- TD -
Tautology checker

The set of propositional formulae, or propositions, is characterised by the following inductive definition:

- the truth values $\top$ and $\bot$ are propositional formulae;
- each propositional variable is a propositional formulae;
- if ϕ is a propositional formula, then $\neg\phi$ is also a propositional formula;
- if ϕ and ψ are propositional formulae, then $\phi \wedge \psi$ and $\phi \implies \psi$ are formulae.

For example, the following are propositions

- $A \wedge \neg B$
- $(A \wedge B) \implies A$
- $A \implies (A \wedge B)$
- $(A \wedge (A \implies B)) \implies B$

A truth assignment is a mapping that associates a truth value with each of the propositional variables. A truth table shows whether a propositional formula is true or false for each possible truth assignment. The meaning of the logical operators can be defined using the classical truth tables.

A tautology is a formula that is true for every possible assignment. We want to define functions which decide if a proposition is a tautology or not.

This exercise is strongly inspired from the one proposed in Chapter 8 of the book Programming in Haskell, Graham Hutton, Cambridge University Press, 2016.

1. Auxiliary functions

Define a function `rmdups` which removes duplicates from a list:

```
rmdups :: Eq a => [a] -> [a]
```

The resulting list doesn't necessarily preserve the order. For example, `rmdups "AABB"` can produce `"AB" or "BA"`.

Propose several versions for this function - involving a recursive definition (using or not `filter`), using `foldl`, sorting the list before removing duplicates.

Hint: for the sort based version you need `Ord a` instead of `Eq a` and you might want to consider using a function grouping together identical elements:

```
group :: Eq a => [a] -> [[a]]
```

Define a function `bools` which builds all the possible lists of length n built out of the booleans `True` and `False`:

```
bools :: Int -> [[Bool]]
```

For example, `bools 3` produces `[[False,False,False],[True,False,False], [True,False,False],[True,True,False],[True,False,False], [True,False,True],[True,True,False],[True,True,True]]`.

Hint: you can observe that each list corresponds to a binary number, by interpreting `False` and `True` as the binary digits 0 and 1 and, given this interpretation, we can think of the function bools as simply counting in binary, from 0 to n. Alternative recursive definitions are also possible.

2. Evaluate propositions

Define a type `Prop` which models the propositional formulae built as defined above. The propositions given as example should be represented as follows:

```
p1 :: Prop
p1 = And (Var 'A') (Not (Var 'A'))
p2 :: Prop
p2 = Imply (And (Var 'A') (Var 'B')) (Var 'A')
p3 :: Prop
p3 = Imply (Var 'A') (And (Var 'A') (Var 'B'))
p4 :: Prop
p4 = Imply (And (Var 'A') (Imply (Var 'A') (Var 'B'))) (Var 'B')
```

In order to evaluate a proposition to a truth value, we need to know the value of each of its variables. For this purpose, define the type `Subst` of substitutions that associate variable names to logical values. For example, the substitution `[('A',False),('B', True)]` assigns the variable `A` to `False`, and `B` to `True`.

Hint: you can use a lookup table like the one defined in the course.

Define the function `evalProp` that evaluates a proposition given a substitution for its variables:

```
evalProp :: Subst -> Prop -> Bool
```

For example, `evalProp [('A',True)] p1` produces `False`.

3. Tautology checker

To decide if a proposition is a tautology, we consider all possible substitutions for the variables that it contains. First, we define a function that returns the variables in a proposition:

```
vars :: Prop -> [Char]
```

For example, `vars p4` produces `"AB"`.

Hint: you can use `rmdups` to eliminate the possible duplicate variables.

We should now define a function `substs` that generates all substitutions for a proposition by extracting its variables and generating, and then zipping the list of variables with each of the resulting lists:

```
substs :: Prop -> [Subst]
```

For example, `substs p4` produces `[[('A',True),('B',True)],[('A',True),('B',False)],[('A',False),('B',True)],[('A',False),('B',False)]]`.

Hint: you can use `bools` to generate all possible lists of logical values for the variables and then zip the list of variables with each of the resulting lists. Alternative recursive definitions are also possible.

Finally, define a function `isTaut` that decides if a proposition is a tautology, by checking if it evaluates to `True` for all possible substitutions:

```
isTaut :: Prop -> Bool
```

For example, `isTaut p4` produces `True`.