Project Functioneel Programmeren: Parser voor een MBot-programmeertaal: **BurgieScript**

Rien Maertenis
3de Bachelor Informatica

16 januari 2017

Op het moment van indienen (zondagavond 15 januari) staat de MBot-patch met de toegevoegde `playNote` functie nog niet online op Hackage. Daarom is het bestand `MBot.hs` meegeleverd in het zip bestand. Zodra de patch online staat volstaat het normaal gezien om de MBot-package toe te voegen aan het cabal bestand.

1 Inleinding

Voor dit project heb ik de *BurgieScript* programmeertaal uitgevonden. Het is een knipoog naar het stereotype van een burgerlijk ingenieur. Natuurlijk is de taal volledig in het Nederlands, gebaseerd op de taal gebruikt in cursussen gegeven door Burgerlijk ingenieur-professoren en door een document met Nederlandstalige Computertermen [1] van de vakgroep ELIS aan de UGent. Schrik dus niet als u termen tegekomt zoals *OntleedFout* (*ParseError*). Het grootste deel van de code waarin de parser geschreven is, is natuurlijk wel in gewoon Engels geschreven. Kwestie van het mezelf ook niet té moeilijk te maken.

De programmeertaal is grotendeels gebaseerd op pseudocode. Maar als keywords heb ik oubolige en lange woorden gebruikt. Daarnaast zult u ook enkele designkeuzes in de programmeertaal zelf tegenkomen die enkele vraagtekens zullen oproepen. Ook deze dienen met een grove korrel zout genomen te worden. Ik ben heus niet écht overtuigd dat \ en / een goed alternatief zijn voor { en } om blokken code aan te duiden. Maar wees gerust, in tegenstelling tot *BurgieScript* heb ik geprobeerd om mijn Haskell-code zo consistent en overzichtelijk mogelijk te schrijven.

[1] `https://www.elis.ugent.be/node/285`

2 Syntax

Hieronder kunt u de syntax van *BurgieScript* vinden in EBNF[2]-achtige vorm: `(...)` stelt een groep voor, `[...]` is optioneel en alles tussen {...} kan herhaald worden. Er wordt geen gebruik gemaakt van aaneenschakelende komma's of terminerende puntkomma's.

BurgieScript.bnf

```
(* BurgieScript EBNF *)

(* Statements *)
<block>             = "\" <body> "/"
<body>              = { <statement> }
<statement>         = <loop> | <conditional> | <command> | <assignment>
                    | <comment>
<loop>              = "Gedurende" <expr> "doe:" <block>
<conditional>       = "Indien" <expr> "doe:" <block>
                    [ "Anderzijds:" <block>]
<assignment>        = "Zet variabele" <variable> "op" <expr> "."
<comment>           = "Terzijde:"  <character> ("." | "?" | "!")
<command>           = ( <motor> | <sound> | <light> | <sleep> ) "!"


(* Commando's *)
<motor>             = "Halt" | "Chauffeer" <direction>
<sound>             = "Zing" <duration> "een" <note>
<light>             = "Kleur" <color> <side>
<sleep>             = "Rust kort" | "Neem op je gemak een pauze"
                    | "Sluit je ogen maar voor even"
                    | "Droom zacht zoete prins"

<direction>         = "rugwaarts" | "voorwaarts" | "te" <side>
<side>              = "bakboordzijde" | "stuurboordzijde"
<duration>          = "kortstondig" | "eventjes" | "langdurig"
                    | "erg langdurig"
<note>              = "do" | "re" | "mi" | "fa" | "sol" | "la" | "si"
<color>             = "rood" | "groen" | "blauw" | "wit"

(* Expressies *)
<expr>              = <comp> [{ <compop> <comp> }]
<compop>            = "overeenkomstig met" | "verschillend met"
                    | "significanter dan" | "minder significant dan"
<comp>              = <term> [{ <termop> <term> }]
<termop>            = "+" | "-"| "hetzij"
<term>              = <fact> [{ <factop> <fact> }]
<factop>            = "x" | ":" | "tevens"
<fact>              = [ "-" "allesbehalve" ] <base>
<base>              = <literalbool> | <literalnum> | "(" <expr> ")"
                    | <query> | <variable>
<literalbool>       = "waarachtig" | "strijdig"
<literalnum>        = { <digit> } [ "," { <digit> } ]
```

[2] https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form

```
44 <query>              = "GeluidWeerkaatsingsApparaatWaarde"
45                      | "LijnVolgApparaatIsWit" <side>
46 <variable>           = { <letter> }
```

Nog een opmerking: *BurgieScript* legt geen verplichtingen op hoe de code geformatteerd moet worden. Hoewel er aanbevolen wordt om ieder statement op een nieuwe lijn te zetten en de statements in de blokken te indenteren, is dit helemaal niet noodzakelijk. Sterker nog: behalve enkele speciale gevalen (een spatie na een variabelenaam en tussen vergelijkingsoperatoren) mogen newlines, spaties, tabs en andere whitespace-karakters achterwege gelaten worden. Dit is bijvoorbeeld een volledig correct programma:

```
GedurendeLijnVolgApparaatIsWitteBakboordzijdedoe:\Chauffeerrugwaarts!/
```

Maar ik raad u af om zo'n programma's te schrijven.

3 Semantiek

Eerst en vooral begin ik met het opdelen van alle constructies in drie onderdelen: statements die de *flow* van een programma bepalen, commando's die de MBot aansturen en expressies die mathematische en booleaanse waarden verwerken en opvragen. In de code is voor elke categorie een eigen parser en evaluator voorzien. Deze opdeling is puur om de Haskell-bestanden niet te lang te maken en ze overzichtelijk te houden.

3.1 Statements

In de programmeertaal zijn er vijf soorten statements die hieronder worden opgesomd. Een blok statements wordt gebruikt bij de conditionele uitvoering en lussen. Een blok begint met een `\`, gevolgd door de statements die tot dit blok behoren (een blok mag ook leeg zijn), afgesloten door een `/`. Om esthetische redenen wordt aanbevolen om de statements in een blok met één spatie te indenteren.

Commentaar begint met het keyword `Terzijde:` en eindigt bij het volgende punt, vraagteken of uitroepteken. Commentaar heeft geen invloed op de evaluatie.

```
    Terzijde: Dit is commentaar.
    Terzijde: En dit ook?
    Terzijde: Jazeker!
```

Assignatie van de waarde van een expressie aan een variabele wordt gedaan met de zin `Zet variabele <naam> op <expr>.` (inclusief de punt op het einde). Na de evaluatie van dit statement kan de variabele gebruikt worden in expressies. Geldige variabelenamen bestaan uit één of meerdere alfabetische karakters en moeten gevolgd worden door een spatie, newline of ander whitespace-karakter.

```
Zet variabele antwoord op 41 + 1.
Zet variabele vals op strijdig.
```

Conditionele uitvoering van statements gebeurt door te beginnen met `Indien <expr> doe:` en daarna de statments die moeten uitgevoerd worden als de expressie geldig is. Optioneel kan daarna nog `Anderzijds:` komen, gevolgd door een blok dat moet uitgevoerd worden wanneer de expressie niet geldig is.

```
Indien waarachtig doe:
\
 Chauffeer te bakboordzijde!
/
Anderzijds:
\
 Chauffeer te stuurboordzijde!
/
```

Lussen worden op dezelfde manier verkregen als de conditionele blokken, maar dan beginnend met `Gedurende <expr> doe:` gevolgd door een blok.

```
Terzijde: programma-executie kan vroegtijdig afgebroken worden
doormiddel van Control-C.
Zet variabele overflow op 0.
Gedurende overflow + 1 significanter dan 0 doe:
\
 Zet variabele overflow op overflow + 1.
/
```

Commando's die kunnen gebruikt worden om de MBot aan te sturen. Deze worden besproken in de volgende subsectie.

3.2 Commando's

Deze worden gebruikt om de MBot aan te sturen. Commando-statements eindigen allemaal met een uitroepingsteken. De programmeur heeft keuze tussen de volgende 'bevelen':

Motor commando's worden gebruikt om de motor van de MBot aan te sturen. Een motorcommando begint met `Chauffeer` en wordt gevolgd door een richting. Om de MBot te stoppen kunt u het commando `Halt!` gebruiken.

```
Chauffeer voorwaarts!
Chauffeer rugwaarts!
Chauffeer te bakboordzijde!
Chauffeer te stuurboordzijde!
Halt!
```

Ledjes kunnen een rood, groen, blauw of wit kleuren en uitgeschakeld worden.

```
Kleur rood te bakboordzijde.
Kleur blauw te stuurboordzijde.
Verduister te bakboordzijde.
Verduister te stuurboordzijde.
```

Slapen. De robot kan enkele momenten niets doen door de volgende slaap-commando's uit te voeren. Merk op dat de MBot wel nog blijft verder rijden indien een motorcommando werd uitgevoerd. Geordend van kort naar lang:

```
Rust kort!
Neem op je gemak een pauze!
Sluit je ogen maar voor even!
Droom zacht zoete prins!
```

Zingen behoort ook tot de mogelijkheden van de MBot, met een gepatchte MBot-bibliotheek kan de MBot een volledige toonladder afspelen. Muzieknoten afspelen met de ingebouwde buzzer kan als volgt:

```
Zing eventjes een sol.
Zing erg langdurig een do.
```

3.3 Expressies

De voorbeeld-parser uitgewerkt in de slides is niet zo flexibel: de gebruiker is verplicht haakjes te gebruiken voor iedere operatie. Om op een flexibelere manier wiskundige expressies te parsen heb ik mij gebaseerd op de paper/tutorial *Monadic Parsing in Haskell*[3]. Meer info over hoe expressies precies geparsed worden kunt u in de sectie met uitleg over de implementatie lezen.

[3] `http://www.cs.nott.ac.uk/~pszgmh/pearl.pdf`

3.3.1 Het typesysteem

BurgieScript kent twee types in zijn expressies: boolese waarden en nummers. Intern worden deze beide voorgesteld als numers, maar iedere waarde is gemerkt met één van de twee types. Tijdens het parsen wordt niet gekeken naar dit type, maar wanneer je een bijvoorbeeld een optelling probeert uit te voeren op twee boolese waarden krijg je een `TypeFout` tijdens de uitvoering. Meer info krijgt u later.

3.3.2 Wat kan er allemaal gebruikt worden in expressies?

Literalen: de boolese literalen zijn `waarachtig` en `strijdig`. Een nummer wordt voorgesteld door een opeenvolging van cijfers, al dan niet gevolgd door een komma en een nieuwe opvolging van cijfers die de fractie van een kommagetal voorstellen.

Variabelen: iedere opeenvolging van alfabetische karakters gevolgd door een spatie wordt als variabele gezien (tenzij woorden die een andere betekenis hebben, zoals `waarachtig` en `GeluidWeerkaatsingsApparaatMeetwaarde`.

Haakjes: haakjes kunnen ook gebruikt worden. Alles binen de haakjes wordt eerst uitgewerkt.

Opvragingen: deze gaan een waarde opvragen aan de robot. De drie opvragingen zijn:

```
LijnVolgApparaatIsWit te bakboordzijde      -> boolese waarde
LijnVolgApparaatIsWit te stuurboordzijde    -> boolese waarde
GeluidWeerkaatsingsApparaatWaarde           -> numerieke waarde
```

3.3.3 Wat kan je allemaal doen?

U kunt de gegeven voorbeelden zelf uitvoeren door een interactieve sessie te starten en de string die u wilt parsen en evalueren mee te geven met de `execExp` functie:

```
execExp "1 + 1"
> Right (2.0, NumT)
execExp "waarachtig tevens strijdig"
> Right (0.0, BoolT)
```

Zoals vermeld kan *BurgieScript* relatief ingewikkelde wiskundige expressies verwerken die worden uitgevoerd zoals je zou verwachten. Enkele voorbeelden:

Deling en multiplicatie: Voor deling en multiplicatie wordt er gebruik gemaakt van de symbolen `x` en `:` .

```
1 : 3 -> 0.333333
3 x 2 -> 6
```

Volgorde van bewerkingen: er wordt correct omgegaan met precedentie, zowel bij nummers als booleans. Ook kunnen meerdere dezelfde operaties na elkaar staan. De volgende expressies geven bijvoorbeeld het correcte resultaat:

```
1 + - 2 x 3 + 4  -> -1
1 x - 2 + 3 x 4  -> 10
2 x 2 x 2 x 2    -> 16
waarachtig tevens waarachtig hetzij strijdig -> waarachtig
allesbehalve waarachtig tevens waarachtig    -> strijdig
strijdig hetzij strijdig hetzij waarachtig   -> waarachtig
```

Unaire operatoren: het verschil tussen de binaire en de unaire min-operator wordt herkend en de unaire operator kan gebruikt worden op alle expressies. Zo wordt het getal -1 geparsed als de unaire min-operator toegepast op de een literaal 1. Dit maakt het mogelijk om een min te plaatsen voor haakjes, variabelen en sensormetingen of de unaire en binaire min te combineren:

```
- ( 1 + 1 ) -> -2
- 10 - - 10 -> 0
allesbehalve ( strijdig tevens waarachtig) -> waarachtig
```

Vergelijkingen: numerieke expressies kunnen vergeleken worden met elkaar en geven een boolean terug. Een vergelijking heeft de laagste precedentie.

```
1 overeenkomstig met 1                       -> waarachtig
1 + 2 x 3 significanter dan 1 x 2 + 3        -> waarachtig
allesbehalve ( 2 : 3 verschillend met 4 : 6) -> waarachtig
```

4 Voorbeeldprogramma's

4.1 Politiewagen

Het eerste voorbeeldprogramma is simpel: doe een politiewagen na door de ledjes te doen knipperen. Als extraatje speelt er ook een sirene af.

programs/police.bs

```
Gedurende waarachtig doe:
\
 Zet variabele wissels op 10.
 Gedurende wissels significanter dan 7 doe:
 \
  Kleur rood te bakboordzijde!
  Kleur wit te stuurboordzijde!
  Zing langdurig een do!

  Kleur wit te bakboordzijde!
  Kleur blauw te stuurboordzijde!
  Zing langdurig een sol!

  Zet variabele wissels op wissels - 1.
 /
 Gedurende wissels significanter dan 0 doe:
 \
  Kleur rood te bakboordzijde!
  Kleur blauw te stuurboordzijde!
  Zing kortstondig een do!

  Kleur blauw te bakboordzijde!
  Kleur rood te stuurboordzijde!
  Zing kortstondig een sol!

  Zet variabele wissels op wissels - 1.
 /
/
```

Het programma werkt door eerst drie keer traag te flikkeren en vervolgens zeven keer ietsje sneller af te wisselen om daarna de cyclus te herhalen (in een oneindige lus). Hiervoor wordt een variabele `wissels` gebruikt waarin opgeslagen wordt hoeveel keer we nog moeten wisselen.

Na iedere twee wissels (een lichtje wordt eerst blauw, daarna rood) wordt deze variabele gedecrementeerd.

Of lichtjes snel of traag flikkeren wordt bepaald door hoe lang er gezongen wordt. Een geluid afspelen met de buzzer blokkeert namelijk de commando's zo lang er geluid afgespeeld wordt.

4.1.1 Hindernissenparcours

Als tweede voorbeeldprogramma probeert de robot rechtdoor te rijden en wanneer er een obstakel geregistreerd wordt door de sensors probeert de robot die te ontwijken.

programs/obstacle.bs

```
Gedurende waarachtig doe:
\
 Zet variabele afstand op GeluidWeerkaatsingsApparaatWaarde.
 Zet variabele bakboordlicht op waarachtig.

 Indien afstand significanter dan 20 doe:
 \
  Kleur groen te stuurboordzijde!
  Kleur groen te bakboordzijde!

  Chauffeer voorwaarts!
 /
 Anderzijds:
 \
  Halt!

  Indien afstand significanter dan 5 doe:
  \
   Kleur blauw te stuurboordzijde!
   Kleur blauw te bakboordzijde!

   Chauffeer te bakboordzijde!
   Neem op je gemak een pauze!
  /
  Anderzijds:
  \
   Kleur rood te stuurboordzijde!
   Kleur rood te bakboordzijde!

   Neem op je gemak een pauze!
   Chauffeer rugwaarts!

   Terzijde: Ga achteruit tot we ver genoeg van het obstakel zijn.
   Gedurende GeluidWeerkaatsingsApparaatWaarde minder significant dan 20
    doe:
   \
    Terzijde: Alterneer wit en rood om te waarschuwen dat we achteruit
    gaan.
    Indien bakboordlicht doe:
    \
     Kleur wit te stuurboordzijde!
     Kleur rood te bakboordzijde!
    /
    Anderzijds:
    \
     Kleur rood te stuurboordzijde!
     Kleur wit te bakboordzijde!
    /
    Zet variabele bakboordlicht op allesbehalve bakboordlicht.

    Rust kort!
   /
```

```
  /
 /
/
```

Ook hier zit het programma weer in een oneindige lus. Deze begint met het registreren van de sensorwaarde in de variabele `afstand`.

Indien deze afstand groter is dan 20 is er niet metteen gevaar dat de robot ergens tegen botst. De robot gaat dan gewoon vooruit en de lichtjes worden op groen gezet.

Indien deze afstand tussen 20 en 5 is moeten we een obstakel ontwijken. We sturen naar bakboord (links) en onze ledjes worden blauw. Er wordt even gewacht om de robot tijd te geven een redelijke hoeveelheid te draaien.

Indien de afstand kleinder dan 5 is zitten we heel dicht bij een obstakel. Dan in het gewoon het beste om een paar stappen terug te nemen. De lichtjes wisselen rood en wit af en er wordt achteruit gereden tot de sensors een afstand groter dan 20 registreren. Zo kan de robot terug wat dichter rijden om vervolgens naar links te draaien (omdat we dan terug een afstand tussen 20 en 5 hebben).

4.2 Lijn volgen

Als laatste voorbeeldprogramma probeert de robot een zwarte lijn op een witte achtergrond te volgen.

programs/follow.bs

```
Gedurende waarachtig doe:
\
 Terzijde: Plaats waar de lijn is.
 Zet variabele bakboord op allesbehalve LijnVolgApparaatIsWit te
    bakboordzijde.
 Zet variabele stuurboord op allesbehalve LijnVolgApparaatIsWit te
    stuurboordzijde.
 Zet variabele laatsteBakboord op waarachtig.
 Indien bakboord tevens stuurboord doe:
 \
  Chauffeer voorwaarts!
 /
 Anderzijds:
 \
  Indien bakboord doe:
  \
   Chauffeer te bakboordzijde!
   Zet variabele laatsteBakboord op waarachtig.
  /
  Anderzijds:
  \
```

```
   Terzijde: Rechts is wit, of niets is zwart.
   Indien stuurboord doe:
   \
    Chauffeer te stuurboordzijde!
    Zet variabele laatsteBakboord op strijdig.
   /
   Anderzijds:
   \
    Indien laatsteBakboord doe:
    \
     Chauffeer te bakboordzijde!
    /
    Anderzijds:
    \
     Chauffeer te stuurboordzijde!
    /
   /
  /
 /
/
```

Opnieuw maken we gebruik van een oneindige lus en beginnen we met twee variabelen te zetten: in `bakboord` wordt opgeslagen of de lijnsensors een zwarte lijn zien aan de linkerkant (die kant is dus niet wit), in `stuurboord` wordt opgeslagen of een lijn aan de rechterkant werd gevonden. De variabele `laatsteBakboord` zal bijhouden of we het laatst naar links of naar rechts zijn gedraaid.

Indien beide kanten op de lijn zitten kunnen we gewoon vooruit gaan.

Als de lijn enkel te zien is aan de linkerkant wordt er ook naar links gedraaid en wordt de variabele `laatsteBakboord` op `waarachtig` gezet. Als de lijn enkel te zien is aan de rechterkant wordt het omgekeerde gedaan.

Indien beide kanten geen lijn zien wordt er gedraaid naar de kant waar het laatst een lijn werd gezien. Dus naar links wanneer `laatsteBakboord` op `waarachtig` staat en anders naar rechts.

5 Implementatie

5.1 De parser

In `Parser.hs` kunt de algemene parser-functies vinden. De parser is een kruising tussen de parser getoond in de slides en de parser zoals uitgewerkt in de paper *Monadic Parsing in Haskell* (zoals vermeld bij de semantiek-bespreking van de expressies). Daarnaast heb ik ook enkele extra's toegevoegd, zo geeft de `parse`-functie (lijnnummer 122) een `Either`

`Error a` terug (de definitie van Error staat op lijn 21). Zodat het falen van de parser mooi kan afgehandeld worden.

Nog enkele noemenswaardige functies:

`chooseFrom` (lijn 134) aanvaard een lijst van parsers en gaat de eerste gebruiken die succesvol is. Met deze functie worden lange reeksen van `parserA <|> parserB <|> ...` vermeden.

`parseStrTo` (lijn 173) neemt een lijst van tuples van een string en een ander item (bijvoorbeeld `("bakboordzijde", Portside)` en geeft een parser terug die het tweede item van de eerste tuple teruggeeft waarvan de string matcht.

`chain` (lijn 187) deze functie maakt de magie van de expressies mogelijk. Deze is deels overgenomen van de eerder vermelde paper/tutorial (met nog een toevoeging). Er worden twee parsers meegegeven: een parser die een `Exp` parset en een parser die een binaire operator parset. De functie begint met het parsen van een expressie en slaat die voorlopig op in de variabele `a`. Daarna wordt er geprobeerd om de binaire operator te parsen als `bif` en wordt er een recursieve oproep gedaan waarvan het resultaat opgeslagen wordt als `b`.

Lukt het niet om een binaire operator te parsen dan wordt enkel `a` gereturned. Lukt dit wel dan wordt de expressie teruggegeven die de binaire functie toegepast op `a` en `b` voorsteld. De recursieve oproep zorgt ervoor dat het tweede operand een expressie met een hogere precedentie kan zijn, of nogmaals een binaire operator toegepast op een expressie. Hierdoor kan `1 + 1` geparsed worden, maar ook `1 + 1 + 1 + 1`. Deze recursieve stap is een toevoeging op de functie in de vermelde paper.

Deze functie kunt u in actie zien vanaf lijn 237, daar begin ik met het 'chain'-en van expressies met de laagste expressie en ga dan telkens een niveau hoger tot ik uitkom bij literalen, variabelen ...

`parseStatements` (lijn 436). Het parsen van een statement begint met deze functie en vanaf daar splitst iedere parser zich op in kleinere deelparsers. Zo zal deze parser proberen om `parseStat` op te roepen die op zijn beurt opsplitst in `parseComment`, `parseAssign`, ... Wanneer het niet lukt om één of meer statements te parsen wordt een lege lijst teruggegeven. Dit maakt het mogelijk om recursief te werken en zorgt er ook voor dat een programma of bok code zonder statements ook een geldig programma is.

In `Parser.hs` staan de gemeenschappelijke parser-functies en hulpfuncties. Daarnaast zijn er nog de specifieke bestanden `ExpressionParser.hs` (lijn 197), `CommandParser.hs` (lijn 320) en `StatementParser.hs` (lijn 422) waarin de specifieke parsers staan.

5.2 Evaluators

Er zijn drie verschillende 'Evaluators', dit zijn combinaties van transformer monads die elk specifieke delen van het programma uitvoeren. hier volgt hun definitie en hun `run`-functie:

```
type CmdEval  a = ReaderT Device IO a
type ExpEval  a = ReaderT ExpEnv ( ExceptT Error IO ) a
type StatEval a = ReaderT StatEnv ( ExceptT Error ( StateT VarMap IO )) a

runCmdEval :: Device -> CmdEval a -> IO a
runCmdEval dev eval = runReaderT eval dev

runExpEval :: ExpEnv -> ExpEval a -> IO (Either Error a)
runExpEval env eval = runExceptT $ runReaderT eval env

runStatEval :: StatEnv -> VarMap -> StatEval a -> IO (Either Error a,
    VarMap)
runStatEval env vm eval = runStateT ( runExceptT $ runReaderT eval env )
    vm
```

`CommandEvaluator` (lijn 571): dit is de simpelste evaluator. Deze stuurt de MBot aan en daarvoor wordt de `Device`-handle van de MBot meegegeven in een `ReaderT`. Dit is een monad die een read-only variabele aanbied die kan opgevraagd worden met `ask`.

`ExpressionEvaluator` (lijn 499): deze evaluator zal expressies uitvoeren. De `ExpEnv` bestaat uit het tuple `(Device, VarMap)` en deze wordt ook weer meegegeven in een `ReaderT`. De evaluatie kan ook falen (bijvoorbeeld wanneer er zich een `TypeFout` voordoet) en daarom zit het resultaat in een `ExcepT`.

De `VarMap` is een `HashMap` die de namen van variabelen (strings) afbeeldt op hun waarde. De reden waarom deze read-only wordt meegegeven met de `ReaderT` is omdat expressies zelf de variabelen niet gaan aanpassen (en dit ook niet zouden mogen kunnen doen). Dit is de verantwoordelijkheid van de volgende evaluator.

`StatementEvaluator` (lijn 611): Deze evaluator is het meest ingewikkeld maar heeft ook als verantwoordelijkheid om de andere evaluators op te roepen waar nodig.

In de `ReaderT` wordt een tuple `(Device, MVar Bool)` meegegeven. De `Device`-handle is logisch. De `MVar Bool` wordt meegegeven om het uitvoeren vroegtijdig te stoppen indien nodig, maar info daarover volgt later.

In de `StateT` wordt de `VarMap` meegegeven, die in het begin van het programma leeg is. Bij iedere `Assign` statement wordt de state aangepast: er wordt een variabele toegevoegd of aangepast in die `VarMap`. De `Conditonal`, `Loop` en `Assign`

statements kunnen vervolgens deze map meegeven met de expressies die ze moeten evalueren.

De `ExcepT` zorgt er hier ook weer voor dat we foutmeldingen kunnen afhandelen. Foutmeldingen die voorkomen tijdens het evalueren van expressies worden automatisch doorgegeven aan deze monad (en de uitvoering wordt ook afgebroken). De `Right` van zal in dit geval meestal gewoon de unit `()` zijn, omdat de evaluatie van het programma geen resultaat teruggeeft.

5.3 GebruikersOnderbreking

Veel programma's gebruiken oneindige loops (`Gedurende waarachtig doe: ...`). Maar op een bepaald moment zouden we die toch graag afsluiten. Daarom heb ik ervoor gezorgd dat de signalen `SIGINT` en `SIGTERM` worden opgevangen en de executie wordt gestopt. Dit wordt gedaan door handlers te installeren (vanaf lijn 746) die een `MVar Bool` die initieel op `False` staat op `True` zetten wanneer zo'n signaal wordt opgevangen. De reden waarom dit een `MVar` is, is omdat deze handlers asynchroon worden verwerkt.

Bij het verwerken van een lijst statements (lijn 657) wordt er voor het evalueren van iedere statement gekeken of deze `MVar Bool` nog steeds `False` is. Indien dit niet het geval is wordt een error gegooid waardoor de uitvoering stopt.

Dit heeft als bijkomend voordeel dat de motoren van de MBot ook gestopt worden omdat vlak voor het sluiten van de verbinding met de MBot (lijn 758) er nog eens een `stop` commando wordt gestuurd naar de MBot. Moest dit signaal niet opgevangen worden, dan wordt het programma direct getermineerd en blijft de robot gewoon verder rijden in dezelfde richting.

5.4 Het typesysteem

Een `Value` (lijn 29) is een tuple `(Number, Type)` waar `Number` een double is. Zo kan er nog altijd aan typechecking gedaan worden, maar is alles intern nog altijd een nummer. Booleans worden dan voorgesteld door de waarden `(1.0, BoolT)` voor `waarachtig` en `(0.0, BoolT)` voor `strijdig`.

Expressies worden tijdens de uitvoering gecontroleerd of ze voldoen aan het verwachte type (bijvoorbeeld door `evalAndCheck` op lijn 535). Indien een expressie niet het verwachte type oplevert wordt er een `TypeFout` opgeworpen.

Deze oplossing is niet de meest elegante, maar na een eindje prutsen was dit het eerste die goed werkte en voor de gebruiker relatief onzichtbaar was.

5.5 Muziek

Om de buzzer van de MBot aan te sturen heb ik een patch opgestuurd naar de professor. Deze patch bestaat uit de toevoeging van de volgende functie:

```
playTone freq time = let (highFreq, lowFreq) = shortToBytes freq
                         (highTime, lowTime) = shortToBytes time
                        -- There is no port value. Instead, the least
                        -- signifcant byte of the frequency-value is
                        -- stored there.
                        -- Something something little-endian?
                        in MBotCommand idx RUN TONE
                            lowFreq [highFreq, lowTime, highTime]
                        where -- Split a short into a tuple of two bytes
                        shortToBytes i = (shiftR i 8, i)
```

Het heeft eventjes geduurd voor ik door had hoe de MBot API precies werkt en hoe de data precies doorgestuurd moet worden. Uiteindelijk komt het er op neer dat twee shorts (16 bits) moeten doorgestuurd worden: eerst de frequentie en vervolgens de duratie. Van deze shorst moet eerst de minst significante byte doorgesturd worden en vervolgens de meest significante byte: de MBot is waarschijnlijk little-endian.

Een minpuntje van deze functie is dat ze slechts afloopt wanneer de MBot zijn geluid volledig heeft afgespeeld.

6 Conclusie

De programmeertaal `BurgieScript` is een taal die kan gebruikt worden om de MBot aan te sturen. Er is een flexibele ondersteuning voor mathematische en boolese expressies en de flow van het programma kan gemakkelijk bepaald worden doormiddel van een while-lus en een if/else statement. Daarnaast worden (sommige) fouten meegedeeld aan de gebruiker, maar er is nog ruimte voor verbetering.

Wat er nog kan verbeterd of toegevoegd worden:

- Het typesysteem kon misschien wat properder door de types die intern worden gebruikt ook te laten overeenkomen met de types die ze moeten voorstellen.
- Een uitgebreidere manier om muziek af te spelen. Het originele plan was een commando toe te voegen als `speelMuziek` waar een bestand kon aan meegegeven worden waarin muziek stond (een mp3, of eventueel gewoon een csv bestand met op iedere lijn de frequentie gevolgd door de duratie van een noot) zodat tijdens het

uitvoeren van een programma de robot ook muziek zou afspelen. Helaas was er niet echt tijd meer om dit te implementeren. Het ging ook geen makkelijke opdracht zijn: de muziek zou parallel moeten afspelen met de executie van het programma, maar omdat zowel het programma als de achtergrondmuziek de MBot-connectie moeten delen zouden hier nog problemen mee kunnen ontstaan.

- Een interactieve mogelijkheid, zodat zoals met GHCi, commando's interactief kunnen uitgevoerd worden.
- De optie om een dummyrobot te gebruiken, zodat er geen connectie met een werkelijke robot nodig is.
- Duidelijkere foutmeldingen bij het parsen. Op dit moment zegt de parser vanaf welk punt er niet meer kon geparsed worden. Indien de foutieve statement in een blok zit van een andere statement (bijvoorbeeld van een while-lus) dan faalt eigenlijk ook de ouder-statement. Hierdoor lijkt het alsof de parser al faalt bij deze while-lus, terwijl de échte fout verder in het programma zit. Een oplosing voor dit probleem zou kunnen zijn door in de parsers zelf foutmeldingen te genereren.
- Toevoegen van tests. Ik was begonnen met unit-tests te schrijven met HUnit, maar heb die dan weer verwijderd om over te schakelen op QuickCheck. Hier was er helaas geen tijd meer voor.

Over het algemeen ben ik wel redelijk tevreden met mijn resultaat. De taal kan het meeste wat ik wou bereiken en ik heb het gevoel dat de kwaliteit van de code nog goed meevalt. Er kroop veel tijd in (of misschien heb ik er net iets teveel tijd in gestoken), maar het was een leuk project om te maken. Ik zal met pijn in het hart afscheid nemen van de MBot.

7 Appendix: broncode

Definitions.hs

```
module Definitions where

import Control.Monad.Except
import Control.Monad.Reader
import Control.Monad.State
import Control.Concurrent.MVar (MVar)

import System.HIDAPI (Device)
import qualified Data.HashMap.Strict as Map


type Number  = Double           -- Our number type
type Note    = (Int, Int)       -- (frequency, duration)
type Color   = (Int, Int, Int)  -- (Red, Green, Blue)
type VarName = String           -- Sadly there is no type which enforces

type Value = (Number, Type)     -- Everything is a number,
data Type = NumT | BoolT        -- but we still want to typecheck
            deriving (Show, Eq)

type Error = (String, ErrorType)
data ErrorType = TypeFout       -- Errors, in Dutch ofcourse!
                | OngebondenVeranderlijke
                | VerbodenVoorwaardeType
                | OntleedFout
                | GebruikersOnderbreking
                deriving Show

data Exp = Lit Value
           | Var VarName
           | UnExp UnFun Exp
           | BiExp BiFun Exp Exp
           | Query Sensor
           deriving Show

--                 Args Ret  Function itself
data UnFun = UnFun Type Type (Number -> Number)
data BiFun = BiFun Type Type (Number -> Number -> Number)

-- Because functions do not implement Show
instance Show UnFun where
    show (UnFun at rt _) = show at ++ "->" ++ show rt
instance Show BiFun where
    show (BiFun at rt _) = show at ++ "->" ++ show rt

data Sensor = Sonar | Line Side deriving Show

data Stat = Comment
            | Assign VarName Exp
            | MBot Command
            | Loop Exp [Stat]
            | Conditional Exp [Stat] [Stat]
            deriving Show

data Command = Motor Direction
             | Sound Note
             | Light Side Color
             | Sleep Time
             deriving Show
```

```haskell

data Time = Short | Normal | Long | AlmostEternally deriving Show

data Direction = Forward
               | Backward
               | Stop
               | Lat Side
               deriving Show

data Side = Portside | Starboard deriving Show


type VarMap = Map.HashMap VarName Value      -- Where variables are stored

type ExpEnv  = (Device, VarMap)              -- See: ExpressionEvaluator
type StatEnv = (Device, MVar Bool)           -- See: StatementEvaluator

-- << Insert Tranformers joke here >>
type CmdEval  a = ReaderT Device IO a
type ExpEval  a = ReaderT ExpEnv ( ExceptT Error IO ) a
type StatEval a = ReaderT StatEnv ( ExceptT Error ( StateT VarMap IO )) a
```

Parser.hs

```haskell
module Parser where

import Definitions

import Control.Monad
import Control.Applicative
import Data.Char

-- Definition here to avoid it becoming an orphan instance
newtype Parser a = Parser (String -> [(a,String)])

instance Monad Parser where
    p >>= k  = Parser $ \s ->
                [ (x2, s2) |
                  (x1, s1) <- apply p s,
                  (x2, s2) <- apply (k x1) s1]

instance Functor Parser where
    fmap = liftM

instance Applicative Parser where
    pure x = Parser $ \s -> [(x, s)]
    (<*>)  = ap

instance MonadPlus Parser where
    mzero = Parser $ const []
    mplus m1 m2 = Parser $ \s -> apply m1 s ++ apply m2 s

-- Enables some (one or more) and many (zero or more)
instance Alternative Parser where
    -- When there are multiple parsing options, select the first one
    pa <|> pb = Parser $ \s -> case apply (mplus pa pb) s of
                                 []    -> []
                                 (x:_) -> [x]
    empty = mzero

-- Parser application
apply :: Parser a -> String -> [(a, String)]
apply (Parser f) = f
```

```
120
121 -- Try to parse a string (trailing newlines are ignored) with a given Parser
122 parse :: Parser a -> String -> Either Error a
123 parse p str = let parsed        = apply (stript p) str
124                   complete      = filter (null . snd) parsed -- Completely
    parsed
125                   failWith msg  = Left (msg, OntleedFout)
126               in case complete of
127                 ((x,_):_)     -> Right x
128                 []  -> case parsed of
129                         ((_,s):_) -> failWith $ "Ontleding niet gelukt:\n"
130                                                 ++ show s
131                         []        -> failWith "Onbekende fout"
132
133 -- Choose from a list of parser by selecting the first successful one
134 chooseFrom :: [Parser a] -> Parser a
135 chooseFrom = foldl1 (<|>)
136
137 -- Take one character
138 char :: Parser Char
139 char = Parser f
140   where
141     f []     = []
142     f (c:cs) = [(c,cs)]
143
144 -- Take a character if it matches a predicate
145 satisfy :: (Char -> Bool) -> Parser Char
146 satisfy pr = do
147     c <- char
148     guard $ pr c
149     return c
150
151
152 -- Match a given character
153 takeChar :: Char -> Parser Char
154 takeChar = satisfy . (==)
155
156 -- Match a given string and remove trailing whitespace. Returns nothing.
157 match :: String -> Parser ()
158 match str = stripl >> mapM_ takeChar str
159
160 -- Throw away leading whitespace
161 stripl :: Parser ()
162 stripl = void $ many $ satisfy isSpace
163
164 -- Throw away trailing whitespace
165 stript :: Parser a -> Parser a
166 stript p = do
167     result <- p
168     stripl
169     return result
170
171
172 -- Easy conversion of exact strings to custom values
173 parseStrTo :: [(String, a)] -> Parser a
174 parseStrTo list = chooseFrom [ match str >> return a
175                              | (str, a) <- list ]
176
177 -- Parse something between a pair of strings
178 parseParens :: String -> Parser a -> String -> Parser a
179 parseParens open parser close = do
180     match open
181     p <- parser
```

```
    match close
    return p

-- Try to parse an operation between two expressions (ex. a + b),
-- if it fails we return the first expression ( a ).
chain :: Parser Exp -> Parser BiFun -> Parser Exp
chain exParser opParser = do
    a <- exParser
    rest a <|> return a
  where
    rest a = do
        bif <- opParser
        b  <- chain exParser opParser
        return $ BiExp bif a b
```

ExpressionParser.hs

```
module ExpressionParser (parseExp) where

import Parser
import Definitions
import CommandParser (parseSide)

import Control.Applicative ((<|>), some)
import Data.Char (isDigit, isAlpha)

-- Notes:
-- * No distinction is made between a numerical of a boolean expression
--   while parsing. I have chosen to throw an error while running the program
--   when an expression is used within the wrong type context.
-- * Every expression has four 'levels': an expression, a comparable expression
    ,
--   a term and a factor. On each level there are binary operators which
    produce
--   a result on a level higher. This makes it possible for multiplication to
--   have a higher precedence than addition for example.

parseLit, parseVar, parseQuery, parseTerm, parseFact, parseComp, parseExp ::
    Parser Exp

-- Numerical Binary Function
nbf :: (Number -> Number -> Number) -> BiFun
nbf = BiFun NumT NumT

-- Boolean Binary Function
bbf :: (Number -> Number -> Number) -> BiFun
bbf = BiFun BoolT BoolT

-- The or, and and not binary functions but with numbers
numOr, numAnd :: Number -> Number -> Number
numOr  1 _ = 1
numOr  _ x = x
numAnd 0 _ = 0
numAnd _ x = x

numNot :: Number -> Number
numNot 0 = 1
numNot _ = 0

-- Parse an expression
parseExp = chain parseComp $ parseStrTo [ ("overeenkomstig met"     , cbf (==))
                                        , ("verschillend met"       , cbf (/=))
                                        , ("significanter dan"      , cbf (>))
```

```
                                          , ("minder significant dan" , cbf (<))
                                          ]
  where  -- cbf: Comparative Binary Function
    cbf f = BiFun NumT BoolT $ toNum f
    toNum f a b = if f a b
                  then 1
                  else 0

-- Parse a comparable expression
parseComp = chain parseTerm $ parseStrTo
    [ ("+"      , nbf (+))
    , ("-"      , nbf (-))
    , ("hetzij" , bbf numOr)
    ]

-- Parse a term
parseTerm = chain parseFact $ parseStrTo
    [ ("x"      , nbf (*))
    , (":"      , nbf (/))
    , ("tevens" , bbf numAnd)
    ]

-- Parse a factor
parseFact = chooseFrom
    [ parseNegation base "-" numNegate
    , parseNegation base "allesbehalve" boolNegate
    , base
    ]
  where
    numNegate         = UnFun NumT NumT negate
    boolNegate        = UnFun BoolT BoolT numNot
    base = chooseFrom
        [ parseLit
        , parseParens "(" parseExp")"
        , parseQuery
        , parseVar
        ]


-- Literal number or boolean
-- A number can have a fractional part after a comma
parseLit = parseNumLit <|> parseStrTo
    [ ("waarachtig", Lit (1, BoolT))
    , ("strijdig",   Lit (0, BoolT))
    ]
  where
    takeDigits = some $ satisfy isDigit
    parseNumLit = do
        stripl
        s1 <- takeDigits
        s2 <- afterComma <|> return []
        return $ Lit (read (s1 ++ s2), NumT)
    afterComma = do
        match ","
        s <- takeDigits
        return ('.':s)

-- Parse a variable name
parseVar = do
    stripl
    name <- some $ satisfy isAlpha
    return $ Var name

```

```
-- Query: request a measurement from one of the MBot's sensors
parseQuery = sonar <|> line
  where
    sonar = do
        match "GeluidWeerkaatsingsApparaatWaarde"
        return $ Query Sonar
    line = do
        match "LijnVolgApparaatIsWit"
        side <- parseSide
        return $ Query $ Line side

-- Negation: an expression preceded by an unary operator
parseNegation :: Parser Exp -> String -> UnFun -> Parser Exp
parseNegation without token unfun = do
    match token
    e <- without
    return $ UnExp unfun e
```

CommandParser.hs

```
module CommandParser (parseMBot, parseSide) where

import Parser
import Definitions

import Control.Applicative

-- Note:
-- This module really does not need much explenation. Each command
-- is parsed by looking for an exact string.

parseMotorCommand, parseSoundCommand, parseLightCommand, parseSleep :: Parser
    Command

-- Combine all the individual command parsers
parseMBot :: Parser Stat
parseMBot = MBot <$> chooseFrom
    [ parseMotorCommand
    , parseSoundCommand
    , parseLightCommand
    , parseSleep
    ]

parseSide :: Parser Side
parseSide = do
    match "te"
    parseStrTo
        [ ("bakboordzijde"   , Portside)
        , ("stuurboordzijde" , Starboard)
        ]

parseDirection :: Parser Direction
parseDirection = chooseFrom [pLineal, pLateral]
  where
    pLateral  = Lat <$> parseSide
    pLineal = parseStrTo
        [ ("voorwaarts", Forward)
        , ("rugwaarts",  Backward)
        ]


parseMotorCommand = chauffeer <|> stop
  where
```

```
    chauffeer = do
        match "Chauffeer"
        dir <- parseDirection
        match "!"
        return $ Motor dir
    stop = do
        match "Halt!"
        return $ Motor Stop


parseSoundCommand = do
    match "Zing"
    duration <- parseStrTo
        [ ("kortstondig", 125)
        , ("eventjes",    250)
        , ("langdurig",   500)
        , ("erg langdurig", 2000)
        ]
    match "een"
    frequency <- parseStrTo
        [ ("do",  523)
        , ("re",  587)
        , ("mi",  659)
        , ("fa",  698)
        , ("sol", 784)
        , ("la",  880)
        , ("si",  988)
        ]
    match "!"
    return $ Sound (frequency, duration)


parseLightCommand = parseLightsOff <|> parseLightsOn
  where
    parseLightsOff = do
        match "Verduister"
        side <- parseSide
        match "!"
        return $ Light side (0,0,0)
    parseLightsOn = do
        match "Kleur"
        color <- parseStrTo
            [ ("rood"   , (100  , 0    , 0    ))
            , ("groen"  , (0    , 100  , 0    ))
            , ("blauw"  , (0    , 0    , 100  ))
            , ("wit"    , (100  , 100  , 100  ))
            ]
        side <- parseSide
        match "!"
        return $ Light side color

parseSleep = do
    time <- parseStrTo
        [ ("Rust kort!"                    , Short)
        , ("Neem op je gemak een pauze!"   , Normal)
        , ("Sluit je ogen maar voor even!" , Long)
        , ("Droom zacht zoete prins!"      , AlmostEternally)
        ]
    return $ Sleep time
```

StatementParser.hs

```
module StatementParser (parseStatements, parseStat) where
```

```
423
424 import Definitions
425 import Parser
426 import CommandParser
427 import ExpressionParser
428
429 import Data.Char
430 import Control.Monad (void)
431 import Control.Applicative ((<|>), many)
432
433 -- Parse a list of statementsby using recursion.
434 -- If the list is not parseable, an empty list is returned.
435 parseStatements :: Parser [Stat]
436 parseStatements = try <|> return []
437     where try = do
438             s  <- parseStat
439             xs <- parseStatements  -- recurse!
440             return (s:xs)
441
442 -- Parses a block: a list of statements between \ and /
443 parseBlock :: Parser [Stat]
444 parseBlock = parseParens "\\" parseStatements "/"
445
446 parseStat :: Parser Stat
447 parseStat = chooseFrom [parseComment,
448                         parseAssign,
449                         parseLoop,
450                         parseConditional,
451                         parseMBot]
452
453 parseComment, parseAssign, parseLoop, parseConditional :: Parser Stat
454
455 -- Parse a comment.
456 -- Comments begin with "Terzijde:" and end with a punctuation mark.
457 parseComment = do
458     match "Terzijde:"
459     void $ many $ satisfy $ not . punc
460     void $ satisfy punc
461     return Comment
462   where punc c = c `elem` ['?','!','.']
463
464 parseAssign = do
465     match "Zet"
466     match "variabele"
467     stripl            -- Allow spaces before a variable name.
468     varName <- many $ satisfy isAlpha
469     match "op"
470     expr <- parseExp  -- To avoid shadowing the 'exp' function, I have to
471     match "."         -- be inconsistent and use 'expr' as variable.
472     return $ Assign varName expr
473
474 -- Helper method wich parses the condition for a loop or condition.
475 parseConditionFor :: String -> Parser Exp
476 parseConditionFor keyword = do
477     match keyword
478     expr <- parseExp
479     match "doe"
480     match ":"
481     return expr
482
483 parseLoop = do
484     expr <- parseConditionFor "Gedurende"
485     blk <- parseBlock
```

```
    return $ Loop expr blk

parseConditional = do
    expr <- parseConditionFor "Indien"
    ifBlk <- parseBlock
    elseBlk <- parseElse <|> return []
    return $ Conditional expr ifBlk elseBlk
  where
    parseElse = do
        match "Anderzijds"
        match ":"
        parseBlock
```

ExpressionEvaluator.hs

```
module ExpressionEvaluator (runExpEval, evalExp) where

import Definitions

import MBot as M
import qualified Data.HashMap.Strict as Map
import Control.Monad.Identity
import Control.Monad.Except
import Control.Monad.Reader
import GHC.Float

-- Lift IO to ExpEval
lift2 :: IO a -> ExpEval a
lift2 = lift . lift

-- The ExpEnv is a tuple (Device, VarMap) which is put in a ReaderT monad
-- because it would not be sane to allow an expression to modify variables.
-- Setting a variable is only allowed with the Assign-statement.
runExpEval :: ExpEnv -> ExpEval a -> IO (Either Error a)
runExpEval env eval = runExceptT $ runReaderT eval env

-- Helper: evaluate expression and check if it matches the expected type
-- if the types do not match, it throws a TypeError
evalAndCheck :: Exp -> Type -> ExpEval Number
evalAndCheck expr exType = do
    (v, t) <- evalExp expr
    when (t /= exType) $
        let msg = "Verwacht type is "   ++ show exType ++
                  " maar was "          ++ show t ++
                  " in de expressie "   ++ show expr
         in throwError (msg, TypeFout)
    return v

-- The main workhorse of this module: evaluate an Exp
evalExp :: Exp -> ExpEval Value
evalExp (Lit n)     = return n                  -- Literal value
evalExp (Var name)  = do                        -- Variable lookup
    (_, vm) <- ask
    let v  = Map.lookup name vm
     in case v of
        Nothing  -> let msg = "Veranderlijke " ++ name ++
                              "werd (nog) niet ëgedefinierd."
                     in throwError (msg, OngebondenVeranderlijke)
        Just val -> return val
evalExp (UnExp (UnFun at rt f) expr) = do       -- Unary operator
    v <- evalAndCheck expr at
    return (f v, rt)
evalExp (BiExp (BiFun argt rett f) a b) = do    -- Binary operator
```

```
    av <- evalAndCheck a argt
    bv <- evalAndCheck b argt
    return (f av bv, rett)
evalExp (Query Sonar) = do                          -- Sonar Query
    (d,_) <- ask
    f <- lift2 $ M.readUltraSonic d
    return (float2Double f, NumT)
evalExp (Query (Line side)) = do                    -- Line Query
    (d,_) <- ask
    l <- lift2 $ M.readLineFollower d
    case side of
        Portside  -> return $ isWhiteLeft l
        Starboard -> return $ isWhiteRight l
  where
    yes   = (1, BoolT)
    no    = (0, BoolT)
    isWhiteLeft l = case l of
                        M.BOTHW  -> yes
                        M.RIGHTB -> yes
                        _        -> no
    isWhiteRight l = case l of
                        M.BOTHW  -> yes
                        M.LEFTB  -> yes
                        _        -> no
```

CommandEvaluator.hs

```
module CommandEvaluator where

import MBot
import Definitions

import System.HIDAPI (Device)
import Control.Monad.Reader
import Control.Concurrent

-- Run the command evaluator.
runCmdEval :: Device -> CmdEval a -> IO a
runCmdEval dev eval = runReaderT eval dev

-- Evaluate a command
evalCommand :: Definitions.Command -> CmdEval ()
evalCommand (Motor dir) = do                                -- Motor
    d <- ask
    lift $ case dir of
        Stop            -> stop d
        Forward         -> goAhead d
        Backward        -> goBackwards d
        Lat Portside    -> goLeft d
        Lat Starboard   -> goRight d
evalCommand (Sound note) = do                               -- Sound
    d <- ask
    lift $ sendCommand d $ uncurry playTone note
evalCommand (Light side (r,g,b)) = do                       -- Light
    d <- ask
    let i = case side of
                Portside    -> 1
                Starboard   -> 2
     in lift $ sendCommand d $ setRGB i r g b
evalCommand (Sleep time) = lift $ threadDelay delay         -- Sleep
  where
    delay = case time of
                Short           -> 250000
```

```
                Normal          -> 1000000
                Long            -> 2000000
                AlmostEternally -> 10000000
```

StatementEvaluator.hs

```
module StatementEvaluator (evalStatements, runStatEval, evalStat) where

import Definitions
import ExpressionEvaluator
import CommandEvaluator

import Control.Monad.Except
import Control.Monad.State
import Control.Monad.Reader
import Control.Concurrent.MVar (readMVar)
import qualified Data.HashMap.Strict as Map

-- Run the statements.
-- * A StatEnv with the device handle and an MVar containing a boolean
--   which indicates if the program needs to be interrupted is put in a
--   Reader monad, because these are read-only.
-- * The HashMap containing all the variables is put in the State monad,
--   because this can change with every Assign statement.
-- * The state and result (mostly the Unit) is wrapped in the Either monad
--   because Error handling.
runStatEval :: StatEnv -> VarMap -> StatEval a -> IO (Either Error a, VarMap)
runStatEval env vm eval = runStateT ( runExceptT $ runReaderT eval env ) vm

-- Powerlift
lift3 :: IO a -> StatEval a
lift3 = lift . lift . lift

-- Because typing 'return ()' is 6 charachters too much.
nop :: StatEval ()
nop = return ()

-- Calculate the value of an expression, without worrying about
-- environments, lifting or errors.
calculate :: Exp -> StatEval Value
calculate expr = do
    vm  <- get
    (d,_) <- ask
    result <- lift3 $ runExpEval (d,vm) $ evalExp expr
    either throwError return result

-- Execute a list (block) of statements.
-- Because after each statement evaluation this function
-- is executed, there is a check here to see wheter the
-- program's termination is requested.
evalStatements :: [Stat] -> StatEval ()
evalStatements [] = nop
evalStatements (st:sts) = do
    (_,mv) <- ask       -- Termination check
    isInterrupt <- lift3 $ readMVar mv
    when isInterrupt $ throwError ("Evaluatie gestopt", GebruikersOnderbreking)
    evalStat st         -- Handle the next statement and loop
    evalStatements sts

-- Decide, based on the given expression, whether to execute the first or the
-- second monadic action.
-- When the type of the expression is not a BoolT, an error is thrown.
decide :: Exp -> StatEval () -> StatEval () -> StatEval ()
```

```
decide expr yes no = do
    (v, t) <- calculate expr
    case t of
        BoolT -> case v of
                    1 -> yes
                    0 -> no
                    _ -> error "BoolT with wrong value"
        _     -> throwError ("Type van de voorwaarde was: " ++ show t,
                             VerbodenVoorwaardeType)

-- Evaluate one statement
evalStat :: Stat -> StatEval ()
evalStat Comment = nop                                  -- Comment
evalStat (Assign name expr) = do                         -- Assignment
    val <- calculate expr
    modify $ Map.insert name val
evalStat (MBot cmd) = do                                -- MBot command
    (dev,_) <- ask
    lift3 $ runCmdEval dev $ evalCommand cmd
-- Loop
evalStat l@(Loop expr blk) = decide expr (evalStatements blk >> evalStat l) nop
-- Conditional
evalStat (Conditional expr ifBlk elseBlk) = decide expr (evalStatements ifBlk)
                                                        (evalStatements elseBlk
    )
```

Main.hs

```
{-# OPTIONS_GHC -fno-warn-unused-binds #-}
-- ^ execExp wordt nergens gebruikt in  de code en zal een warning geven
-- Het is echter een handige functie om te debuggen. Omdat ik -Werror en -Wall
-- gebruik om te compileren is bovenstaande optie dus nodig.
import MBot
import Definitions
import Parser
import ExpressionParser
import ExpressionEvaluator
import StatementParser
import StatementEvaluator

import Control.Monad (void, when)
import Control.Concurrent.MVar (newMVar, swapMVar)
import System.Environment (getArgs)
import System.Exit (die)
import System.Posix.Signals (installHandler, Handler(Catch), sigINT, sigTERM)
import qualified Data.HashMap.Strict as Map

main :: IO ()
main =  do
    args <- getArgs
    when (length args /= 1)
        $ die Éé"n (1) argument nodig: het pad naar het uit te voeren bestand."
    executeFile $ head args

-- Parse and evaluate file contents.
executeFile :: FilePath -> IO ()
executeFile file = do
    program <- parseFile file
    -- If parsing failed: pass the error,
    -- If parsing succeeded: execute the statements
    result  <- either (return . Left) executeStatements program
    either failure success result
    putStrLn "Vaarwel."
```

```
731     where failure (msg, err) = do
732             putStrLn "!! De volgende fout werd opgegooid:"
733             putStrLn $ show err ++ ": " ++ msg
734           success _ = putStrLn "Het programma werd zonder fouten uitgevoerd."
735
736
737 -- Parse a file.
738 parseFile :: FilePath -> IO (Either Error [Stat])
739 parseFile file = do
740     cont <- readFile file
741     return $ parse parseStatements cont
742
743 -- Evaluate a list of statements
744 executeStatements :: [Stat] -> IO (Either Error ())
745 executeStatements program = do
746     -- Install handler to catch Control-C
747     intMVar <- newMVar False
748     let handleInt = void $ swapMVar intMVar True
749      in do
750         void $ installHandler sigINT  (Catch handleInt) Nothing
751         void $ installHandler sigTERM (Catch handleInt) Nothing
752
753     d <- openMBot
754     putStrLn "Opgepast, we gaan starten!"
755     -- Run the parsed statements
756     (result, vm) <- runStatEval (d, intMVar) Map.empty $ evalStatements program
757     -- Stop the MBot before closing it to avoid accidents
758     stop d
759     closeMBot d
760     putStrLn "Programma afgelopen."
761     print vm
762     return result
763
764 -- Execute a string with an expression
765 execExp :: String -> IO (Either Error Value)
766 execExp str = either (return . Left) exec $ parse parseExp str
767   where
768     exec e = runExpEval undefined $ evalExp e
```