VU University Amsterdam | Haskell exercises set 1
equational programming 2016-2017 | deadline 2017 01 13

Preparation and administration

- We use the Glasgow Haskell Compiler (GHC).
- If you work on your own laptop, download and install the Glasgow Haskell Compiler. The following website contains the relevant information:

  `https://www.haskell.org/platform/`

  See for this link and other links the homepage of the course.
- Collect the answers to the exercises in the section 'Exercises' in the skeleton that is available via the homepage.
- Keep the name `Practicum1.hs` for your skeleton-based file.
- Submit a zip containing your sketelon-based file via `http://cn.dashnet.tech`.

Goal

The goal of the first set of Haskell exercises is to get used to the basics of functional programming and elementary syntax of Haskell.

First steps

The first exercises are meant to be an introduction to Haskell. The exercises in this section do not need to behanded in.

1. Start the interactive GHC by typing

   ```
   ghci
   ```

   The effect should be something as follows:

   ```
   GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
   Prelude>
   ```

   The prompt `>` indicates that the system is waiting for an expression to be evaluated.
2. Use Haskell as a calculator, and check the preferences of the arithmetical operations, by calculating for example the following:

```
11 + 7
11 + (7 * 4)
11 + 7 * 4
(11 + 7) * 4
11 / 7
mod 11 7
11 'mod' 7
mod 7 11
7 'mod' 11
(-3) * (-7)
3.14 * 1.4
```

Note: we can use `mod` in infix by putting quotes around it. The parentheses around 'minus 3' are necessary; without them minus is interpreted as subtraction.

3. Use Haskell to do some boolean calculations, for example:

```
not (1<2)
not not True
not (not True)
(1<2) && (2<3)
True && Talse
False && x
True || False
True || x
```

Note: Haskell is case-sensitive.

4. Use Haskell to manipulate with characters and strings, for example:

```
‘a’
‘‘a’’
‘‘this is a string’’
```

5. We often work with lists, for example:

```
[1,2,3]
[True,True,False]
['a', 'b', 'c']
["dit" , "is" , "een", "lijst"]
[]
```

All elements in a list must be of the same type. Elements are separated by comma's.

6. We try some predefined operations on lists, for example:

   ```
   head [1,2,3]
   tail [1,2,3]
   take 2 [1,2,3]
   drop 2 [1,2,3]
   length [1,2,3]
   sum [1,2,3]
   [1,2,3] ++ [4,5]
   reverse [1,2,3]
   ```

   We will sometimes redefine operations that are already predefined. This is because one of the aims is to understand (not only to use) the definitions in Haskell.

7. Try a pair or product, for example:

   ```
   (1,True)
   ```

   Elements of a product may be of different types.

8. Haskell is strongly typed. We can use the command `':type` to see the type of an expression, for example:

   ```
   :type (length [1,2,3])
   ```

9. We can write (anonymous) functions using lambda-abstraction, for example:

   ```
   (\x -> \y -> x+y) 3 5
   ```

Using the interpreter

The following commands may be useful:

- `:load <name>`
  load script name
- `:reload`
  reload current script
- `:set editor vim`
  take vim as editor
- `:edit <name>`
  edit script name

- `:edit`

  edit current script
- `:type <expr>`

  show type of expr
- `:set <prop>`

  change various settings
- `:show <info>`

  show various information
- `:! <cmd>`

  execute cmd in shell
- `:?`

  show help text
- `:quit`

  quit

Any command can be abbreviated by its first character.

If you work with a file, as in the section 'Exercises', it is convenient to open the file in an editor for modifying your script, and use the Haskell compiler in a terminal. Once your file is correctly loaded, you can use in the interpreter the functions that are defined in the file.

An alternative to using the interpreter is using the compiler.

Exercises

Write the solutions to the following exercises in a file, starting from the skeleton available via the homepage of the course. Test every definition on at least two different inputs. Write the tests in the file, and add the answers to every test as a comment.

1. Define a function

   ```
   maxi :: Integer -> Integer -> Integer
   ```

   that takes two Integers as input and returns the largest (according to the usual ordering) one.

2. Define a function

   ```
   fourAscending :: Integer -> Integer -> Integer -> Integer -> Bool
   ```

that returns `True` exactly if the four arguments are strictly increasing according to the usual ordering on integers.

3. Define a function

   ```
   fourEqual :: Integer -> Integer -> Integer -> Integer -> Bool
   ```

   that returns `True` exactly if the four arguments are equal.

4. Define a function

   ```
   fourDifferent :: Integer -> Integer -> Integer -> Integer -> Bool
   ```

   that returns `True` exactly if all four arguments are distinct.

5. The following function is meant to return `True` exactly if all arguments are different.

   ```
   threeDifferent :: Integer -> Integer -> Integer -> Bool
   threeDifferent a b c = ( ( a /= b) && (b /= c) )
   ```

   Provide it with an input showing that the function is incorrect.

6. Define a function for factorial:

   ```
   factorial :: Integer -> Integer
   ```

7. Define a function

   ```
   fib :: Integer -> Integer
   ```

   such that `fib n` returns the nth Fibonacci number.

   The Fibonacci numbers F_i are defined by $F_0 = 0$, $F_1 = 1$, en $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$.

8. `strangeSummation :: Integer -> Integer`

   that for input n calculates the summation $\sum_{i=n}^{n+7} i$.

9. First some explanations about lists in Haskell.

   For every type `A` there is the of lists of items of type `A`, written as `A`.

   Use the command `:type` to see the type of the following:

   ```
   [1,2,3]
   [True,True]
   ['a','b']
   ["a","ab"]
   [[1,2],[3,4]]
   ```

(`Num` is a basic numeric class.)

Functions on lists are usually defined using recursion. Consider for example the function computing the length of a list of Integers:

```
lengthList :: [Integer] -> Integer
lengthList []  = 0
lengthList (h:t) = 1 + lengthList t
```

The intuition is: a list is either empty (first clause) or non-empty (second clause). There are no other possibilities. The exhaustive set of possibilities is described using pattern matching.

After loading the file containing the function definition, we can use the function as follows:

```
*Main> lengthList [1,2,3]
3
```

An alternative notation for the function above is:

```
lengthListAlternative :: [Integer] -> Integer
lengthListAlternative l =
  case l of
    [] -> 0
    (h:t) -> 1 + (lengthListAlternative t)
```

Note the indentation! This is part of the code (‘whitespace is code’).

Check both function on two different inputs each.

Now comes the real exercise.

Define a function

```
sumList :: [Integer] -> Integer
```

that takes as input a list of Integers and gives as output the sum of all items. The empty list is mapped to `0`.

10. Define a function

```
doubleList :: [Integer] -> [Integer]
```

that takes as input a list of Integers and gives as output the list in which every item is (elementswise) doubled. For example, input `[1,2,3]` yields output `[2,4,6]`.

11. Define yourself a function

```
myappend :: [a] -> [a] -> [a]
```

that takes as input two finite lists, and gives back as output the concatenation of those lists.

Note that this function is already present in Haskell with infix notation `++`. For example:

```
Prelude> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

Your own definition should (of course) not use `++`. We should for example have the following:

```
*Main> myappend [1,2,3] [4,5,6]
[1,2,3,4,5,6]
```

12. Define a function

    ```
    myreverse :: [a] -> [a]
    ```

    that takes as input a list, and gives as ouput a list with the same elements but in reverse order. Use your previously defined function `myappend`; do not use the predefined function `reverse`. We should for example have the following:

    ```
    *Main> myreverse [1,2,3]
    [3,2,1]
    ```

13. Define a function

    ```
    mymember :: a -> [a] -> Bool
    ```

    that indicates whether an element does or does not occur in a list. We should for example have the following:

    ```
    *Main> mymember 3 [1,2]
    False
    *Main> mymember 3 [1,2,3]
    True
    ```

14. Define a function

    ```
    mysquaresum :: [Integer] -> Integer
    ```

    that takes as input a list and calculates the sum of the squares of the elements. It returns 0 if the input is the empty list.

15. Define a function

    ```
    range :: Integer -> Integer -> [Integer]
    ```

    that takes as input two Integers, and gives back as output the following: the list of Integers starting at the first input and up to the second input. In case the first input is larger than the second, the output is the empty list.

    This function is already present in Haskell using notation `[m .. n]`. For example:

    ```
    Prelude> [3 ..7]
    [3,4,5,6,7]
    ```

    Your own definition should (of course) not use `[m .. n]`.

16. The function `concat` is defined in Haskell. It takes as input a list of lists, and gives as output the concatenation of the lists. An example of the use of this function:

    ```
    Prelude> concat [ [1,2,3] , [4,5,6] , [] ]
    [1,2,3,4,5,6]
    ```

    Give your own definition

    ```
    myconcat :: [[a]] -> [a]
    ```

    with the same behaviour (but of course not using) `concat`.

17. Define insertionsort. First define a function

    ```
    insert :: Ord a => a -> [a] -> [a]
    ```

    that given an Integer and a sorted list inserts the Integer at the right place (that is: such that the sorting is respected) in the list. Example:

    ```
    *Main> insert 7 [1,2,3,5,7,8,9,12]
    [1,2,3,5,7,7,8,9,12]
    ```

    Second, use `insert` to define insertionsort:

    ```
    insertionsort :: Ord a => [a] -> [a]
    ```

    The type of this function is intuitively: for every type `a` that is totally ordered, it takes as input a list of elements of type `a`, and returns a list of elements of type `a`.

18. First some remarks about the `filter`.

    The function `filter` in Haskell takes as input a predicate and a list, and gives back as output the list with only those elements from the input-list for which the predicate holds. (The order is kept intact.) Example:

    ```
    evensA xs = filter even xs
    ```

    Then:

    ```
    *Main> evensA [1,2,3,4,5,6]
    [2,4,6]
    ```

    Use filter on at three well-chosen tests.

    Note that using lambda-notation you can use anonymous functions in Haskell. Example:

    ```
    \x -> x > 5
    ```

    Such predicate can also be used:

    ```
    Prelude> filter (\x -> x >5) [1,5,8,3]
    [8]
    ```

    Now comes the real exercise.

    Use `filter` to define two auxiliary functions, and use those to define quicksort:

    ```
    quicksort :: Ord a => [a] -> [a]
    ```

    Note that the functional version of quicksort is not in-place.

    The function `filter` is an example of a higher-order function because one of its two arguments is not of base type, but of function type.

19. List comprehension is an operation that yields a list of elements that satisfy some predicate. The construction is for example as follows:

    ```
    [x | x<- ys, x<2]
    ```

    We can read this as 'the list consisting of the elements x such that x is in the list and x is smaller than y'.

    Use list comprehension to define

    ```
    evensB :: [Integer] -> [Integer]
    ```

with the same behaviour as `evensA` defined earlier.

20. First some remarks about `map`.

    The function `map` is defined in Haskell. The function takes as input a function and a list, and gives back as output the list where the function is applied elementswise to the list. An example of the use of `map`:

    ```
    Prelude> map (\x -> x+1) [1,2,3]
    [2,3,4]
    ```

    Here we use an namefree function. Note further that `map` is a higher-order function.

    The type of `map` is shown as follows:

    ```
    Prelude> :t map
    map :: (a -> b) -> [a] -> [b]
    ```

    Looking at the type of `map` we see that it expects a first input of type `a ->b`, so a function type, and then a second input of type `[a]`, so the type of lists over `a`. It gives back an output of type `[b]`, which is the type of lists over `b`.

    Test `map` on three well-chosen inputs.

    Now comes the real exercise.

    Define your own version of map

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

    with the same behaviour as `map`.

21. Give a definition of the function

    ```
    twice :: (a -> a) -> a -> a
    ```

    that takes as input a function and an argument, and that gives back as output the function applied twice to the argument. Example:

    ```
    *Main> twice (\x -> x+1) 3
    5
    ```

22. Give a definition of the function

    ```
    compose :: (b -> c) -> (a -> b) -> a -> c
    ```

    that takes as input two functions `f` and`g`, and that gives back as output the function `f` after `g`, where first `g` and next `f` is applied. Example:

```
*Main> compose (\x -> x * 5) (\x -> x +4)  7
55
*Main> compose (\x -> x + 4) (\x -> x * 5)  7
39
```

23. The functions `head` and `last` are defined in Haskell. If given a non-empty list as input, they return the first respectively last element of the input-list.

    Give your own definition

    ```
    mylast :: [a] -> a
    ```

    using `head`.

24. The function `take` takes as input a positive integer n and a list, and gives back as output the list consisting of the first n elements of the input-list (in order).

    The function `drop` takes as input a postitive integer b and a list, and gives back as output the list obtained by removing the first n elements of the input-list.

    Test these function, and give a new (not very natural) definition of

    ```
    mylastb :: [a] -> a
    ```

    with the same behaviour as `last`, using both `head` and `drop`.

25. Give two different definitions of

    ```
    myinit, myinitb :: [a] -> [a]
    ```

    which, with input a non-empty list, returns the input-list minus the last element. It is possible to use `tail` for one definition, and `take` for another one.

26. Use `foldr` to give two new definitions:

    ```
    mysecondconcat :: [[a]] -> [a]
    ```

    and

    ```
    mysecondreverse :: [a] -> [a]
    ```

    (For the latter you probably need an auxiliary function.)

27. Define a function

    ```
    prefix :: [a] -> [[a]]
    ```

that takes as input a list, and gives as output the list consisting of all prefixes of the input-list. An example of the use of `prefix`:

```
*Main> prefix [1,2,3]
[[],[1],[1,2],[1,2,3]]
```

First hint: the constructor `:` for lists is also considered a function, that takes as first input an element of type `a`, as second input an element of type `[a]`, and that gives back an output of type `[a]`. This can be seen from the following (note that the parentheses are necessary):

```
*Main> :t (:)
(:) :: a -> [a] -> [a]
```

Second hint: both in the λ-calculus and in functional programming we have the notion of *partial application*: it is possible to provide a function with only the first few, not necessarily all, arguments. Such a partially applied function can also be used in a larger context. See for example:

```
Prelude> let plus x y = x+y
Prelude> let plusdrie = plus 3
Prelude> :t plusdrie
plusdrie :: Integer -> Integer
Prelude> plusdrie 8
11
```