Prof. Dr. Peter Thiemann
Janek Spaderna
janek.spaderna@pluto.uni-freiburg.de

Winter Term 2022/23
2022-10-21

**Functional Programming**
http://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2022/

# Exercise Sheet 1

## Exercise 1 (Warming up)

1. Write two functions `maxi` and `mini` which compute the maximum and the minimum of two `Integer`s. Provide type signatures for each function. (Don't use the predefined `min` and `max`, obviously!)
2. Define two functions `max3` and `max3Tupled` which compute the maximum of three `Integer`s. `max3` should take three arguments, whereas `max3Tupled` should take a single 3-tuple argument.
3. Define a function `med`, which computes the median of three `Integer`s.
4. Test your definitions with QuickCheck properties. Use `Data.List.sort` as as reference:[1]
   https://hackage.haskell.org/package/base-4.16.3.0/docs/Data-List.html#v:sort

## Exercise 2 (Stack calculator)

We will now implement the core of a small stack-based calculator. A stack computer is capable of using `Integer`s with the following operations: `push` $n$, `pop`, `dup`, `add`, `subtract`, `multiply` and `neg`.

We represent the stack as a list of `Integer`s: `[Integer]`. The initial stack is infinitely deep and filled with zeros. This means the following sequence of operations succeeds and results in 8 on top of the stack:

```
pop
push 8
add
```

1. Implement the stack operations as functions that take a stack as their argument and return the updated stack.
2. Formulate properties about the stack operations and test your functions with QuickCheck.
3. In order to make our stack calculator convenient to use, we want users to be able to provide textual commands. Implement a function

   ```
   readCommand :: String -> [Integer] -> [Integer]
   ```

   which decodes the provided string and calls the appropriate function. An unrecognized operation should leave the stack unchanged (noop). The exact format of textual commands is up to you.

**Tips:**

- Strings are lists of `Char`, character literals (e. g. `'x'`) and string literals (e. g. `"ABC"`) can be used as patterns.
- The function `read` can be used to decode a string into a number:
  https://hackage.haskell.org/package/base-4.17.0.0/docs/Prelude.html#v:read

[1] On macOS these kinds of links might be broken, depending on your PDF viewer. If so, replace the `%23` near the end with a `#` in the opened URL, or try to copy the text from the PDF.

**Exercise 3** (List functions)

Implement the following functions (roughly in increasing complexity):

- `head`, `tail`, `last`
- `length`, `and`, `init`
- `(++)`, `zip`, `reverse`

**Tips:**

- All of these functions are defined in `Data.List`. You can find their documentation here: https://hackage.haskell.org/package/base-4.16.3.0/docs/Data-List.html
- Don't be put of by the type signature of, for example, `length`. You don't have to understand what a `Foldable` is.
- To avoid name clashes you can add a prime to at the end of the names of your functions. `(++')` is not a valid operator, though. Here you could use `(+++)`, for example.

**Useful links**

- Documentation of the `base` package: https://hackage.haskell.org/package/base-4.16.3.0
- Haskell-specific search engine: https://hoogle.haskell.org

**Getting QuickCheck to work**

If you want to use *stack* you can install it via GHCup with the command `ghcup install stack`.

You start a project by running either `stack new NAME` or `cabal init`. The former will create a new directory `NAME` the latter will put the files into your current directory.

Add QuickCheck as a dependency by editing either `package.yaml` (stack) or the `.cabal` file (don't touch the latter when using *stack*):

`package.yaml`

```
...
dependencies:
- base
- QuickCheck
...
```

`.cabal` file

```
...
build-depends:
  base,
  QuickCheck
...
```

You can also use the code from the lecture as a starting point: https://github.com/proglang/FunctionalProgramming/tree/master/code2022