Prof. Dr. Ralf Hinze
M.Sc. Sebastian Schweizer

TU Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

Functional Programming: Exercise 2

Sheet published: Tuesday, April 23rd
Submission deadline: Tuesday, April 30th, 12:00 noon

Submission Instructions Please do all submissions in the `ex2` folder. Some tasks come with a skeleton file, edit those files directly. Tasks marked as *pen and paper* should be solved without GHCi, please submit your solutions as photo, scan, or plain text file.

Exercise 2.1 (Pen and paper)**.**

a) Recall the implementation of Insertion Sort from §0.5 (listed below, with some minor modifications).

$$
\begin{aligned}
&\mathit{insertionSort} :: [\mathit{Integer}] \to [\mathit{Integer}] \\
&\mathit{insertionSort}\ [\,] \qquad\quad = [\,] \\
&\mathit{insertionSort}\ (x : xs) = \mathit{insert}\ x\ (\mathit{insertionSort}\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{insert} :: \mathit{Integer} \to [\mathit{Integer}] \to [\mathit{Integer}] \\
&\mathit{insert}\ a\ [\,] = a : [\,] \\
&\mathit{insert}\ a\ (b : xs) \\
&\quad \mid a \leqslant b = a : b : xs \\
&\quad \mid a > b = b : \mathit{insert}\ a\ xs
\end{aligned}
$$

The function *insert* takes an element and an ordered list and inserts the element at the appropriate position, e.g.

$$
\begin{aligned}
&\quad \mathit{insert}\ 7\ (2 : (9 : [\,])) \\
\Longrightarrow &\quad \{\ \text{definition of } \mathit{insert} \text{ and } 7 > 2\ \} \\
&\quad 2 : (\mathit{insert}\ 7\ (9 : [\,])) \\
\Longrightarrow &\quad \{\ \text{definition of } \mathit{insert} \text{ and } 7 \leqslant 9\ \} \\
&\quad 2 : (7 : (9 : [\,]))
\end{aligned}
$$

Recall that Haskell has a very simple computational model: an expression is evaluated by repeatedly replacing equals by equals. Evaluate (by hand, using the format above) the expression $\mathit{insertionSort}\ (7 : (9 : (2 : [\,])))$.

b) The function *twice* applies its first argument twice to its second argument.

$$\mathit{twice}\ f\ x = f\ (f\ x)$$

(Like *map* and *filter*, it is an example of a higher-order function as it takes a function as an argument.) Evaluate $\mathit{twice}\ (+1)\ 0$ and $\mathit{twice}\ \mathit{twice}\ (*2)\ 1$ by hand.

Use the computer to evaluate

$\ggg\!\!\rangle$ *twice* `("|"++) ""`
$\ggg\!\!\rangle$ *twice twice* `("|"++) ""`
$\ggg\!\!\rangle$ *twice twice twice* `("|"++) ""`
$\ggg\!\!\rangle$ *twice twice twice twice* `("|"++) ""`
$\ggg\!\!\rangle$ *twice* (*twice twice*) `("|"++) ""`
$\ggg\!\!\rangle$ *twice twice* (*twice twice*) `("|"++) ""`
$\ggg\!\!\rangle$ *twice* (*twice* (*twice twice*)) `("|"++) ""`

Is there any rhyme or rhythm? Can you identify any pattern?

Exercise 2.2 (Sekeleton: `Poem.hs`)**.** Define the string *thisOldMan* :: *String* that produces the following poem (if you type *putStr thisOldMan*).

This old man, he played one,
He played knick-knack on my thumb;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played two,
He played knick-knack on my shoe;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played three,
He played knick-knack on my knee;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played four,
He played knick-knack on my door;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played five,
He played knick-knack on my hive;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played six,
He played knick-knack on my sticks;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played seven,
He played knick-knack up in heaven;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played eight,
He played knick-knack on my gate;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played nine,
He played knick-knack on my spine;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played ten,
He played knick-knack once again;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

Try to make the program as short as possible by capturing recurring patterns. Define a suitable function for each of those patterns.

Exercise 2.3 (Submit as: `Booleans.lhs`).

a) How many *total* functions are there that take one Boolean as an input and return one Boolean? Or put differently, how many functions are there of type $Bool \to Bool$? Define all of them. Think of sensible names.

   Note: If you choose sensible names, then chances are high that your functions collide with Haskell's predefined functions in *Prelude*. Name your functions foo' instead of foo to avoid that problem.

b) How many total functions are there that take two Booleans as an input and return one Boolean? Or put differently, how many functions are there of type $(Bool, Bool) \to Bool$? Define at least four. Try to vary the definitional style by using different features of Haskell, e.g. predefined operators such as $||$ and $\&\&$, conditional expressions (**if** .. **then** .. **else** ..), guards, and pattern matching.

c) What about functions of type $Bool \to Bool \to Bool$?

Exercise 2.4 (Skeleton: `Char.hs`). Haskell's *String*s are really lists of characters i.e. **type** $String = [Char]$. Thus, quite conveniently, all of the list operations are applicable to strings, as well: for example, *map toLower* `"Ralf"` $\Longrightarrow$ `"ralf"`. (Recall that *map* takes a function and a list and applies the function to each element of the list.)

a) Define an equality test for strings that, unlike $==$, disregards case, e.g. `"Ralf"` $==$ `"raLF"` $\Longrightarrow$ *False* but *equal* `"Ralf"` `"raLF"` $\Longrightarrow$ *True*.

b) Define predicates

   $$isNumeral :: String \to Bool$$
   $$isBlank \quad :: String \to Bool$$

   that test whether a string consists solely of digits or white space. You may find the predefined function $and :: [Bool] \to Bool$ useful which conjoins a list of Booleans e.g. $and\ [1 > 2, 2 < 3] \Longrightarrow False$ and $and\ [1 < 2, 2 < 3] \Longrightarrow True$. For your convenience, the skeleton file already imports *Data.Char*[1].

[1] This provides you some useful functions, see `http://hackage.haskell.org/package/base/docs/Data-Char.html`

c) Implement the Caesar cipher $shift :: Int \to Char \to Char$, e.g. $shift\ 3$ maps `'A'` to `'D'`, `'B'` to `'E'`, ..., `'Y'` to `'B'`, and `'Z'` to `'C'`. Try to decode the following message (*map* is your friend).

```
msg = "MHILY LZA ZBHL XBPZXBL MVYABUHL HWWPBZ JSHBKPBZ "
    ++ "JHLJBZ KPJABT HYJHUBT LZA ULBAYVU"
```

Hint 1. Functional programming folklore has it that a functional program is correct once it has passed the type-checker. Sadly, this is not quite true. Anyway, the general message is to exploit the compiler for *static* debugging: compile often, compile soon. (To trigger a re-compilation after an edit, simply type `:reload` or `:r` in GHCi.)

We can also instruct the compiler to perform additional sanity checks by passing the option `-Wall` to GHCi e.g. call `ghci -Wall` (turn all warnings on). The compiler then checks, for example, whether the variables introduced on the left-hand side of an equation are actually used on the right-hand side. Thus, the definition $f\ x\ y = x$ will provoke the warning "Defined but not used: y". Variables with a leading underscore are not reported, so changing the definition to $f\ x\ \_y = x$ suppresses the warning.