# Functioneel Programmeren

Project 3 - Sviatoslav Harasymchuk

## Inleiding

Ik zal beginnen met vermelden dat ik bijna geen code uit mijn "Project 2 - Colorban" gebruikt heb. Toen had ik ervoor gekozen om een datatype `World` te gebruiken die een lijst van `FieldObject` bijhoudt en elke object in die lijst heeft het veld `type` met bijbehorend type van object. Dat is duidelijk niet zo netjes dus nu heb ik voor een datatype `Level` (zie `libGameData.hs`) veel verschillende arrays met verschillende types van objecten in het spel.

In feite door deze verandering kon ik bijna geen enkele lijn code van mijn vorige project gebruiken. Mijn logica voor het bepalen van `isSolved :: World -> Bool`, `canMove :: ... -> Bool` en dergelijke functies moesten ook herschreven worden.

Verder leg ik in grote lijnen het ontwikkelingsproces uit.

## Parsec

Ik maak gebruik van bibliotheek `Parsec` om de configuratiebestand te parsen. Om eerlijk te zijn: eerste uur (of 2) heb ik meer "gevochten" met Parsec dan iets nuttig geparsed, dus het was een teken om tutorials te gaan zoeken.

Ik ging bottom->up tewerk dus eerst een layout veld parsen, dan een lijn met layout velden, dan een array van lijnen en zo verder.

In mijn `lib/Parser` directory zie je dat ook aan file namen die telkens voor het parsen van een array elementen staan.

Afzonderlijke parsers gebruiken allemaal `MyParser.hs` die alle nodige abstracties voorziet. Belangrijkste ervan is `parseArray :: String -> Parser a -> Parser [a]`. Deze functie is met veel "trial and error (en tranen)" gemaakt en werkt eindelijk zoals het moet. Nog een belangrijke functie is `whitespaceTillNewLine`, `parseTabsCount` en `parseTabs`, die worden gebruikt om het juiste aantal tabs te parsen **voor** het array element. Eigenlijk is dat ook de enige plaats waarin de tabs geforceerd worden. Dat was nodig om het parsen van meerdere levels mogelijk te maken: bij het parsen van bv. coins array (laatste array in het level) weten wij niet zeker of volgende array element een coin is of al level 2 is.

Voor het parsen van kleuren, tegels, coördinaten en float (met infinite) heb ik typeclass `Read` met functie `readsPrec :: Int -> ReadS FColor` voor verschillende datatypes (`IntF`, `FColor` en `TileType`) geinstantieerd. Instantiatie van een tuple `Coordinate` was immers niet nodig omdat dit al door haskell geïmplementeerd is.

## Typeclasses

Om de code duplicatie te verminderen heb een aantal eigen typeclasses gemaakt.

### BoardObject (`lib/BoardObject.hs`)

```
class BoardObject b where
  coordinate :: b -> Coordinate
  otype :: b -> ObjectType
  color :: b -> FColor
  color _ = None
```

Datatypes zoals `Robot`, `Crate`, `Tile`, `Storage` en andere implementeren deze klasse. Deze zal gebruikt worden bij het beslissen of de gegeven move/push geldig is in `lib/Game.hs` (eens we weten dat er een kist is moeten wij nagaan of die kist de kist erachter kan duwen: functie voor robot = functie voor kist, zie later).

### Drawable (`src/Drawable.hs`)

```
class (BoardObject a) => Drawable a where
  getImagePath :: a -> String
  getImagePath obj = ...
  draw :: a -> Textures -> Picture -> Picture
  draw _ _ p = p
```

Alle datatypes die `BoardObject` implementeren, implementeren ook `Drawable`. `Drawable` is een typeclass die aan een BoardObject de verantwoordelijkheid delegeert om die op het scherm te tonen. Zo zal elke object zelf beslissen welke image/texture die gebruikt en hoe het wordt getekend.

## Game.hs

Wij komen eindelijk aan de hart/engine van Colorban-v2 met alle regels die het spel implementeert.

Belangrijkste functies zijn hier `move`, `isSolved`, `selectNextRobot` en `selectPreviousRobot`.

Als extra functionaliteiten heb ik gekozen om deze 3 te implementeren:

```
- 1. Deuren en knoppen
- 4. Bewegende platformen over leegtes
- 7. Geld verzamelen
```

Puntje `1` en `7` zijn successvol geïmplementeerd maar puntje `4` niet. Dat is te verklaren door het feit dat ik veel meer dan 24-32 uur aan het project heb gewerkt (het getal ligt dicht bij 50). De reden hiervoor is misschien door de grote structuur veranderingen dat ik heb gemaakt (vermeld in inleiding).

Aan andere kant heb ik alle andere functionaliteiten beter dan verwacht gemaakt wat volgens mij ook goed is.

Terug naar onze engine. Functie `move` gaat na of de robot kan bewegen:

```
canPush level robot (strength robot) direction
```

De functie `canPush` is recursief voor alle volgende kisten, maar niet voor robots (robot kan andere robots niet duwen).

Als `canPush True` geeft, dan zullen wij `moveRobot` oproepen die `pushCrates` oproept (die ook recursief is). Dan zullen wij volgende uitvoeren:

```
handleCoins $ handleDoorsAndButtons $ handleSpots
```

Deze 3 functies zullen de states van deuren, spots en munten veranderen (indien nodig). Bijvoorbeeld er zal een munt verdwijnen als er een robot op dezelfde positie staat en de waarde zal bij de collectedCoins toegevoegd worden.

Hoe bepalen wij nu of de level opgelost is?

```
isSolved :: Level -> Bool
isSolved level = enoughCoins && cratesSolved
  where
    enoughCoins = requiredCoins level <= collectedCoins level
    cratesSolved = all (\crate -> not $ null $ filter' (coordinate crate) $ storages level)
```

Simpel! Voor alle kisten moet er op zijn positie een opslagplaats aanwezig zijn. Ah ja, en natuurlijk moeten de robots genoeg geld verzameld hebben.

## Windows

Er zijn 3 windows:

- `MenuWindow`
- `GameWindow`
- `EndGameWindow`

Er is ook een belangrijke datatype:

```
data GameData = GameData
  { windowType :: WindowType,
    levels :: [Level],
    playingLevel :: Maybe Level,
    playingLevelSolved :: Bool
  }
```

`GameData` houdt bij welke window gerendert moet worden. In `Main` delegeren wij gewoon telkens alle events (en renderings) naar 1 van die 3 schermen.

Ik zal niet te diep in detail gaan over rendering van alle objecten op het scherm omdat ~~dit saai is~~ ik vind dat Gloss (en haskell in het algemeen) niet zo goede keuze is om een goede UI te maken…

## Monad transformers

Mijn kennis over monad transformers heb ik `src/Main.hs:50` toegepast voor volgende functies: `readConfigFile` en `getGameData`.

Deze functies worden binnen de MaybeT IO monad transformer uitgevoerd, wat helpt met mogelijke fouten af te handelen (gerepresenteerd door Nothing) en/of IO acties uitvoeren.

## Testen

Ik heb 10 testen geschreven om de nieuwe functionaliteiten te testen. Ze maken gebruik van triviaal op te lossen levels uit `levels/example.txt`.

Ik vond het overbodig om mijn parser te testen omdat dit maar een layer boven Parsec is en niet te moeilijk geimplementeerd is.