Haskell: Fabriek

Enkele praktische afspraken

- In de map `/localhost/examen/loginnaam/OPGAVEN` vind je de bestanden `Factory.hs` en `FactoryTest.hs`.
  - In het bestand `Factory.hs` staat reeds een template voor je oplossing.
  - Als eerste vul je bovenaan je naam, studentennummer en richting in.

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

  - Bovenaan in het bestand worden eventueel reeds een aantal functies geïmporteerd die handig *zouden* kunnen zijn. **Zoek hun werking op** indien je ze nog niet kent. Je mag ook **extra functies en types importeren**.
  - Pas het bestand `Factory.hs` aan met de oplossing.
  - Je kan je oplossing testen m.b.v. `FactoryTest.hs`. Dit doe je door in de map waarin de twee `.hs`-bestanden staan het volgende commando uit te voeren:

    ```
    runhaskell FactoryTest.hs
    ```

    N.B. dat alle testen slagen betekent niet per se dat je programma helemaal correct is of dat je het maximum van de punten verdient.

- Om je oplossing in te dienen kopieer je het bestand
  `/localhost/examen/loginnaam/OPGAVEN/Factory.hs`
  naar het bestand
  `/localhost/examen/loginnaam/haskell.hs`.

Oefening

In deze oefening zullen we een fabriek met een aantal machines modelleren. Elke machine neemt grondstoffen die hij kan gebruiken van een lopende band, en als hij genoeg resources heeft om iets te construeren dan doet hij dat. Hij begint nooit met resources te nemen voor een tweede object zolang het eerste object niet is afgewerkt.

We werken hier een voorbeeld uit voor een autofabriek:

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

Om een dure auto te bouwen heben we 4 wielen, 2 lagen verf en 1 dure carroserie nodig, voor een goedkope slechts 1 laag verf en een goedkopere carroserie.

```
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
```

Onze fabriek kan beide auto's produceren, en de machine die dure auto's maakt krijgt voorrang aangezien hij de eerste in de lijst staat. We definiëren ook 2 lijsten met resources die we door onze fabriek kunnen laten lopen.

```
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]
```

Als we nu onze fabriek opstarten met een lijst resources dan zien we welke resources er ongebruikt zijn gebleven, en welke auto's er zijn gebouwd. (Resources die vastgenomen worden door een machine die dan toch niet gebruikt worden verdwijnen)

```
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 het eerste voorbeeld, kon een ExpensiveCar gebouwd worden en is er 1 paint door geen enkele machine opgenomen. In de cheapCarStation zit dan nog 2 Wheel, 1 Paint en 1 IkeaBody, maar dat was niet genoeg om de auto mee te bouwen. In het tweede voorbeeld kwamen de resources relatief goed verdeeld binnen bij de machines, en konden er dus veel auto's gemaakt worden. Bij het derde voorbeeld, waren er veel resources, maar de volgorde van de resources zorgde ervoor dat de machines veel resources niet konden gebruiken, omdat ze bijvoorbeeld geen 5de wiel mogen nemen voor hun vorige auto afgewerkt is.

Opdracht 1: Een datarepresentatie voor de fabriek

- Maak een datatype aan `Station a b` aan, dit stelt een werkstation voor in een fabriek dat resources van type `a` omzet in objecten van type `b`. Een werkstation is ofwel een machine, ofwel een reeks andere werkstationnen na elkaar geplaatst. Een machine is gedefinieerd door zijn requirements (een lijst van tupels met daarin telkens een nodige resource van het type `a` en het aantal dat de machine nodig heeft), en het object dat de machine maakt, van het type `b`.

Maak ook volgende functies:

- De functie `machine :: [(a, Int)] -> b -> Station a b`, die een `Station` construeert die een machine is op basis van een lijst van resources en zijn target (het geproduceerde object).

- De functie `combine :: [Station a b] -> Station a b` die meerdere werkstationnen combineert tot een groot werkstation. Dit betekent dat de stroom van resources eerst zal binnenkomen in het station dat het eerst in de lijst voorkomt, dan zullen de resources die niet worden vastgenomen door dat eerste station, in het tweede station in de lijst binnenkomen. Die neemt van de band wat hij kan gebruiken, en dat gaat door naar het derde station, en zo verder.

Opdracht 2: Een nutteloze machine

Definieer de functie:
`trivial :: (Bounded a,Enum a) => Station a a`.
Die een station teruggeeft die alle resources consumeert en terug uitvoert. Een meer specifieke implementatie voor Bool zou zijn:

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

Hint: Gebruik `:i` in ghci om te zien van welke functies je kunt gebruiken maken uit de Bounded en Enum type classes.
Hint: Als deze deelvraag niet lukt, kun je zonder problemen de rest van de opgave maken.

Opdracht 3: Een datarepresentatie voor de status van een machine

Een machine heeft op ieder moment een aantal resources, schrijf naar keuze een datatype of een type-alias `Resources a` om de Resources die een machine vast heeft, voor te stellen.
Definieer hierop de volgende functies:

- De functie `startResources :: Resources a`, die de initiële toestand van een machine voorstelt, wanneer die geen resources vastheeft.
- De functie `insert :: Ord a => Resources a -> a -> Resources a`, die de nieuwe toestand voorstelt als een machine, een nieuwe resource vastneemt
- De functie `amount :: Ord a => Resources a -> a -> Int`, die teruggeeft hoeveel van die resources de machine al vastheeft.

Enkele voorbeelden:

```
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
```

Opdracht 4: Een simpele runner voor een machine

Definieer de functie:
`run :: Ord a => [(a,Int)] -> b -> Resources a -> [a] -> ([a],[b])`.
Deze functie stelt de productie van resources voor in het geval het om een simpele machine gaat. In de oproep `run resources target currentResources inputResources`, hebben `resources` en `target` dezelfde betekenis als bij de functie `machine`. `currentResources` stelt de resources voor die de machine vastheeft aan het begin van de run, en `inputResources` stelt de resources voor die aan de machine passeren. De functie geeft een tupel `(rommel,output)` terug. In `rommel` zitten de resources die de machine niet heeft kunnen gebruiken, in `output` zitten de resources die de machine heeft geproduceerd.
Een machine neemt een resource van de band als hij nog niet genoeg resources van dat type heeft in zijn huidige resources. Als hij al genoeg resources heeft van dat type, of hij kan die resources niet gebruiken,

komt die terecht in de `rommel`-output. Als de machine genoeg resources heeft om zijn output te produceren, wordt deze geproduceerd en start de machine opnieuw zonder resources. Dit betekent dat als een `expensiveCarStation` al 4 wielen vastheeft, het een 5de wiel zal laten passeren, en een IkeaBody in elk geval zal laten passeren.

Enkele voorbeelden:

```
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])
```

Opdracht 5: Een werkende fabriek
Definieer de functie:
`runStation :: Ord a => Station a b -> [a] -> ([a],[b])`
Deze functie laat een simpele machine lopen zoals in opdracht 4, en voor een gecombineerde machine laat deze eerst de machine lopen op de hele resource-list hij binnenkrijgt, de `rommel`-output wordt dan gegeven als invoerlijst aan de tweede machine, en zo verder. Al de geproduceerde b's worden achter elkaar gezet in volgorde van de machine die deze b's produceert. Voorbeelden hiervoor staan in de inleiding van de oefening.

Have fun(ctie)!