Tutorial 1: Introduction to Haskell Programming

The goal of this tutorial is to get the students familiar with Haskell programming. Students are encouraged to bring their own laptops to go through the installation process of Haskell and corresponding editors, especially if they haven't tried to install Haskell before or if they had problems with the installation. In any case the lab machines will have Haskell installed and students can also use these machines for the tutorial.

(Optional) Installing Haskell and related tools

If you have your laptop and have not installed Haskell yet, you can try to install it now. If you have problems we can try to help you.

The Haskell Platform is the easiest way to install Haskell in Windows or Mac OS:

http://www.haskell.org/platform/

In Ubuntu Linux you can use:

```
> sudo apt-get install haskell-platform
```

Basic steps in Haskell

In this tutorial we are going to write our first Haskell code.

Part 1: Create a file called "Tutorial1.hs"

Download the file `Tutorial1.hs` from Moodle and open it, in the first line, you will see the module header as follows:

```haskell
module Tutorial1 where
```

Part 2: First definitions in Haskell

In the Monday lecture we have seen some basic Haskell definitions. For example, we have seen how to define a function that computes the absolute value of a number:

```haskell
absolute :: Int -> Int
absolute x = if (x < 0) then -x else x
```

Everything is an Expression in Haskell: Generally speaking Haskell definitions have the following basic form:

$$name\ arg_1\ \ldots\ arg_n = expression$$

Note that in the right side of = (the body of the definition) what you have is an expression. This is different from a conventional imperative language, where the body of a definition is usually a statement. In fact in Haskell there are no statements, and in particular the `if` construct in `absolute` is also an expression.

Question 1: Are both of the following Haskell programs valid?

```
nested_if1 x = if ( (absolute x) <= 10)
               then x
               else error "Only numbers between [-10,10] allowed"

nested_if2 x = if ( (if x < 0 then -x else x) <= 10)
               then x
               else error "Only numbers between [-10,10] allowed"
```

Once you have thought about it you can try these definitions on your Haskell file and see if they are accepted or not.

Question 2: Can you have nested "if" statements in Java, C or C++? For example, would this be valid in Java?

```
int m(int x) {
  if ((if (x < 0) -x else x) > 10) return x; else return 0;
}
```

Part 3: Parametric Polymorphism and Type-Inference

We have seen that Haskell supports definitions with a type-signature or without. When a definition does not have a signature, Haskell infers one and it is still able to check whether some type-errors exist or not. For example, for the definition

```
newline s = s ++ "\n"
```

Haskell is able to infer the type: `String -> String`

(**Note**: In Haskell strings are represented as lists of characters (`[Char]`). The operator `++` is a built-in function in Haskell that allows concatenating two lists.)

However for certain definitions it appears as if there is not enough information to infer a type. For example, consider the definition:

```
identity x = x
```

This is the definition of the identity function: the function that given some argument returns that argument unmodified. This is a perfectly valid definition, but what type should it have?

The answer is:

```
identity :: a -> a
```

The function `id` is a **(parametrically) polymorphic** function. **Polymorphism** means that the definition works for multiple types; and this type of polymorphism is called **parametric** because it results from abstracting/parametrizing over a type. In the type signature the a is a **type variable** (or **parameter**). In other words it is a variable that can be replaced by some valid type (for example `Int`, `Char` or `String`). Indeed the `identity` function can be applied to any type of values. Try the following in `ghci`, and see what is the resulting type:

```
Tutorial1> identity 3
Tutorial1> identity 'c'
Tutorial1> identity identity  -- you may try instead :t identity identity
```

Question 3: Have you seen this form of polymorphism in other languages? Perhaps under a different name?

Part 4: Pattern matching

One feature that many functional languages support is **pattern matching**. Pattern matching plays a role similar to conditional expressions, allowing us to create definitions depending on whether the input matches a certain pattern. For example, the function `hd` (to be read as "head") given a list returns the first element of the list:

```
hd :: [a] -> a
hd []        = error "cannot take the head of an empty list!"
hd (x:xs)    = x
```

In this definition, the pattern `[]` denotes the empty list, whereas the pattern `(x:xs)` denotes a list where the first element (or the head) is `x` and the remainder of the list is `xs`. Note that instead of a single clause in the definition there are now two clauses for each case.

Question 4: Define a `tl` function that given a list, drops the first element and returns the remainder of the list. That is, the function should behave as follows for the sample inputs:

```
Tutorial1> tl [1,2,3]
[2,3]
Tutorial1> tl ['a', 'b']
['b']
```

More Pattern Matching: Pattern matching can be used with different types. For example, here are two definitions with pattern matching on tuples and integers:

```haskell
first :: (a,b) -> a
first (x,y) = x

isZero :: Int -> Bool
isZero 0 = True
isZero n = False
```

Part 5: Recursion

In functional languages mutable state is generally avoided and in the case of Haskell (which is purely functional) it is actually forbidden. So how can we write many of the programs we are used to? In particular how can we write programs that in a language like C would normally be written with some mutable state and some type of loop? For example:

```c
int sum_array(int a[], int num_elements)
{
   int i, sum=0;
   for (i=0; i<num_elements; i++)
   {
     sum = sum + a[i];
   }
   return(sum);
}
```

The answer is to use **recursive** functions. For example here is how to write a function that sums a list of integers:

```
sumList :: [Int] -> Int
sumList []     = 0
sumList (x:xs) = x + sumList xs
```

Question 5: The factorial function can be defined mathematically as follows:

$$n! = \begin{cases} 1 & \text{if } x = 0 \\ (n-1)! \times n & \text{if } x > 0 \end{cases}$$

Translate this definition into Haskell using recursion and pattern matching.

Question 6: The Fibonacci sequence is:

```
0, 1, 1, 2, 3, 5, 8, 13, ...
```

write a function:

```
fibonacci :: Int -> Int
```

Question 7: Write a function:

```
mapList :: (a -> b) -> [a] -> [b]
```

that applies the function of type `a -> b` to every element of a list. For example:

```
Tutorial1> mapList absolute [4,-5,9,-7]
[4,5,9,7]
```

Question 8: Write a function that given a list of characters returns a list with the corresponding ascii number of the character. Note that in Haskell, the function `ord`:

```
ord :: Char -> Int
```

gives you the ascii number of a character. To use it add the following just after the module declaration:

```
import Data.Char
```

Question 9: Write a function `filterList` that given a predicate and a list returns another list with only the elements that satisfy the predicate.

```
filterList :: (a -> Bool) -> [a] -> [a]
```

For example, the following filters all the even numbers in a list (`even` is a built-in Haskell function):

```
Tutorial1> filterList even [1,2,3,4,5]
[2,4]
```

Question 10: In the `Prelude` library, there is a function `zip`:

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

that given two lists pairs together the elements in the same positions. For example:

```
Tutorial1 > zip [1,2,3] ['a', 'b', 'c']
[(1, 'a'),(2, 'b'),(3, 'c')]
```

Write a definition `zipList` that implements the `zip` function.

Question 11: Define a function `zipSum`:

```
zipSum :: [Int] -> [Int] -> [Int]
```

that sums the elements of two lists at the same positions.

(Suggestion: You can define this function recursively, but a simpler solution can be found by combining some of the previous functions.)

Part 6: User-defined Datatypes

You may be wondering how we we can make our own types out of existing ones in Haskell. The short answer is: use **datatype** declaration. In fact, many built-in types in Haskell like Boolean and lists are actually defined using datatypes. For example, the built-in type `Bool` is defined in the standard library as follows:

```
data Bool = True | False
```

The keyword **data** introduces a new datatype. The part before `=` defines a new type, which in our case is `Bool`. The parts after `=` are data constructors. They tell us what values this type can have. The `|` symbol can be read as *or*. So in the above example, `Bool` type can only have two values, i.e., either `True` or `False`. Note that both the type name and the data constructors much begin with capital letters.

A data constructor can take some parameters and produce a new value. In the `Bool` example, this is not the case since `True` and `False` take no parameters. Let us define a *list* datatype to illustrate the use of type parameters:

```
data MyList a = Nil | Cons a (MyList a)
```

Here `a` is a type parameter, which can be used in the data constructors. Depending on what kind of values we want our own version of list type to hold, we can have different types of lists. Once the type parameter `a` is fixed to some concrete type, we can only have elements of that type in the list. For example, if we pass `Int` as the type parameter to `MyList`, we end up having a type `MyList Int`, and the two data constructors `Nil` and `Cons` have the following types correspondingly:

```
Nil  :: MyList Int
Cons :: Int -> MyList Int
```

Note that the above types tell us that we can only make a list of integers, not a list of characters or some other type. So

```
Cons 1 (Cons 2 (Cons 3 Nil))
```

creates a list of integers, whose elements are 1, 2 and 3.

A `Maybe` datatype is defined as

```
data Maybe a = Nothing | Just a
```

Quite look like our `MyList` example. You can think of this datatype as representing computations which might "go wrong" by not returning a value, or "succeeded" in producing some value

The `Maybe` datatype provides a way to make a safe version of a *partial* function. A partial function is simply a function that is undefined for some argument. For example, the built-in function `hd` is partial, because if we pass an empty list `[]` to it, it raises an exception, and makes our program crash. A safe version of `hd` won't do that:

```
safeHead :: [a] -> Maybe a
```

As can be seen from the type, it returns a value of `Maybe a`. So if we pass an empty list to `safeHead`, it return `Nothing`, otherwise, it wraps the head element of the list inside `Just`. Some examples below:

```
Tutorial1> safeHead []
Nothing

Tutorial1> safeHead [1,2,3]
Just 1
```

Question 12: Define the function `safeHead`.

Question 13: Define a `catMaybes` function, which takes a list of `Maybe`s and returns a list of all the `Just` values. For example:

```
Tutorial1> catMaybes [Just 1, Nothing, Just 2]
[1,2]
```

Assignment 1 (Optional and not graded)

Your first assignment is to try to complete as many questions in the tutorial as you can. You should submit your solutions to Moodle. Look for Submission of Tutorial 1 (optional) under section *Submissions and Score Reporting*.

The deadline for this assignment is **September 16, 11:55 PM**.

This assignment is not graded and it is also optional, but we will try to provide some feedback on your answers, and give some suggestions on how to improve the code. It is fine to submit only some answers. The goal of this assignment is simply for you to get some practice with Haskell programming and the corresponding tool-chain (ghci, editors, ...) as this will be important to get started with developing interpreters for programming languages.