line in `Week4.hs`.

```
import Data.Char
```

7. Using a list comprehension, write a function `onlyDigits` that only keeps the numerical characters in a string.

Test	Output
`onlyDigits "ac245d62"`	`"24562"`
`onlyDigits "6:00 am"`	`"600"`

```
onlyDigits :: String -> String
```

Hint: Look for a "character classification" function in the `Data.Char` module.

8. Using your `capMark` function and a list comprehension, write a function that caps the mark of each student on a given list of `StudentMarks`.

```
capMarks :: [StudentMark] -> [StudentMark]
```

Test	Output
`capMarks [("Yahya",37), ("Phan",76)]`	`[("Yahya", 37), ("Phan", 40)]`

Test your function using `testData` by evaluating the following:

```
capMarks testData
```

9. Using your `grade` function and a list comprehension, write a function that grades every student on a given list of `StudentMarks`.

```
gradeStudents :: [StudentMark] -> [(String,Char)]
```

Test	Output
`gradeStudents [("Linh",47), ("Olu",76)]`	`[("Linh','D'), ("Olu','A')]`

Also, remember to test your function using `testData`:

```
gradeStudents testData
```

10. Write a function `duplicate` that repeats a string a given number of times.

```
duplicate:: String -> Int -> String
```

Test	Output
`duplicate "Hi" 3`	`"HiHiHi"`
`duplicate "1 2 3 4 " 2`	`"1 2 3 4 1 2 3 4 "`

Start by writing a recursive solution that uses guards. Then write a new solution that uses list comprehension similar to worked example 1. You must not use the `replicate` function as part of your solutions. However, you may wish to use the concatenation operator (++) or the `concat` function.

11. Using a list comprehension, write a function that lists all the divisors of a number.

```
divisors :: Int -> [Int]
```

Hint: See how we used the `mod` function in worked example 1.

Test	Output
`divisors 15`	`[1, 3, 5, 15]`
`divisors 1`	`[1]`
`divisors -20`	`[]`

12. Using your `divisors` function, write a function that checks whether a number is prime.

```
isPrime :: Int -> Bool
```

Hint: We have not provided any test cases for this question, so think about some prime numbers and list their divisors.

13. Using list comprehensions, write a polymorphic function (i.e., a function that works for any type `a` and `b`) that makes a list of pairs into a pair of lists.

```
split:: [(a,b)] -> ([a],[b])
```

Test	Output
`split [(1,'a'), (2,'b'), (3,'c')]`	`([1,2,3], "abc")`
`split [("2 > 3",False),("5 == 5",True)]`	`(["2 > 3","5 == 5"],[False,True])`
`split []`	`([],[])`

Write two solutions for this function. The first solution should use the `fst` and `snd` functions. The second function should use tuple unpacking as shown in the second worked example.