Till Mossakowski
Benjamin Junge

1. Excercise sheet

Issued: 2023-10-12
Due: 2023-10-16 & 2023-10-17

Warmup

The following is a warmup for setting up a working Haskell environment on your system and getting used to working with the ghci.

1. Install a Haskell version from https://www.haskell.org/downloads/. After the installation you should be able to use the commands `ghc --version` and `ghci` in an command line interpreter without getting any errors.
2. Create a directory for exercise relevant files.
3. With an editor of your choice, create the file `Task00.hs` in this directory.
4. Copy and paste the code from the lecture slides into the file:

   ```
   fac :: Int → Int
   fac n = if n ≤ 0 then 1 else n * fac (n - 1)
   ```

   use '->' for → and '<=' for ≤
5. Start a command shell (`cmd.exe` on Windows, `bash` on Linux based systems) and navigate to you exercise directory.
6. Start the Haskell interpreter `ghci`.
7. Load the file `Task00.hs` into the interpreter: `:load Task00` or `:l Task00`.
8. The interpreter allows for interactive evaluation of functions defined in the loaded file and functions of the prelude. (for more information on prelude types and functions see https://hackage.haskell.org/package/base-4.14.0.0/docs/Prelude.html or use `:browse Prelude` in ghci) Evaluate the following commands with the ghci:

`17 + 4`	`100 / 7`	`sqrt 100 - 1`
`(-1) * (-1)`	`fac 10`	`fac (-3)`
`fac "kolibri"`	`fac 9 - 20`	`foo 10`

   Note that some of the expressions may throw an error with helpful information (that may look cryptic for now).
9. To list all possible commands used by the ghci type either `:help` or just `:h`. Some of the listed commands and options aren't relevant yet. The commands `:reload`(or `:r`), `:type`(`:t`) and `:browse` are quite helpful.

   The development cycle usually looks like the following: edit and save the file within an editor, loading/reloading the file in ghci (with `:l` or `:r`), evaluating error messages or testing the functions, returning to the editor.
10. After getting familiar with the `ghci` you can also try solving the tasks with IHaskell (https://github.com/gibiansky/IHaskell), which is a Haskell kernel for Jupyter (https://jupyter.org/) Notebooks. In case of the installation being to cumbersome you can also find an online version at https://mybinder.org/v2/gh/gibiansky/IHaskell/master.

1.1 *Sums*

1. Implement a recursive function `sum` that sums all integers from 1 to n:

$$sum(n) = \sum_{i=1}^{n} i$$

   What does the signature of `sum` look like?

2. Save your definition in a *.hs file and load it via ghci.

3. Think about test cases for your implementation and test them via ghci.

1.2 *Digit sums*

The *digit sum* of an integer is the sum of its digits.

It can be defined recursively as follows:

$$digit(x) \stackrel{def}{=} \begin{cases} 0 & \text{x= 0} \\ x \bmod 10 + digit(x \div 10) & \text{else} \end{cases} \tag{1}$$

with $x \div y$ being the integer division of x and y.

Implement the Haskell function

```
digit :: Integer → Integer
```

that calculates the digit sum:

```
digit 5 ⇝ 5
digit 457 ⇝ 16
```

The digit sum is usually only defined for positive integers. Think of a way to modify your definition such that $digit(-x) = -digit(x)$ for all positive integers x.

1.3 *Prefixes*

A string s is called *prefix* of string t if there exists a string e such that $t = s \circ e$, $\circ$ being string concatenation. One could call t continuation of s.

This can be defined recursively: s is a prefix of t, if

- s is empty, or
- the heads of s and t are equal and the tail of s is a prefix of the tail of t.

Define a function

```
isPrefixOf :: String→ String→ Bool
```

which tests for this prefix relation. e.g.:

```
isPrefixOf "" "Foo" ⇝ True
isPrefixOf "baz" "bazbar" ⇝ True
isPefixOf  "bar" "bazbar" ⇝ False
```

1.4 *Guards vs. If*

The lecture introduced the syntactic sugar of *guards* for writing (potentially nested) if statements.

a) alternative implementation
Write an alternative implementation for your solutions of 1.2 and 1.3. Use guards if your original solution used if statements, or vice versa. If your original solution didn't use guards or if statements, implement a solution with guards.

b) Otherwise
The lecture also introduced `otherwise` as a 'catch-all' condition to prevent runtime errors from no case matching. Think of at least two alternative constructs that act just like the otherwise condition, allowing to catch all cases.

1.5 *Laziness*

The lecture introduced the fact that Haskell is a *non-strict* language while most other languages are *strict*. Define *strict* and *non-strict* function, *strict* and *non-strict* language. Explain the difference between *non-strictness* and *laziness*. Name at least one *non-strict* construct from an otherwise *strict* language. Should every programming language have at least one *non-strict* construct or are they unnecessary?

1.6 * *Completing prefixes*

Write a function `asPrefixOf :: String -> String -> String` that prepends a prefix string p to a given string s in a way that the following test cases hold:

```
asPrefixOf "Foo" "" ⇝ "Foo"
asPrefixOf "foo" "barfoo" ⇝ "foobarfoo"
asPrefixOf "foo" "foobar" ⇝ "foobar"
asPrefixOf "bar" "bazbar" ⇝ "barzbar"
asPrefixOf "foobar" "obrbaz" ⇝ "foobarbaz"
```

Note that if parts of p are already contained in s they are not duplicated. To illustrate this property the last test case got colour-coded to indicate the origin of each letter in the resulting string: asPrefixOf foobar obrbaz ⇝ f o o b a rbaz