Haskell: Factory

- In the folder `/localhost/examen/loginnaam/OPGAVEN`, you'll find the files `Factory.hs` en `FactoryTest.hs`.
  - The file `Factory.hs` contains a template for your solution.
  - First of all, fill in your name, student number and major.

    ```
    -- Jan Jansen
    -- r0123456
    -- master cw
    ```

  - At the top of the file, a number of functions that *might* come in handy have already been imported. **Look up how they work** if you're not familiar with them. You can also **import extra functions and types**.
  - Add you solution into the file `Factory.hs`.
  - You can test your solution using `FactoryTest.hs`. Run the following command in the directory you put the two `.hs`-files in:

    ```
    runhaskell FactoryTest.hs
    ```

    N.B. the fact that all your tests pass doesn't mean that your program is completely correct, nor that you will get the maximum score.
- To hand in your final solution, copy the file
  `/localhost/examen/loginnaam/OPGAVEN/Factory.hs`
  to the file
  `/localhost/examen/loginnaam/haskell.hs`.

Exercise

In this exercise we will model a factory with a number of machines. Each machine takes the resources which he can use from a conveyor belt. Whenever he has enough resources to construct something, he will do this. He never starts taking resources for a second object as long as the first object is not finished.

The leading example will be a car factory:

```
data Resource  = Wheel | Paint | BMWBody | IkeaBody deriving (Ord,Show,Eq)
data Car  = ExpensiveCar | CheapCar deriving (Ord,Show,Eq)
```

To build an expensive car, we need 4 wheels, 2 layers of paint and 1 expensive car body. For a cheap car we only need 1 layer of paint and a cheap car body.

```
expensiveCarStation :: Station Resource Car
expensiveCarStation = machine [(Wheel,4),(Paint,2),(BMWBody,1)] ExpensiveCar

cheapCarStation :: Station Resource Car
cheapCarStation = machine [(Wheel,4),(Paint,1),(IkeaBody,1)] CheapCar
```

Our factory can produce both cars, and the machine producing those expensive cars is the first in line. We also define 2 lists with resources we can put on our conveyor belt.

```haskell
factory :: Station Resource Car
factory = combine [expensiveCarStation,cheapCarStation]

resources1 :: [Resource]
resources1 = concat . replicate 5 $ replicate 6 Wheel ++ replicate 3 Paint ++ [BMWBody,
    IkeaBody]

resources2 :: [Resource]
resources2 = replicate 5 Paint ++ replicate 20 Wheel ++ replicate 5 IkeaBody

resources3 :: [Resource]
resources3 = replicate 6 Wheel ++ replicate 4 Paint ++ [BMWBody, IkeaBody]
```

If we now start our fabric with a list of resources, then we can see which resources are not used at all, and which cars are being built. (Resources which are taken by a machine and not used disappear)

```
Factory> runStation factory resources3
([Paint],[ExpensiveCar])

 Factory> runStation factory resources1
([Paint,IkeaBody,Paint,IkeaBody],
[ExpensiveCar,ExpensiveCar,ExpensiveCar,ExpensiveCar,ExpensiveCar,CheapCar,CheapCar])

Factory> runStation factory resources2
([Paint,Paint,Wheel,Wheel,Wheel,Wheel,Wheel,Wheel,Wheel,Wheel,Wheel,Wheel,Wheel,
Wheel,IkeaBody,IkeaBody,IkeaBody]
,[CheapCar])
```

In the first example, we could build an ExpensiveCar, en there is 1 paint that was not taken by any machine. At the end, the cheapCarStation still held 2 Wheel, 1 Paint and 1 IkeaBody, but that was still not enough to build an extra car. In the second example, the resources entered the machine in a well propertioned way, so many cars could be manufactored. In the third example, there were a lot of resources, but the order of the resources caused that the machines couldn't used most of them, e.g. because they can't take a fifth wheel before their previous car is finished.

Exercise 1: A datarepresentation for a factory

- Define a datatype `Station a b` which represents a working station in a factory, which converts resources of type `a` in objects of type `b`. A station can either be a simple machine, or a sequence of machines which is placed one after another. A machine is defined by its requirements (a list of tuples with a necessary resources of type `a` and the number of resources needed) and the object that the machine produces, of type `b`.

Define the following functions:

- The function `machine :: [(a, Int)] -> b -> Station a b`, which constructs a `Station` which is a simple machine based on a list of resources and its target (the produced object).
- The function `combine :: [Station a b] -> Station a b` which combines multiple stations to a bigger station. This means that the stream of resources will enter the first machine in the list, the resources which are not taken by the first machine, will enter in the second machine in the list. This machines takes what he can use from the conveyor belt, the rest goes on to the third station, and so on.

Exercise 2: A useless machine

Define the function:
`trivial :: (Bounded a,Enum a) => Station a a`.
Which returns a station which consumes all resources, and immediately produces the same resource. A more specific implementation for Bool would be:

```
trivial :: Station Bool Bool
trivial = combine [machine [(True,1)] True, machine [(False,1)] False]
```

Hint: Use `:i` in ghci to see which functions are available to use from the Bounded and Enum type classes.
Hint: If this exercise poses a problem, you can continue with the rest of the exercises without problems.

Exercise 3: A data representation for the status of a machine.

A machine has a number of resources at each instant. Write a datatype (or type-alias) `Resources a` to represent the resources which are held by a machine.
Define the following functions:

- The function `startResources :: Resources a`, which represents the initial state of a machine, when it holds no resources.
- The function `insert :: Ord a => Resources a -> a -> Resources a`, which returns the new state when the machine takes a resource from the conveyor belt.
- The function `amount :: Ord a => Resources a -> a -> Int`, which returns the amount of resources which are held by a specific machine.

Some examples:

```
Factory> amount (insert startResources Wheel) Paint
0
Factory> amount (insert startResources Wheel) Wheel
1
Factory> amount (flip insert Wheel . flip insert Wheel  $ startResources ) Wheel
2
```

Exercise 4: A simple runner for a machine

Define the function:
`run :: Ord a => [(a,Int)] -> b -> Resources a -> [a] -> ([a],[b])`.
This function represents the production of resources in the case of a simple machine. In the function call `run resources target currentResources inputResources` is the meaning of `resources` and `target` the same as in the function `machine`. `currentResources` represents the resources that the machine holds at the beginning of the run, and `inputResources` represents the resources which are passing by on the conveyor belt. The function returns a tuple `(garbage,output)`. `garbage` contains the resources which were not used by the machine, `output` contains the produced resources.
A machine takes a resource from the conveyor belt if he has not enough resources of that type in its current resources. If he has already enough resources of that type, or he can not use this type, this resource is redirected to the `garbage`-output. If the machine has enough resources to produce its output, this output gets produced and the machine will start again without resources. This means that if an `expensiveCarStation` holds 4 wheels, it will not take a fifth wheel, and an IkeaBody will never be taken by such a machine.

Some examples:

```
Factory> run [(Wheel,4)] ExpensiveCar  startResources [Wheel,Wheel,Wheel]
([],[])
```

```
Factory> run [(Wheel,4)] ExpensiveCar  startResources [Wheel,Wheel,Wheel,Wheel]
([],[ExpensiveCar])
Factory> run [(Wheel,4)] ExpensiveCar  startResources [IkeaBody]
([IkeaBody],[])
Factory> run [(Wheel,2)] Bike  startResources [Wheel,Wheel,Wheel,Wheel]
([],[Bike,Bike])
```

Exercise 5: A working factory

Define the function:

`runStation :: Ord a => Station a b -> [a] -> ([a],[b])`

This function runs a simple machine such as in Exercise 4, and for a combined machine, it will first run the first machine on the resource list, the `garbage`-output will be given to the second machine, and so on. The produced b's will be put after each other in order of the machines who produce them. Examples of this are present in the introduction.

Have fun(ction)!