Worksheet 5

Algebraic data types

5.1 Consider the following custom data type for lists:

```
data List a = Empty | Cons a (List a)
```

Define two recursive functions `toList :: [a] -> List a` and `fromList :: List a -> [a]` to convert to and from the above data type and ordinary Prelude lists.

These functions are inverse of one another. In particular, we should have `fromList (toList xs) == xs` for all Prelude lists `xs`. This is enough to show that the above type is isomorphic to the ordinary lists.

5.2 Define a `Card` data type for representing playing cards of a standard deck: four suits (clubs ♣, spades ♠, hearts ♡, diamonds ♢) and 13 faces (2, 3, ..., 10, J, Q, K, A). You should define auxiliary types for suits and faces.

Define a list `allCards :: [Card]` with all cards of the playing deck; don't enumerate them manually—write a compact representation using a list comprehension.

5.3 There is a special algebraic data type for results of comparisons:

```
data Ordering = LT | EQ | GT  -- defined in the Prelude
```

The `compare` function compares two values and returns one of these values: `LT` means *less-than*, `EQ` means *equal*[1] and `GT` means *greater-than*.[2]

For example: `compare 3 1 == GT` (because 3 is greater than 1).

(a) Write a function `cmp1 :: Card -> Card -> Ordering` to compare playing cards of the previous exercise so that *all suits are ordered together* i.e. first clubs, then spades, then hearts and finally diamonds. Withing a suit, cards should be ordered by face.

(b) Write another comparison function `cmp1 :: Card -> Card -> Ordering` so that *all faces are ordered together*, i.e. all 2s, then all 3s, etc. then Js, Qs, Ks, As. Within a face, cards should be ordered by suit.

You can test your comparison function by using the `sortBy` function from `Data.List` with the complete deck: `sortBy cmp1 allCards` and `sortBy cmp2 allCards`.

[1] Don't confuse `EQ` (a constructor for `Ordering`) with the type class `Eq`.

[2] Note that `compare` allows distinguishing the three outcomes with a single comparision. Using (say) `<=` can only distinguish two outcomes, requiring a second comparison to disambiguate the third outcome.

Search trees and syntax trees

5.4 Consider the implementation of sets using binary trees presented in Lecture 10.

(a) Define a recursive function `size :: Set a -> Int` to compute the size of a set, i.e. its number of elements.

(b) Defined a recursive function `height :: Set a -> Int` to compute the height of the binary tree representing the set.

(c) Investigate the height of the following two sets of integers:

```
set1 = foldr insert empty [1..1000]
set2 = fromList [1..1000]
```

What can conclude about the search efficiency in each of these?

5.5 Consider the spelling checker program of Exercise 4.7 in the Worksheet 4. Change the data structure used for the dictionary from a list of words to a set of words using the binary tree implementation of Lecture 10.

You can leave `Set` as separate module and import its defintions in the spelling checker program. You'll need to use the `fromList` function to build the set from the list of words and use the set `member` function instead of list `elem`.

Once you get the change to compile, try it with a large text file (e.g. from Project Gutenberg). You should notice a substantial improvement in its running time because of the more efficient dictionary search.

5.6 Consider the type `Prop` for logic propositions defined the Lecture 10. Add a new `Or` constructor for disjuntions and extend the definiton of `eval` for this case.

5.7 Consider the type `Prop` for logic propositions defined the Lecture 10. Define a recursive function `vars :: Prop -> [Name]` that collects all names of variables occuring in the proposition (where `Name` is a type synonym for `Char`).

Example: `vars (And (Var 'p') (Not (Var 'p'))) == ['p','p']`. Note that we do not remove duplicates from the result (we could use `nub` to do so).

5.8 Define a recursive function `booleans :: Int -> [[Bool]]` that lists all possible sequences of n booleans for some non-negative n. Note that the length of `booleans n` is always 2^n.

Example: `booleans 2 = [[False,False], [False,True], [True,False], [True,True]]`.

5.9 Recall that an environment `Env` is a list of pairs `(Name,Bool)`. Define a function `environments :: [Name] -> [Env]` that generates all possible variable name assignments of some given variables to boolean values.

For example: `environments ['a','b']` should give a list with 4 environments with all possible assignments to variables a and b, i.e.

```
environments ['a','b'] == map (zip ['a','b']) (booleans 2)
```

where `booleans` is the solution to the previous exercise; this should give you an idea of how to generalize the example.

5.10 The *truth table* of a proposition is a list of type `[(Env,Bool)]` where each entry represents a line in the truth table, namely, the values of variables and the value of the proposition.

For example, the truth table for the proposition `And (Var 'x') (Var 'y'))` has the following four lines:

```
[ ([('x',False),('y',False)], False)
, ([('x',False),('y',True)], False)
, ([('x',True),('y',False)], False)
, ([('x',True),('y',True)], True)
]
```

Define a function `table :: Prop -> [(Env,Bool)]` that computes the truth table for a proposition. You should need to combine functions from Exercises 5.7 to 5.9 and the evaluation function presented in Lecture 10.

5.11 Modify the solution to the previous exercise to define a function `satisfies :: Prop -> [Env]` that computes all assignments of variables that make a proposition valid. Example:

```
> satisfies (And (Var 'x') (Not (Var 'y')))
[[('x',True),('y',False)]]
```

If the proposition is unsatisfiable, you should get an empty list of results:

```
> satisfies (And (Var 'x') (Not (Var 'x')))
[]
```