Prof. Dr. Peter Thiemann

Hannes Saffrich
saffrich@informatik.uni-freiburg.de

Winter Term 2024/25

Functional Programming

https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2024/

Exercise Sheet 1

Exercise 1 (Warming up)

1. Write two functions `maxi` and `mini`, which take two `Integer`s as arguments and return their maximum and minimum, respectively. Provide type signatures for each function. (Don't use the predefined `min` and `max` functions, obviously!)
2. Write two functions `max3` and `max3Tupled`, which take three `Integer`s as arguments and return their maximum. `max3` should take three arguments (curried function), whereas `max3Tupled` should take a single 3-tuple argument (uncurried function).
3. Define a function `med`, which takes three `Integer`s, and returns their median, i.e. the second largest argument.
4. Test your definitions with `QuickCheck` properties. For example, you can let `QuickCheck` check that your functions behave the same as packing the integers into a list, sorting the list, and then taking the first or last element of the list. The `sort` function is defined in the `Data.List` module and its documentation can be found at this URL:[1]
   https://hackage.haskell.org/package/base-4.16.3.0/docs/Data-List.html#v:sort

   The `Data.List` module is part of the `base` package, which is Haskell's standard library. This package is automatically listed as a dependency in your `.cabal` file. You can import the definitions of the `Data.List` module by writing `import Data.List` at the top of your file (after the line starting with `module`).

Exercise 2 (List functions)

In this exercise we're going to reimplement some of the standard library functions from the `Data.List` module.

This way you can practice writing recursive functions with pattern matching and getting comfortable with Haskell's lists, and at the same time explore the standard library.

For this purpose we use the same function names as in the standard library, but append a single quote to avoid name clashes, e.g. `null'` instead of `null`.

Try to write type signatures for the functions and make them reasonably polymorphic, e.g. a function which reverses a list does not only work for integer lists but lists with elements of arbitrary types.

1. Write a function `null'`, which takes a list as an argument and returns whether the list is empty.

   Examples:

   ```
   >>> null' []
   True
   >>> null' [1, 2]
   False
   ```

[1] On macOS, your PDF viewer might open the wrong URL. If so, replace the `%23` near the end with a `#` in the opened URL, or try to copy the text from the PDF.

2. Write a function `sum'`, which takes a list of integers and returns their sum.

   Examples:

   ```
   >>> sum' [1, 2, 3]
   6
   ```

3. Write a function `concat'`, which takes a list of lists and joins them together

   Examples:

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

4. Write a function `elem'`, which takes a value and a list and checks whether the value is in the list.

   Since we didn't cover type classes yet, you can assume the value and list elements are integers.

   Examples:

   ```
   >>> elem' 3 [1, 2, 3, 4]
   True
   >>> elem' 9 [1, 2, 3, 4]
   False
   ```

5. Write a function `take'`, which takes an integer `n` and a list `xs` as arguments and returns a list with the first `n` elements of `xs`.

   Examples:

   ```
   >>> take' 3 [1, 2, 3, 4, 5]
   [1, 2, 3]
   ```

6. Write a function `drop'`, which takes an integer `n` and a list `xs` as arguments and returns a list with everything but the first `n` elements of `xs`.

   Examples:

   ```
   >>> drop' 3 [1, 2, 3, 4, 5]
   [4, 5]
   ```

7. Write two functions `last'` and `lastSafe`, which take a list as an argument and return the last element of the list.

   If the list is empty, then `last'` should throw an `error`, whereas `lastSafe` should return `Nothing`.

   Examples:

   ```
   >>> last' [1, 2, 3]
   3
   >>> last' []
   *** Exception: last' is undefined on the empty list
   CallStack (from HasCallStack):
     error, called at /home/m0rphism/test.hs:1:12 in main:Main
   >>> lastSafe [1, 2, 3]
   Just 3
   >>> lastSafe []
   Nothing
   ```

8. Write two functions `init'` and `initSafe`, which take a list as an argument and returns the list without its last argument.

   If the list is empty, then `init'` should throw an `error`, whereas `initSafe` should return `Nothing`.

   Hint: `initSafe` is more difficult. You might need to use a `case` expression to pattern match on the result of the recursive call. You can leave this part of the exercise to the end if you get stuck.

   Examples:

   ```
   >>> init' [1, 2, 3]
   [1, 2]
   >>> init' []
   *** Exception: init' is undefined on the empty list
   CallStack (from HasCallStack):
     error, called at /home/m0rphism/test.hs:1:12 in main:Main
   >>> initSafe [1, 2, 3]
   Just [1, 2]
   >>> initSafe []
   Nothing
   ```

9. Write a function `zip'`, which takes two lists as arguments and returns a list of tuples of the elements of the argument lists.

   If one list is longer than the other, its extra elements are ignored.

   Examples:

   ```
   >>> zip [1, 2, 3] [True, False, True]
   [(1, True), (2, False), (3, True)]
   >>> zip [1, 2, 3, 4] [10, 20]
   [(1, 10), (2, 20)]
   ```

10. Write a function `reverse'`, which takes a list as an argument and returns the reversed list.

    Examples:

    ```
    >>> reverse [1, 2, 3]
    [3, 2, 1]
    ```

11. Write a function `intersperse'`, which takes a value and a list as arguments and places the value between each two elements of the list.

    This function is particularly useful with `String`s, which in Haskell are literally lists of characters.

    Examples:

    ```
    >>> intersperse' 666 [1, 2, 3, 4]
    [1, 666, 2, 666, 3, 666, 4]
    >>> intersperse' ',' "abcde"
    "a,b,c,d"
    ```

12. Check out the documentation of the `Data.List` module for many more useful functions: https://hackage.haskell.org/package/base-4.20.0.1/docs/Data-List.html

    Some of the functions, that we reimplemented in this exercise, have a more general type in `Data.List`. They don't have to make sense to you now and we will come back to them once we covered type classes in detail.