Functional programming in Haskell
First exercise sheet

TU Braunschweig

Sebastian Muskalla

Summer term 2018

Out: May 30

The exercises will be discussed during the exercise class on Thursday, May 31, 9:45 in IZ 305.

The code snippets from Exercise 1 are available as .hs file on the lecture page.

Exercise 1: Types

a) For each of the following expressions, determine the most general type.

```haskell
[True]
[]
\x -> x
\x y -> (y, x)
\x y -> [y, x]
\x -> (if x == [] then x else x)
head
\xs -> tail xs
f x y z = if x then y else z
\xs -> tail (tail xs)
g x y z = if x == y then y else z
```

b) For each of the functions h to `m`, determine the most general type.

```haskell
data Tree a = Leaf | Node [a] (Tree a) (Tree a)
h Leaf = "c"
h (Node "x" l r) = "x"
h (Node x l r) = x
i (Node x l r) = id
j (Node [] l r) = l
k Nothing = Nothing
l (Node a b c) = a
m (Node x l r) = Node [length x] (m l) (m r)
```

Exercise 2: Permute and sort

a) Write a function `permutations :: [Integer] -> [[Integer]]` that, given a list, computes a list containing all its permutations.

   What is the most general type of your function?

b) Write a function `idiotic_sort :: [Integer] -> [Integer]` that takes a list and sorts it by computing all permutations and returning the first one that is sorted.

   Note: `idiotic_sort` is a non-randomized version of Bogosort.

Exercise 3: Higher order functions

Write a function ...

a) `flatmap :: (a -> [b]) -> [a] -> [b]` that maps a function of type `a -> [b]` over a list and flattens the result.

b) `partition' :: (a -> Bool) -> [a] -> ([a],[a])` that partitions a list into the elements that satisfy a given predicate and the ones that do not satisfy the predicate.

c) `length' :: [a] -> Int` that computes the length of a list using `foldl`.

d) `minimum_wrt :: (a -> Integer) -> [a] -> a` that computes the minimum of a given list with respect to a given evaluation function. What is the most general type of your function?

e) `sort_wrt :: (a -> a -> Bool) -> [a] -> [a]` that sorts a list with respect to a given comparison operator. Which properties should the operator satisfy such that its usage makes sense?

Exercise 4: Davis Putnam

We want to implement the Davis Putnam algorithm for checking satisfiability in propositional logic.

a) A **formula** (in conjunctive normal form) is a set of clauses. A **clause** is a set of literals. A **literal** is either either a positive or a negative occurrence of a variable. The **variables** are taken from a countable set.

   The **negation** $\neg L$ of a literal L is the literal for the same variable with the opposite polarity.

   Design appropriate data types in Haskell. (How are the special formulas *true* and *false* represented?)

b) If φ is some formula and L is a literal, then the formula $\varphi[L]$ is obtained as follows:

   - Replace all occurrences of L by *true*
   - Replace all occurrences of $\neg L$ by *false*
   - Remove all occurrences of *false* from a clause
   - Remove all clauses containing an occurrence of *true*

   (What happens if a clause / the formula becomes empty?)

   Design a function `assign :: Formula -> Literal -> Formula` that implements this.

c) The Davis Putnam algorithm applies the following rules to a formula until one either finds a satisfying assignment or has proven the formula to be unsatisfiable.

   - **Unit**: If a formula φ contains a clause consisting of a single literal L, φ is satisfiable if and only if $\varphi[L]$ is satisfiable.

     Design a function `unit :: Formula -> Maybe Literal` that returns a unit literal if one exists.

   - **Pure**: If a formula φ contain only one type of literal L for some variable (i.e. the variable only occurs with one polarity), then φ is satisfiable if and only if $\varphi[L]$ is satisfiable.

     Design a function `pure :: Formula -> Maybe Literal` that returns a pure literal if one exists.

   - **Split**: A formula φ is satisfiable if $\varphi[L]$ or $\varphi[\neg L]$ is satisfiable for some literal L.

d) Design a function `sat :: Formula -> Bool` that checks whether a formula is satisfiable by using the Davis Putnam rules recursively.

   Advanced version: Write a function `sat :: Formula -> Maybe [Literal]` that checks whether a formula is satisfiable and if it is, also returns a satisfying assignment.