

Project Functioneel Programmeren

Simon Schellaert
Billie Devolder

1. Introductie

In het kader van het project voor het vak Functioneel Programmeren ontwikkelden we de taal 🤖. Deze taal kan gebruikt worden om een fysieke mBot of een virtuele mBot in de simulator, die we eveneens ontwikkelden, aan te sturen. Geïnspireerd door de overweldigende populariteit van Emoji, besloten we in de syntax veelvuldig gebruik te maken van Emoji. Dit bood niet alleen technisch een interessante uitdaging, maar ook een aangename afwisseling voor de zoveelste *if..then..else*.

Een belangrijk principe bij het ontwerp van onze taal was dat 🤖 - programma's er elegant en 'aaibaar' moesten uitzien. Hiermee rekening houdend, besloten we om onze taal niet louter uit Emoji op te bouwen, maar enkel Emoji te gebruiken op plaatsen waar onze taal er effectief leesbaarder door werd. Zo kozen we bijvoorbeeld voor 📏 in plaats van een functie `getDistance()`, maar kozen we niet voor ➡ in de plaats van / aangezien deze laatste substitutie de leesbaarheid niet ten goede komt. Met dit principe in het achterhoofd, probeerden we de 'clutter' in onze taal zoveel mogelijk te beperken. Zo maken we, net zoals bv. Python, gebruik van indentatie in plaats van paren haakjes om blokken aan te duiden en worden statements gescheiden door een nieuwe lijn in plaats van een puntkomma. Aan de andere kant namen we het dubbelpunt na de conditie van een if-statement dan weer niet over uit Python, aangezien dit onnodige visuele ruis introduceerde.

Uiteraard zijn we ons ervan bewust dat we niet de eerste zijn met het idee om een taal te maken gebaseerd op Emoji. Zo gebruikt de taal 4Lang louter Emoji, maar ook Apple's Swift laat Emoji-karakters toe in namen van variabelen of klassen. We besloten om ons zo weinig mogelijk te baseren op bestaande Emoji-talen omdat het ons leuker en leerrijker leek om iets volledig vanaf nul te ontwerpen.

Los van het esthetische aspect, valt uiteraard te discussiëren over het praktisch nut van de taal. Zo bevatten de meeste toetsenborden geen Emoji-karakters. We zijn er dan ook geenszins van overtuigd dat een programmeertaal gebruik makend van Emoji effectief een goed idee is voor praktische toepassingen. Waar wel van overtuigd zijn, is dat het ontwikkelen en implementeren van deze taal een erg interessante en aangename leerervaring was.

2. Syntax van de taal

We maken gebruik van de *Extended Backus-Naur form* om de syntax van onze taal te beschrijven. Om dit overzichtelijk te houden, voegen we wat extra opmaak toe om duidelijk het onderscheid te maken tussen elementen in onze taal en elementen van EBNF. Zo worden niet-terminalen **omkaderd** en krijgen symbolen die witruimte aanduiden een **blauwe achtergrond**. Syntaxelementen van EBNF worden vervolgens aangeduid met een **lichte kleur**. Alle terminalen die de gebruiker effectief intypt (bv. haakjes of 🡕), zijn tenslotte een emoji of gewoon zwart.

```

StmtSeq      := Stmt { 🡕 Stmt }
Stmt         := Assignment | While | If | Command | Skip

Assignment  := Identifier 🡕 AExp
While       := 🡕 BExp 🡕 StmtSeq 🡕
If          := ? BExp 🡕 StmtSeq 🡕 { 🡕 !? BExp 🡕 StmtSeq 🡕 }
              [ 🡕 ! 🡕 StmtSeq 🡕 ]
Skip        := 🗑 Some text

AExp        := ATerm { ( + | - ) ATerm }
ATerm       := AFactor { ( * | / ) AFactor }
AFactor     := ( AExp ) | Constant | Sensor | Identifier
Constant    := Literal | 🟦 | 🟡🟦 | 🟦🟡 | 🟡
Sensor      := 📡 | 📶

```

BExp	:=	BTerm	{		BTerm	}
BTerm	:=	BFactor	{	&&	BFactor	}
BFactor	:=	👍		👎		! BFactor (BExp) BRel
BRel	:=	AExp	<	AExp		AExp == AExp AExp > AExp
Command	:=	👤	Direction		🕒	Duration 🚩 Flank AExp AExp AExp
Direction	:=	⬇️		➡️		⬅️ ⬆️
Duration	:=	🕒		🕒		🕒 🕒 🕒 AExp
Flank	:=	👉		👈		

In bovenstaande definities duidt *Identifier* de naam van een variabele aan. Variabelenamen beginnen steeds met een letter gevolgd door nul of meerdere alfanumerieke karakters. Een *Literal* is een geheel getal in het decimale talstelsel. *Some text* duidt vervolgens één regel tekst met eender welke karakters, waaronder dus ook witruimte, aan.

3. Semantiek van de taal

Een programma in 🤖 is een StmtSeq, m.a.w. een opeenvolging van één of meerdere statements die elk op een eigen lijn staan. We maken dus geen gebruik van puntkomma's om statements te scheiden, zoals dat vaak gebeurt, maar wel van één of meerdere newline-karakters. Ook worden blokken niet aangeduid door ze te omringen met overeenkomstige haakjes, maar wel door de statements die deel uitmaken van het blok meer te laten inspringen. Verder zijn de statements die we onderscheiden erg gelijkaardig aan diegene die je vindt in traditionele imperatieve programmeertalen. De vijf types statements worden hieronder individueel toegelicht.

Assignment

Dit is een toekenning. De waarde van de aritmetische expressie rechts van het toekenningsteken (👈) wordt opgeslagen onder de naam *Identifier*, die links die van het toekenningsteken staat.

While

Dit is een lus. De body wordt uitgevoerd zolang de booleaanse expressie die de conditie voorstelt naar 👍 evalueert.

If

Dit is een conditional. De structuur is dezelfde als bij een traditioneel if/else-if/else-statement. Eerst wordt de booleaanse expressie naast ? geëvalueerd. Als deze naar 👍 evalueert, wordt de bijhorende body uitgevoerd. Indien de conditie naar 👎 evalueert, testen we de !? -condities één voor één en voeren we de body uit die hoort bij de eerste conditie die naar 👍 evalueerde. Indien geen enkele conditie naar 👍 evalueerde, wordt de body van ! -tak uitgevoerd indien deze gegeven is. Er mogen dus een nul of meerdere !? -takken zijn en hoogstens één ! -tak.

Skip

Dit is commentaar. Alle tekst rechts van het ☹-symbool op dezelfde lijn wordt genegeerd.

Command

Dit stelt een commando voor dat naar de mBot of de simulator wordt gestuurd. We onderscheiden vier types commando's. De argumenten van een commando worden van elkaar gescheiden door een spatie.

Drive

Stel de motor in om in de opgegeven richting te rijden. Mogelijke richtingen zijn vooruit (⬆), achteruit, (⬇), naar links (⬅) en naar rechts (➡).

Sleep

Slaap even alvorens verder te gaan met de uitvoering. Gedurende het slapen blijft de MBot actief. Hij rijdt dus gewoon verder in de richting waarin hij reeds aan het rijden was. Als argument kan een aritmetische expressie, die het aantal te slapen milliseconden voorstelt, meegegeven worden. Een andere mogelijkheid is om gebruik te maken van één van de ingebouwde constanten om 400 ms (⌚), 800 ms (⌚), 1200 ms (⌚), 1600 ms (⌚) of 2 s (⌚) te slapen.





Light







Laat het opgegeven lichtje branden in de opgegeven kleur. Het eerste argument duidt aan als het commando betrekking heeft op het linkse (👉) of rechtse (👈) lichtje. De volgende drie argumenten zijn aritmetische expressies die de RGB-waarde van het in te stellen kleur voorstellen. Het bereik van de RGB-waarden loopt van 0 t.e.m. 100 en waarden buiten dit interval worden geclipped.

In de bovenstaande bespreking van statements, werden de termen *booleanse expressie* en *aritmetische expressie* veelvuldig gebruikt. We leggen nu ook de semantiek van deze taalelementen precies vast.

Aritmetische expressies

Een aritmetische expressie is een uitdrukking die als resultaat een geheel getal oplevert. Om de prioriteit van de bewerkingen eenduidig vast te leggen, gebruiken we een hiërarchie van taalelementen. Een expressie bestaat uit één of meerdere termen gescheiden door een plus- of min-symbool. Deze twee operatoren zijn links-associatief en hebben de laagste prioriteit. Een term bestaat vervolgens uit één of meerdere factoren gescheiden door een maal- of deling-symbool, welke ook beide links-associatief zijn maar een hogere prioriteit hebben. Een factor tenslotte is het meest elementaire deel en kan bestaan uit een expressie omringd door haakjes, een constante numerieke waarde, een sensor of een identifier.

Een constante is ofwel de decimale representatie van een geheel getal of één van de ingebouwde constanten - , ,  en  - die respectievelijk 0, 1, 2 en 3 voorstellen. Denk aan de binaire voorstelling van deze getallen om in te zien waarom deze representatie steek houdt.

Een sensor verwijst naar de sensorwaarde van een van de ingebouwde sensoren. De afstandssensor, , stelt de afstand voor tot het object voor de mBot. De waarde van deze sensor is steeds een geheel getal. De lijnsensor, , geeft aan welke kleur de mBot onder zich ziet. De waarde van deze sensor is steeds een waarde van 0 t.e.m. 3:  (beide zwart),  (enkel links zwart),  (enkel rechts zwart) of  (beide wit).

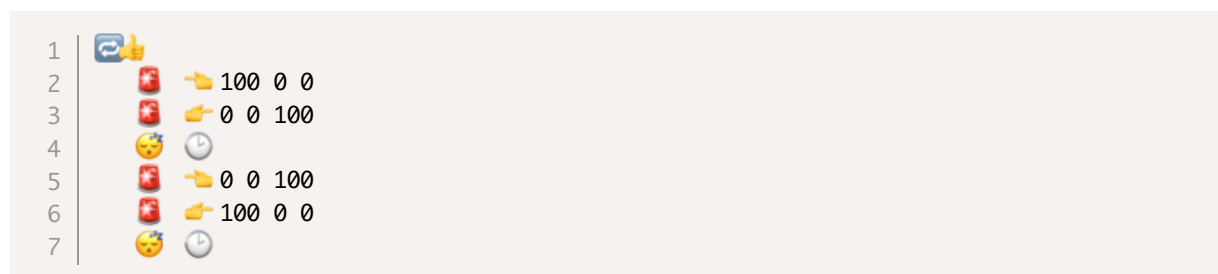
Een identifier verwijst vervolgens naar de waarde van een variabele. Indien er nog niet eerder geschreven werd naar de corresponderende variabele, wordt er een fout opgegooid tijdens de uitvoering.

Booleaanse expressies

Een booleaanse expressie is een uitdrukking die als resultaat de waarde waar (👍) of onwaar (👎) oplevert. Om de prioriteit van de bewerkingen ook hier eenduidig vast te leggen, gebruiken we opnieuw een hiërarchie van taalelementen. Een expressie bestaat uit één of meerdere termen gescheiden door het OR-symbool (\vee). Een term bestaat uit zijn beurt weer uit één of meerdere factoren gescheiden door het AND-symbool (\wedge). Een factor tenslotte is waar (👍), onwaar (👎), het omgekeerde van factor, een expressie tussen haakjes of het resultaat van een vergelijking. De mogelijke vergelijkingen zijn kleiner dan ($<$), gelijk aan ($=$) en groter dan ($>$).

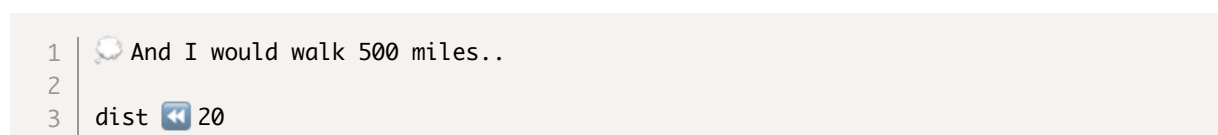
4. Voorbeeldprogramma's

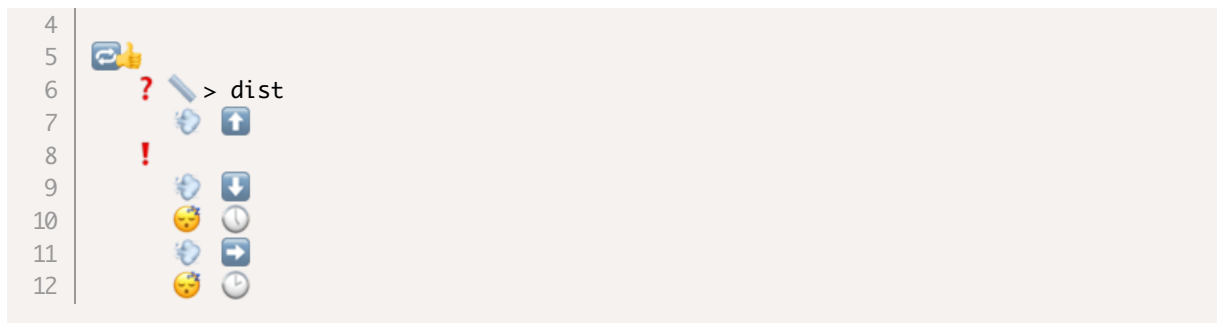
Politiewagen



Dit programmaatje bestaat uit één oneindige lus. In deze lus wordt eerst het linkerledje op rood (RGB-waarde 100 0 0) en het rechterledje op blauw (RGB-waarde 0 0 100) gezet. Vervolgens wachten we 400 ms en doen we het omgekeerde, m.a.w. het linkerledje op blauw zetten en het rechterledje op rood. Tenslotte wachten we opnieuw 400 ms en wordt de lus opnieuw uitgevoerd.

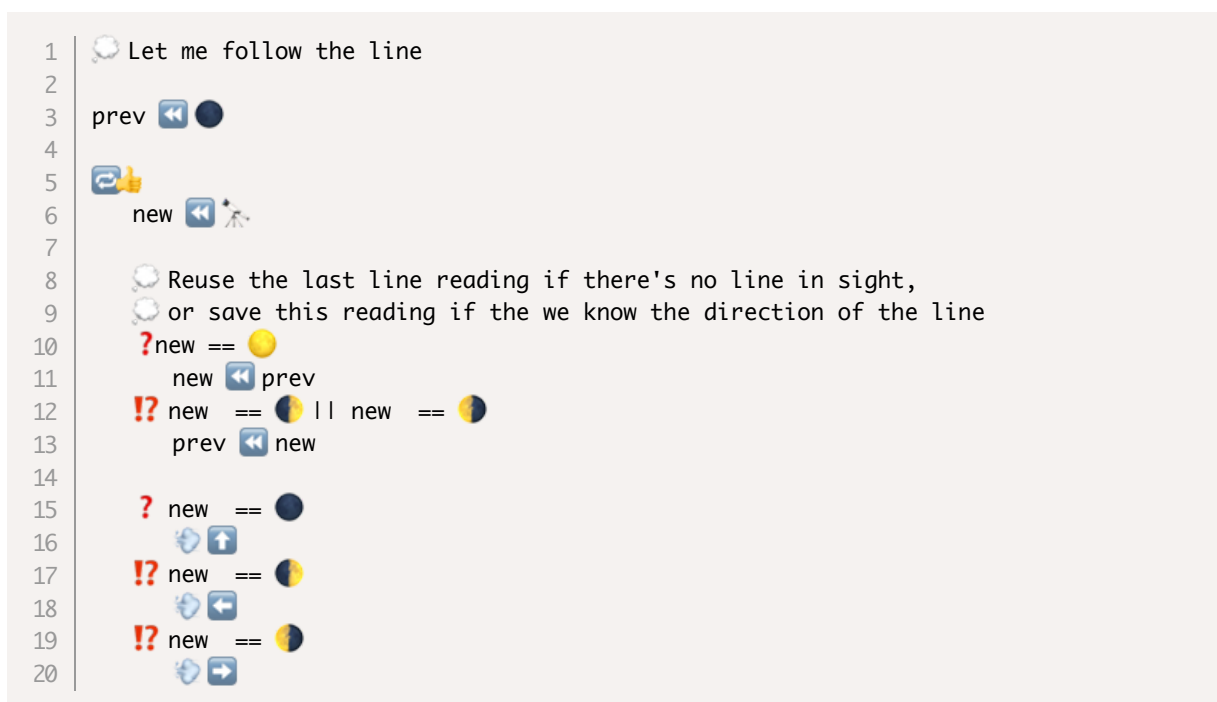
Obstakels ontwijken





Het merendeel van de functionaliteit situeert zich wederom in een oneindige lus. In deze lus wordt de afstandssensor (📏) uitgelezen en vergeleken met de voorafbepaalde threshold *dist*. Indien de afstand tot een object groter is dan deze threshold, blijft de robot rechtdoor rijden. Indien niet, dan rijden we gedurende 800 ms achteruit en draaien we vervolgens gedurende 400 ms naar rechts vooraleer de lus opnieuw wordt uitgevoerd.

Lijn volgen



Net zoals bij de vorige twee programma's, wordt ook hier het merendeel van het werk uitgevoerd in de oneindige lus. Het programma is ietsje ingewikkelder doordat we merkten dat de mBot soms kortstondig de lijn kwijtraakt. Om te vermijden dat het lijn-volgen vervolgens volledig de mist in gaat, houden we telkens de meest recente bruikbare waarde van de lijnsensor bij in de variabele *prev*. Het eerste wat we doen in de lus is het uitlezen van de laatste waarde van de lijnsensor. Vervolgens controleren we als we iets met

deze waarde kunnen aanvangen. Indien de waarde namelijk tweemaal wit is, hergebruiken we de laatste geldige waarde die opgeslagen zit in *prev*. Indien de waarde ofwel links ofwel rechts zwart is, slaan we deze waarde op. Vervolgens sturen we de motoren aan op basis van deze waarde. Concreet betekent rijden we naar links (resp. rechts) als we links (resp. rechts) zwart zien of rijden we rechtdoor indien we tweemaal zwart uitlezen.

5. Implementatie

We overlopen nu kort de belangrijkste punten van onze implementatie.

Preprocessor

De eerste stap na het inlezen van het invoerprogramma is het verwerken door de preprocessor. Doordat onze taal, in plaats van paren van overeenkomstige haakjes, gebruik maakt van indentatie (de zgn. off-side rule) om blokken aan te duiden, is de taal namelijk niet langer contextvrij. Het parsen van een niet-contextvrije taal kan echter al snel relatief ingewikkeld worden. Gelukkig bestaat er een elegant trucje, dat onder andere door parsers voor Python wordt gebruikt, om alsnog een contextvrije taal te bekomen. Op die manier kunnen we vervolgens opnieuw gebruik maken van de klassieke Parser-monad. Het trucje bestaat eruit om onze taal eerst te laten verwerken door een preprocessor die telkens een **INDENT** token toevoegt als de indentatie vermeerderd en een **DEDENT** token als deze vermindert. Deze preprocessor houdt dan toestand bij, zijnde een stack van voorgaande indentatieniveaus, maar we bekomen vervolgens een nieuwe taal die wel contextvrij is. Deze nieuwe taal, met als enige verschil dat alle indentatie vervangen is door **INDENT/DEDENT** tokens, wordt vervolgens verwerkt door de parser, die geen rekening meer hoeft te houden met indentatie. Concreet gebeurt deze transformatie in de functie `preprocess` ^[Parser.hs: 271].

Parser

Na het preprocessen, wordt het programma vervolgens geparset met behulp van de Parser-monad. De meeste functies die we gedefinieerd hebben, komen rechtstreeks overeen met syntaxelementen uit de Backus-Naur form. Zo is er

een parser voor aritmetische expressies ^[Parser.hs: 161] en booleaanse expressies ^[Parser.hs: 180]. Het programma zelf is dan weer een sequentie van statements ^[Parser.hs: 196]. Om te vermijden dat de Emojis verspreid staan doorheen onze code, aggregeren we alle constante symbolen onderaan het bestand ^[Parser.hs: 292].

Evaluator

De uitvoer van de parse-fase is uiteindelijk één statement ^[Evaluator.hs: 34] die de AST van het programma voorstelt. Door de functie `runStmt` ^[Evaluator.hs: 79] op te roepen met dit statement, wordt het programma vervolgens uitgevoerd. Het eerste argument van deze functie is een zogenaamd **Device** ^[Evaluator.hs: 53]. Dit is een datastructuur die de functies zoals `setMotor` etc. bevat die opgeroepen worden door de evaluator. Op die manier kan eenvoudig gekozen worden als het programma wordt uitgevoerd op de fysieke mBot ^[Interpreter.hs: 14] of op de simulator ^[Simulator.hs: 12].

Fysieke mBot

Om het programma uit te voeren op de fysieke mBot, moet er eerst een geschikt device worden aangemaakt dat communiceert met de mBot ^[Interpreter.hs: 8]. Vervolgens kan het meegegeven programma ingelezen, geparset en geëvalueerd worden met het zonet aangemaakte device. Merk op dat deze laatste stappen ook van toepassing zijn bij uitvoering op de simulator. Om code duplicatie te vermijden, hebben we deze functionaliteit dan ook ondergebracht in een functie `initialize` ^[Initialize.hs: 12]. Die functie kan vervolgens opgeroepen worden met een geschikt device om het inlezen, parsen en uitvoeren van het programma te starten.

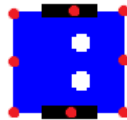
Simulator

De communicatie tussen het programma en de simulator gebeurt met behulp van een **MVar** ^[SimulatorInterface.hs: 18]. De **MVar** wrappt een **World**. Het **World** datatype bevat alle informatie over de wereld en de robot. Beide threads werken met de **MVar** op dezelfde manier: eerst wordt de wereld uit de **MVar** gehaald, daarna

wordt een aangepaste wereld berekend en tenslotte wordt de aangepaste wereld in de MVar gestopt.

Collision detection

Collision detection is geïmplementeerd door te kijken of er ten minste één van de acht punten aangeduid op de onderstaande figuur zich in een muur bevindt. Opmerkelijk hierbij is dat we niet enkel de hoekpunten van de robot controleren.



De nood aan de 4 extra punten wordt geïllustreerd aan de hand van de volgende twee scenario's: In het eerste scenario wil de robot zich door een kleine ingang tussen twee muren bewegen. Een muur en de robot zijn precies even groot dus zullen de hoekpunten van de robot precies samenvallen met de rand van de muur. We zijn dus genoodzaakt om punten die precies op de rand vallen niet te zien als collisions.



In het tweede scenario rijdt de robot tegen een muur die zich op dezelfde y-coördinaat bevindt als de robot. Indien we enkel de hoekpunten zouden bekijken, zouden alle punten precies samenvallen met een rand van een muur. Doordat punten net op de rand van een muur mogen vallen, zou er dus geen collision zijn. De robot zou dus in staat zijn om door de muur te rijden. We zijn dus genoodzaakt om 4 extra punten te controleren.



Ultrasonen sensor

Hiervoor construeren we een denkbeeldig segment die loodrecht staat op het midden van de voorkant van de robot. We berekenen de intersecties van dat segment met de muren. Daarna berekenen we de afstand van elke intersectie


tot het midden van de voorkant van de robot. De kleinste afstand is de waarde van de ultrasone sensor ^[SimulatorInterface.hs: 58].

Line follow sensor

Hiervoor moeten we controleren of een bepaalde coördinaat zich op een segment bevindt. We staan toe dat een coördinaat zich maximaal een half vakje van het segment bevindt. Eerst berekenen we het punt X op de lijn, waartoe het segment behoort, die zich het dichtst bij de coördinaat bevindt. We moeten controleren of X zich effectief op het segment bevindt. X bevindt zich op het segment gedefinieerd door punten A en B a.s.a $|AX| + |XB| = |AB|$. Indien X zich op het segment bevindt en de afstand tussen X en de coördinaat kleiner is dan een half vakje, dan bevindt de coördinaat zich op het segment. Deze berekening gebeurt in de functie `getLineStatus`

^[SimulatorInterface.hs: 83].

6. Conclusie

De programmeertaal  kan gebruikt worden om een al dan niet virtuele mBot aan te sturen. De meeste klassieke functionaliteiten, zoals aritmetische en booleaanse expressies, conditionele statements en lussen worden ondersteund door de taal. Verder ontwikkelden we ook een simulator waarin een mBot kan rondrijden in een fictieve wereld. De mBot in deze simulator ondersteunt alle mogelijkheden die de echte mBot aanbiedt.

Een obstakel waarmee we te maken kregen is dat niet alle programma's even goed werken in de simulator als in de echte wereld. De belangrijkste oorzaak hiervan is dat de wiskundige formules die gebruikt worden in de simulatie geen perfecte weerspiegeling zijn van de werkelijkheid. Hierdoor gedraagt de mBot zich niet altijd helemaal zoals je zou verwachten en is bv. het volgen van de lijn soms schokkerig. Door empirisch wat te spelen met de parameters van de simulatie hebben we dit probleem echter deels kunnen verhelpen.

Verder zijn we persoonlijk wel tevreden over onze implementatie. We hebben er heel wat tijd in gestoken maar zijn wel fier op het resultaat. Indien de tijd ons dat toeliet, hadden wij uiteraard nog enkele verbeteringen kunnen doorvoeren zoals:

- **Meerdere gegevenstypes**

Momenteel is het enkel mogelijk om gehele getallen op te slaan in variabelen. De taal zou nog uitgebreid kunnen worden zodat het ook mogelijk is om vlottende kommagetallen en booleaanse waarden op te slaan.

- **Foutafhandeling**

Op dit moment is foutafhandeling nog erg beperkt. Zo zijn de foutmeldingen die opgegooid worden wanneer de invoer een syntaxfout bevat niet erg behulpzaam.

- **Realistischere simulatie**

Zoals hierboven reeds vermeld, zou de simulatie nog uitgebreid kunnen worden zodat deze de werkelijkheid beter benadert.

- **Verbeterde collision detection**

Momenteel worden de uitstekende wielen van de mBot genegeerd bij collision detection. Deze ook in rekening brengen zou de simulatie eveneens realistischer maken.

7. Broncode

Parser.hs

```
1 | module Parser where
2 |
3 | import           Control.Applicative
4 | import           Control.Monad
5 | import           Control.Monad.State
6 | import           Data.Char
7 | import           Data.Foldable
8 | import           Data.List
9 | import qualified Data.Map           as Map
10 | import           Data.Maybe
11 | import           Evaluator
12 | import           Prelude             hiding (Left, Right)
13 | import qualified Text.Read          as Read
14 |
15 | -- Let's start by creating the Parser monad we all know and love. After creating the monad-instance, we can create the
16 | -- functor and applicative instance with minimal effort.
17 | newtype Parser a = Parser { parse :: String -> [(a, String)] }
18 |
19 | instance Functor Parser where
20 |     fmap = liftM
21 |
22 | instance Applicative Parser where
23 |     pure = return
24 |     (<*>) = ap
25 |
26 | instance Monad Parser where
27 |     return x = Parser $ \inp -> [(x, inp)]
28 |     x >=> f = Parser $ \inp -> concat [parse (f x') inp' | (x', inp') <- parse x inp]
29 |
30 | instance Alternative Parser where
31 |     empty = mzero
32 |     (<|>) = mplus
33 |
34 | instance MonadPlus Parser where
35 |     mzero      = Parser $ const []
36 |     f `mplus` g = Parser $ \inp -> parse f inp ++ parse g inp
37 |
38 | -- A parser that consumes a single character if the input is non-empty and fails otherwise.
39 | item :: Parser Char
40 | item = Parser $ \inp -> case inp of
```

```

41     [] -> []
42     (x:xs) -> [(x, xs)]
43
44 -- A parser that consumes a single character that satisfies the given predicate and fails otherwise.
45 sat :: (Char -> Bool) -> Parser Char
46 sat p = do c <- item
47         guard (p c)
48         return c
49
50 -- Various parsers that consume a single character of the specified type.
51 digit  = sat isDigit
52 lower  = sat isLower
53 upper  = sat isUpper
54 letter = sat isAlpha
55 alphanum = sat isAlphaNum
56 char x  = sat (==x)
57
58 -- A parser that consumes the specified string
59 string :: String -> Parser ()
60 string [] = return ()
61 string (x:xs) = do char x
62                  string xs
63                  return ()
64
65 -- A parser that consumes an identifier (i.e. an alphanumeric string starting with a lowercase letter).
66 ident :: Parser String
67 ident = do c <- letter
68         cs <- many alphanum
69         return (c:cs)
70
71 -- A parser that consumes a natural number.
72 nat :: Parser Int
73 nat = fmap toInt (first (some digit))
74     where toInt = foldl (\n c -> 10 * n + (ord c - ord '0')) 0
75
76 -- A parser that consumes an integer (i.e. either a natural number or a natural number prefixed with a minus sign).
77 int :: Parser Int
78 int = do char '-'
79         fmap negate nat
80     <|> nat
81
82 -- A parser that applies the three parsers `open`, `p` and `close` one after another. Only the results of `p` are kept
83 -- and returned. This is useful to take care of brackets, hence the name.
84 brackets :: Parser () -> Parser b -> Parser () -> Parser b
85 brackets open p close = do open
86                          x <- p
87                          close
88                          return x
89
90 -- A parser that recognizes non-empty sequences of `p` where instances of `p` are separated by `sep`.
91 sepby1 :: Parser a -> Parser b -> Parser [a]
92 p `sepby1` sep = do x <- p
93                  xs <- first (many (sep >> p))
94                  return (x:xs)
95
96 -- A parser that consumes the strings produced by grammar 'E -> E | E `op` p'.
97 chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
98 p `chainl1` op = do x <- p
99                  fys <- many (do f <- op
100                                y <- p
101                                return (f, y))
102                  return (foldl (\l (f, r) -> f l r) x fys)
103
104 -- A parser that transforms the given parser by only keeping it first (and thus longest) parse.
105 first :: Parser a -> Parser a
106 first p = Parser $ \inp -> case parse p inp of
107     [] -> []
108     (x:_) -> [x]
109
110 -- A parser that applies each of the given parsers and returns its associated value iff it succeeds.
111 ops :: [(Parser a, b)] -> Parser b
112 ops xs = foldr1 (<|>) [p >> return val | (p, val) <- xs]
113
114 -- A parser that consumes whitespace, but not newlines. Note that this parser can fail if there's no whitespace left to
115 -- consume.
116 spaces :: Parser ()
117 spaces = first . void $ some (sat isWhite)
118     where isWhite c = isSpace c && c /= '\n'
119
120 -- A parser that applies the given parser and then tries to consume any remaining whitespace till the end of the line.
121 token :: Parser a -> Parser a
122 token p = first (do x <- p
123                  spaces <|> return ())

```

```

124         return x)
125
126 -- Various parser that consuming any remaining whitespace till the end of the line after consuming the specified type
127 -- of input. These parsers will prove very useful for implementing higher-level parsers.
128 integer :: Parser Int
129 integer = token int
130
131 symbol :: String -> Parser ()
132 symbol = token . string
133
134 identifier :: Parser String
135 identifier = token ident
136
137 boolean :: Parser Bool
138 boolean = ops [(trueSymbol, True), (falseSymbol, False)]
139
140 newline :: Parser ()
141 newline = void (token (char '\n'))
142
143 -- A parser that consumes the indent symbol emitted by the preprocessor.
144 indent :: Parser ()
145 indent = newline >> indentSymbol >> newline
146
147 -- A parser that consumes the dedent symbol emitted by the preprocessor.
148 dedent :: Parser ()
149 dedent = newline >> dedentSymbol >> return ()
150
151 -- A parser for an indented sequence of statements. Note that we don't consume a possible newline after the dedent
152 -- symbol. This corresponds to the notion that each statement (and thus also an indented statement sequence) ends with
153 -- a newline character.
154 block :: Parser Stmt
155 block = do indent
156         body <- statementSeq
157         dedent
158         return body
159
160 -- A parser for arithmetic expressions. The parsers defined below correspond nicely to the BNF described in the report.
161 aExpression = aTerm `chainl1` ops [(addSymbol, (+:)), (subtractSymbol, (-:))]
162
163 aTerm :: Parser AExpr
164 aTerm = aFactor `chainl1` ops [(multiplySymbol, (:*)), (divideSymbol, (:/:))]
165
166 aFactor :: Parser AExpr
167 aFactor = aConstant
168         <|> aSensor
169         <|> fmap AVar identifier
170         <|> brackets openParSymbol aExpression closeParSymbol
171
172 aConstant :: Parser AExpr
173 aConstant = fmap AConst (integer
174         <|> ops [(zeroSymbol, 0), (oneSymbol, 1), (twoSymbol, 2), (threeSymbol, 3)])
175
176 aSensor :: Parser AExpr
177 aSensor = fmap ASensor $ ops [(lineSymbol, Line), (distanceSymbol, Distance)]
178
179 -- A parser for boolean expressions. The parsers defined below again correspond nicely to the BNF in the report.
180 bExpression :: Parser BExpr
181 bExpression = bTerm `chainl1` ops [(orSymbol, (:|:))]
182
183 bTerm :: Parser BExpr
184 bTerm = bFactor `chainl1` ops [(andSymbol, (:&:))]
185
186 bFactor :: Parser BExpr
187 bFactor = fmap BConst boolean
188         <|> fmap Not (negateSymbol >> bFactor)
189         <|> liftM2 (:<:) aExpression (ltSymbol >> aExpression)
190         <|> liftM2 (:=:) aExpression (eqSymbol >> aExpression)
191         <|> liftM2 (:>) aExpression (gtSymbol >> aExpression)
192         <|> brackets openParSymbol bExpression closeParSymbol
193
194 -- A parser for a sequence of statements. A sequence of statements consists of one or more statements separated by a
195 -- newline character.
196 statementSeq :: Parser Stmt
197 statementSeq = fmap Seq (singleStatement `sepby1` (char '\n'))
198
199 -- A parser for a single statement. Note that this parser consumes any leading whitespace before attempting to parse
200 -- the actual statement.
201 singleStatement :: Parser Stmt
202 singleStatement = do spaces <|> return ()
203                 assignStatement <|> ifStatement <|> skipStatement <|> whileStatement <|> cmdStatement
204
205 -- A parser for the assignment statement. We first parse the identifier, then consume and discard the assignment symbol
206 -- and finally the arithmetic expression. This is then wrapped in an `Assign` statement.

```

```

207 assignStatement :: Parser Stmt
208 assignStatement = liftM2 Assign identifier (assignSymbol >> aExpression)
209
210 -- A parser for an if statement. An if statement consists of the if symbol, followed by a boolean expression and an
211 -- indented block. We then attempt to parse any else-if clauses and final the else clause. Note that an else clause is
212 -- treated as a final else-if clause with condition 'true'. If there's no else clause, we pretend the else body consists
213 -- of a single skip statement. That way, we don't have to discern between the presence/absence of an elseclause.
214 ifStatement :: Parser Stmt
215 ifStatement = do ifSymbol
216   cond <- bExpression
217   body <- block
218   elifClauses <- first (many elifClause)
219   elseClause <- (newline >> elseSymbol >> block) <|> (return Skip)
220   return . If $ [(cond, body)] ++ elifClauses ++ [(BConst True, elseClause)]
221   where elifClause :: Parser (BExpr, Stmt)
222         elifClause = newline >> elifSymbol >> liftM2 (,) bExpression block
223
224 -- A parser for a while statement. It parser the while symbol and boolean expression on the same line and then the
225 -- indented block that forms the body of the loop.
226 whileStatement :: Parser Stmt
227 whileStatement = do whileSymbol
228   cond <- bExpression
229   body <- block
230   return (While cond body)
231
232 -- A parser for a command statement. The command is always one of the pre-defined types.
233 cmdStatement :: Parser Stmt
234 cmdStatement = driveCmdStatement <|> sleepCmdStatement <|> lightCmdStatement
235
236 -- Parsers for the various commands. Most of these should be straightforward. They all first parse the symbol indicating
237 -- the command and then attempt to parse their arguments.
238
239 -- A parser for the drive command. We first parse the drive symbol, followed by the symbol signifying the direction.
240 driveCmdStatement :: Parser Stmt
241 driveCmdStatement = do driveSymbol
242   fmap (Exec . Drive) $ ops [(leftSymbol, Left), (rightSymbol, Right),
243                               (upSymbol, Up), (downSymbol, Down)]
244   --
245
246 -- A parser for the sleep command. We first parse the sleep symbol, and then the duration.
247 sleepCmdStatement :: Parser Stmt
248 sleepCmdStatement = do sleepSymbol
249   duration <- fmap Exact aExpression
250   <|> ops [(veryShortSymbol, VeryShort), (shortSymbol, Short),
251           (mediumSymbol, Medium), (longSymbol, Long), (veryLongSymbol, VeryLong)]
252   return . Exec . Sleep $ duration
253
254 -- A parser for a light command. We first parse the light symbol, followed by either the left or right flank symbol and
255 -- then three arithmetic expressions signifying the RGB-values we want to set the light to.
256 lightCmdStatement :: Parser Stmt
257 lightCmdStatement = do lightSymbol
258   flank <- ops [(leftFlankSymbol, LeftFlank), (rightFlankSymbol, RightFlank)]
259   cmd <- liftM3 (Light flank) aExpression aExpression aExpression
260   return . Exec $ cmd
261
262 -- A parser for skip (= comment) statements. That is, it consumes the skip symbol and then consumes all remaining
263 -- characters until the end of the line.
264 skipStatement :: Parser Stmt
265 skipStatement = do skipSymbol
266   first . many $ sat (/='\n')
267   return Skip
268
269 -- Executes the preprocessor step. Before we add the indents, we first remove all the lines containing only whitespace
270 -- and then add a final newline at the end of the script to make sure each statement ends with a newline. The initial
271 -- stack of indents consists of just a single 0, signifying that the initial indent level is zero.
272 preprocess :: String -> String
273 preprocess = unlines . (flip evalState [0]) . addIndents . (+ [""]) . filter (not . all isSpace) . lines
274
275 -- Adds an indent symbol each time the indentation level increases and a dedent symbol each time it decreases in the
276 -- given list of lines. This function utilizes the state monad to carry the stack of current indentation levels around.
277 addIndents :: [String] -> State [Int] [String]
278 addIndents (l:ls) = do indents <- get
279   let cur = length . takeWhile isSpace $ l
280   l' <- case compare cur (head indents) of
281     GT -> do modify (cur:)
282             return ("{" ++ l)
283     LT -> do let indents' = dropWhile (>cur) indents
284             put indents'
285             let diff = length indents - length indents'
286             return ((concat . replicate diff $ "}") ++ l) -- much dedent symbols
287     EQ -> return l
288   -- Indent level increased,
289   -- so output an indent symbol
290   -- Indent level decreased, so
291   -- pop all bigger indents from
292   -- the stack and output just as
293   -- much dedent symbols
294   -- Indent level didn't change

```

```

290 liftM2 (:) (return l') (addIndents ls)
291
292 -- In order not to clutter our code with Emoji, we aggregate all the tokens used in our language here. Note that
293 -- `symbol` is a function that maps a string to a parser that consumes that string and any remaining whitespace.
294 skipSymbol    = symbol " "
295 whileSymbol   = symbol "🔄"
296 ifSymbol      = symbol "?"
297 elifSymbol    = symbol "!!"
298 elseSymbol    = symbol "!"
299 sleepSymbol   = symbol "😴"
300 veryShortSymbol = symbol "⏏"
301 shortSymbol   = symbol "⏴"
302 mediumSymbol  = symbol "⏵"
303 longSymbol    = symbol "⏶"
304 veryLongSymbol = symbol "⏷"
305 upSymbol      = symbol "⬆️"
306 downSymbol    = symbol "⬇️"
307 leftSymbol    = symbol "⬅️"
308 rightSymbol   = symbol "➡️"
309 driveSymbol   = symbol "🚗"
310 trueSymbol    = symbol "👉"
311 falseSymbol   = symbol "👈"
312 leftFlankSymbol = symbol "⚔️"
313 rightFlankSymbol = symbol "⚔️"
314 distanceSymbol = symbol "📏"
315 lineSymbol    = symbol "⚡"
316 assignSymbol  = symbol "🔄"
317 lightSymbol   = symbol "💡"
318 zeroSymbol    = symbol "0️⃣"
319 oneSymbol     = symbol "1️⃣"
320 twoSymbol     = symbol "2️⃣"
321 threeSymbol   = symbol "3️⃣"
322 ltSymbol      = symbol "<"
323 eqSymbol      = symbol "=="
324 gtSymbol      = symbol ">"
325 addSymbol     = symbol "+"
326 subtractSymbol = symbol "-"
327 multiplySymbol = symbol "*"
328 divideSymbol  = symbol "/"
329 andSymbol     = symbol "&&"
330 orSymbol      = symbol "||"
331 negateSymbol  = symbol "!"
332 indentSymbol  = symbol "{"
333 dedentSymbol  = symbol "}"
334 openParSymbol = symbol "("
335 closeParSymbol = symbol ")"

```

Evaluator.hs

```

1 module Evaluator (AExpr(..), BExpr(..),
2                   Stmt(..), Command(..), Device(..), Sensor(..), Direction(..), Duration(..), Flank(..),
3                   runStmt) where
4
5 import           Control.Applicative
6 import           Control.Monad
7 import           Control.Monad.Except
8 import           Control.Monad.Identity
9 import           Control.Monad.Reader
10 import           Control.Monad.State
11 import           Data.Char
12 import           Data.Foldable
13 import           Data.List
14 import qualified Data.Map           as Map
15 import           Data.Maybe
16 import           Prelude           hiding (Left, Right)
17
18 data AExpr = AConst Int
19           | AVar Name
20           | ASensor Sensor
21           | AExpr :+: AExpr
22           | AExpr :-: AExpr
23           | AExpr :*: AExpr
24           | AExpr :/: AExpr
25
26 data BExpr = BConst Bool
27           | Not BExpr
28           | BExpr :&: BExpr
29           | BExpr :|: BExpr
30           | AExpr :<: AExpr

```



```

31 | AExpr :=: AExpr
32 | AExpr >: AExpr
33
34 data Stmt = Assign String AExpr
35 | Seq [Stmt]
36 | If [(BExpr, Stmt)]
37 | While BExpr Stmt
38 | Exec Command
39 | Skip
40
41 data Command = Drive Direction
42 | Sleep Duration
43 | Light Flank AExpr AExpr AExpr
44
45 data Sensor = Line | Distance
46 data Direction = Left | Right | Up | Down deriving (Eq)
47 data Duration = VeryShort | Short | Medium | Long | VeryLong | Exact AExpr
48 data Flank = LeftFlank | RightFlank deriving (Eq)
49
50 -- By passing a device to the evaluator, the user can choose ad-hoc how to handle each supported command. The evaluator
51 -- itself is device-agnostic and just calls the appropriate methods on the passed device. This allows the user to
52 -- easily switch between running on a physical device (i.e. the mBot itself) or a virtual device (i.e. the simulator)
53 data Device = Device {
54   sleep      :: Int -> IO (),           -- Sleep for the specified number of milliseconds
55   setRGB     :: Int -> Int -> Int -> IO (), -- Set left (0) or right (1) LED to the specified RGB values
56   setMotor   :: Int -> Int -> IO (),     -- Set left and right motor speeds
57   readDistance :: IO Int,               -- Read the distance reported by the ultrasonic sensor
58   readLine   :: IO Int                 -- Read the measurement reported by the line follower
59 }
60
61 -- The environment used in the evaluator consists of a map from variable names to their corresponding values. This map
62 -- is passed around using the state monad transformer and can thus be modified when assigning to variables.
63 type Name = String
64 type Mem = Map.Map Name Int
65
66 -- Next to the mutable state described above, an immutable `Device` is also carried around in the evaluator. This device
67 -- is the device that commands are sent to. This lets us easily swap the physical mBot for the simulator or vice versa.
68 -- Finally, an except monad transformer is used to gracefully deal with errors like undefined variables.
69 type Eval a = ReaderT Device (StateT Mem (ExceptT String IO)) a
70
71 -- This function actually executes the specified evaluator with the specified device. Its result is an IO-action that
72 -- executes the program.
73 runEval :: Device -> Eval () -> IO (Either String ())
74 runEval device evaluator = runExceptT (evalStateT (runReaderT evaluator device) Map.empty)
75
76 -- As a convenience method for users of the evaluator, we offer a function that takes a device and statement and returns the
77 -- IO-action that executes the program with the specified device.
78 runStmt :: Device -> Stmt -> IO (Either String ())
79 runStmt device stmt = runEval device (eval stmt)
80
81 -- An evaluator for boolean expressions. The operators is our language map nicely to operators built into Haskell.
82 evalB :: BExpr -> Eval Bool
83 evalB (BConst b) = return b
84 evalB (Not b) = fmap not (evalB b)
85 evalB (b1 & b2) = liftM2 (&) (evalB b1) (evalB b2)
86 evalB (b1 || b2) = liftM2 (||) (evalB b1) (evalB b2)
87 evalB (a1 < a2) = liftM2 (<) (evalA a1) (evalA a2)
88 evalB (a1 == a2) = liftM2 (==) (evalA a1) (evalA a2)
89 evalB (a1 > a2) = liftM2 (>) (evalA a1) (evalA a2)
90
91 -- An evaluator for arithmetic expressions. The operators again map nicely to those built into Haskell.
92 evalA :: AExpr -> Eval Int
93 evalA (AConst c) = return c
94 evalA (AVar name) = gets (Map.lookup name) >=> maybe (throwError $ "Undefined variable: " ++ name) return
95 evalA (ASensor Line) = asks readLine >=> liftIO -- Return the value of executing readLine on the device
96 evalA (ASensor distance) = asks readDistance >=> liftIO -- Return the value of executing readDistance on the device
97 evalA (a1 + a2) = liftM2 (+) (evalA a1) (evalA a2)
98 evalA (a1 - a2) = liftM2 (-) (evalA a1) (evalA a2)
99 evalA (a1 * a2) = liftM2 (*) (evalA a1) (evalA a2)
100 evalA (a1 / a2) = liftM2 div (evalA a1) (evalA a2)
101
102 -- Evaluates a duration. A duration is an arithmetic expression, in which case the expression is evaluated to determine
103 -- its value, or a built-in constant.
104 evalDuration :: Duration -> Eval Int
105 evalDuration (Exact a) = evalA a
106 evalDuration constant = return $ case constant of
107   VeryShort -> 400
108   Short      -> 800
109   Medium     -> 1200
110   Long       -> 1600
111   VeryLong   -> 2000
112
113 -- An evaluator for statements. Note that this evaluator doesn't return anything since the sole purpose of evaluating

```

```

114 -- a statement is its side effects.
115 eval :: Stmt -> Eval ()
116
117 -- Evaluates a sequence of statement by evaluating each statement in turn.
118 eval (Seq stmts) = forM stmts eval
119
120 -- Evaluates an assignment statement by first evaluating the arithmetic expression on the right and then inserting the
121 -- value under the corresponding name in the map of variables.
122 eval (Assign name e) = do val <- evalA e
123                        modify (Map.insert name val)
124
125 -- Evaluates an if statement by first evaluating the conditions of each branch, and then evaluating the first branch
126 -- whose condition did evaluate to true, or evaluating nothing if the condition of all branches is false.
127 -- Note that an eventual else branch always appears last with `BConst True` as its condition.
128 eval (If branches) = do conditions <- mapM (evalB . fst) branches
129                        case findIndex id conditions of
130                          Just i  -> eval . snd . (!!i) $ branches
131                          Nothing -> return ()
132
133 -- Evaluates a while statement by evaluating the condition, and iff the condition is true, evaluating the body and then
134 -- restarting this procedure.
135 eval (While e stmt) = do cond <- evalB e
136                        when cond (eval stmt >> eval (While e stmt))
137
138 -- Evaluates a skip statement by doing nothing. A skip statement is a comment in the source code so it simply shouldn't
139 -- do anything.
140 eval Skip = return ()
141
142 -- Evaluates a statement calling the sleep command by first evaluating the duration and then executing the sleep action
143 -- for the determined amount of time.
144 eval (Exec (Sleep dur)) = do millis <- evalDuration dur
145                        asks sleep >=> liftIO . ($millis)
146
147 -- Evaluates a statement calling the drive command. We first lookup the desired speed of both motors based on the
148 -- direction and then send a command to the device with these speeds.
149 eval (Exec (Drive dir)) = do let speed = fromJust . (`Data.List.lookup` directions) $ dir
150                        asks setMotor >=> (liftIO . ($speed) . uncurry)
151                        where directions = [(Left, (0, 70)), (Right, (70, 0)),
152                                           (Up, (70, 70)), (Down, (-70, -70))]
153
154 -- Evaluates a statement calling the light command. We first inspect the flank to know which of the 2 light indices to
155 -- send to the device, and then evaluate each of the three arithmetic expressions passed as arguments before sending a
156 -- command to the device with those values.
157 eval (Exec (Light flank r g b)) = do let light = if flank == LeftFlank then 1 else 2
158                        cmd <- asks setRGB
159                        liftIO <=< liftM4 cmd (return light) (evalA r) (evalA g) (evalA b)

```

Gui.hs

```

1  module Gui where
2
3  import           Control.Concurrent           (MVar, forkIO, newMVar,
4                                                putMVar, readMVar, takeMVar,
5                                                threadDelay)
6
7  import           Data.Fixed
8  import           Data.Maybe                 (mapMaybe)
9  import           Graphics.Gloss
10 import           Graphics.Gloss.Data.Vector
11 import           Graphics.Gloss.Geometry.Angle
12 import           Graphics.Gloss.Geometry.Line
13 import qualified Graphics.Gloss.Interface.IO.Game as G
14 import           Util
15 import           WorldParser
16
17 -- The size of one cell.
18 cell = 32.0
19
20 -- Renders the world.
21 render :: G.Picture -- Picture of a wall
22        -> World      -- The world that should be rendered
23        -> G.Picture -- Picture of the world
24 render wp (World robot walls lns) = G.pictures $
25     map (\ln -> renderPicAt (linePicture ln) $ fst ln) lns
26   ++ map (renderPicAt wp) walls
27   ++ [renderPicAt (G.rotate angle $ robotPicture robot) position]
28
29 where position = rPosition robot
30       angle = radToDeg $ rAngle robot
31       size which = maximum $ map which walls

```

[illegible]

```

114 runSimulator :: MVar World -> IO ()
115 runSimulator m = do world <- readMVar m
116                  [wp] <- mapM loadBMP ["images/wall.bmp"]
117                  G.playIO (G.InWindow "MBot" (700,500) (0,0)) -- display
118                          G.white -- background
119                          60 -- fps
120                          world -- initial world
121                          (return . render wp) -- render world
122                          (const return) -- handle input
123                          (step m) -- step world in time

```

Initialize.hs

```

1 module Initialize where
2
3 import Control.Exception
4 import Control.Monad
5 import Evaluator
6 import Parser
7 import System.Environment
8 import System.Exit
9 import System.IO
10
11 -- Parses the arguments passed to the program and executes the script passed as the argument on the specified device.
12 initialize :: Device -> IO ()
13 initialize mDevice = do args <- getArgs
14                      when (length args /= 1) (die "Expects the script to execute as only argument")
15                      input <- catch (readFile . head $ args) (readHandler . head $ args)
16                      let prs = parse statementSeq . preprocess $ input
17                      unless (null prs) (do let prog = fst (head prs)
18                                           void $ runStmnt mDevice prog)
19
20 readHandler :: String -> IOError -> IO a
21 readHandler name _ = die ("Cannot open file: " ++ name)

```

Util.hs

```

1 module Util where
2
3 import WorldParser
4
5 rotateAround :: Coord -- Origin
6              -> Angle -- Angle in radians
7              -> Coord -- Coordinate that should be rotated
8              -> Coord -- Updated coordinate
9 rotateAround (xo, yo) angle (x, y) = ( xo + (x - xo) * cos angle - (y - yo) * sin angle
10                                     , yo + (x - xo) * sin angle + (y - yo) * cos angle)
11
12 -- Calculates the distance between 2 coordinates
13 distance :: Coord -> Coord -> Float
14 distance (x0, y0) (x1, y1) = sqrt ((x1 - x0) ** 2 + (y1 - y0) ** 2)

```

WorldParser.hs

```

1 module WorldParser where
2
3 import Data.List
4 import Data.Maybe
5 import Data.Tuple ()
6
7 type X = Float
8 type Y = Float
9
10 type Coord = (X, Y)
11 type Angle = Float
12
13 type Line = (Coord, Coord)
14 type Color = (Int, Int, Int)
15
16 data Robot = Robot { rSpeedLeft  :: Int
17                   , rSpeedRight :: Int
18                   , rPosition   :: Coord

```

```

19         , rAngle      :: Angle
20         , rColorLeft  :: Color
21         , rColorRight :: Color
22     } deriving (Eq, Ord, Show)
23
24 data World = World { wRobot :: Robot
25                   , wWalls :: [Coord]
26                   , wLines :: [Line]
27                   } deriving (Eq, Ord, Show)
28
29 emptyRobot = Robot 0 0 (0.0, 0.0) 0 (255, 255, 255) (255, 255, 255)
30 emptyWorld = World { wRobot = emptyRobot, wWalls = [], wLines = [] }
31
32 addPiece :: Coord -> Char -> World -> World
33 addPiece co ch w
34   | ch `elem` wallChars = w { wWalls = co:wWalls w }
35   | ch `elem` botChars  = w { wRobot = emptyRobot { rPosition = co, rAngle = angle } }
36   | otherwise          = w
37   where wallChars = ['X', '+', '|', '-']
38         botChars  = ['>', 'v', '<', '^']
39         angle     = (fromIntegral . fromJust $ elemIndex ch botChars) * pi / 2
40
41
42 -- Adds the pieces specified in the ASCII input representation of the grid world to the world.
43 -- Note that this method expects only the representation of the grid itself and not the line segments underneath.
44 addPieces :: World -> String -> World
45 addPieces w txt = foldr (uncurry addPiece) w withCoords
46   where withCoords = [(x, y), c] | (y, line) <- zip [0..] (lines txt), (x, c) <- zip [0..] line
47
48 -- Adds line segments to the world based on a sequence of line segments represented in the format "(x1, y1), (x2, y2)".
49 addLines :: World -> [String] -> World
50 addLines w [] = w
51 addLines w (l:ls) = w { wLines = (x, y):wLines (addLines w ls) }
52   where [x, y] = map read (words l)
53
54 -- Creates a world based on the ASCII input representation of that world.
55 makeWorld :: String -> World
56 makeWorld txt = addLines (addPieces emptyWorld (unlines pcs)) lns
57   where (pcs, lns) = span (notElem '(') (lines txt)

```

SimulatorInterface.hs

```

1 module SimulatorInterface where
2
3 import           Control.Concurrent           (MVar, forkIO, newMVar, putMVar,
4                                               readMVar, takeMVar, threadDelay)
5
6 import           Data.Fixed
7 import           Data.Maybe                   (mapMaybe)
8 import           Graphics.Gloss.Data.Vector
9 import           Graphics.Gloss.Geometry.Angle
10 import           Graphics.Gloss.Geometry.Line
11 import           Gui
12 import           Util
13 import           WorldParser
14
15 newtype Simulator = Simulator (MVar World)
16 type Command = World -> World
17
18 -- Creates a new simulator and runs it on another thread.
19 openSimulator :: IO Simulator
20 openSimulator = do
21   world <- fmap makeWorld (readFile "worlds/world1.txt")
22   m <- newMVar world
23   let s = Simulator m
24   forkIO (runSimulator m)
25   return s
26
27 -- Sends a command to the simulator.
28 sendCommand :: Simulator -> Command -> IO ()
29 sendCommand (Simulator m) command = do world <- takeMVar m
30   let world' = command world
31   putMVar m world'
32
33 -- Creates a command that changes the rgb values of the leds of the robot
34 setRGB :: Int -- The left that should be changed. 1 = left and 2 = right
35       -> Int -- red
36       -> Int -- green
37       -> Int -- blue
38       -> Command

```

```

38 setRGB side r g b world = case side of
39   1 -> world { wRobot = robot { rColorLeft=(r, g, b)}}
40   2 -> world { wRobot = robot { rColorRight=(r, g, b)}}
41   where robot = wRobot world
42
43 -- Creates a command that sets the speed of the motors of the robot.
44 setMotor :: Int      -- Speed of the left wheel
45          -> Int      -- Speed of the right wheel
46          -> Command
47 setMotor l r world = world { wRobot = robot {rSpeedLeft = l, rSpeedRight = r}}
48   where robot = wRobot world
49
50 -- Returns the distance between the front of the robot and the nearest wall.
51 readUltraSonic :: Simulator -> IO Float
52 readUltraSonic (Simulator m) = do world <- readMVar m
53                                return $ getDistance world
54
55 -- Calculates the distance between the front of the robot and the nearest wall.
56 -- The front of the robot is equal to (x + 1, y + 0.5) when the robot is looking to the right
57 -- where (x, y) is the coordinate of the robot.
58 getDistance :: World -> Float
59 getDistance world@(World robot walls _) = minimum $ map (distance (xo, yo)) intersections
60   where position@(x, y) = rPosition robot
61         angle = rAngle robot
62         origin@(xo, yo) = rotateAround (x + 0.5, y + 0.5) angle (x + 1.0, y + 0.5)
63         end = rotateAround (x + 0.5, y + 0.5) angle (x + 50, y + 0.5)
64         intersectionsWith (x', y') = mapMaybe (uncurry $ intersectSegSeg origin end)
65                                     [ ((x', y'), (x' + 1, y'))
66                                       , ((x' + 1, y'), (x' + 1, y' + 1))
67                                       , ((x' + 1, y' + 1), (x', y' + 1))
68                                       , ((x', y' + 1), (x', y'))
69                                     ]
70         intersections = concatMap intersectionsWith walls
71
72 -- Reads the line status of the robot.
73 -- The possible return values are equal to:
74 -- 0 = no sensor detects a black line
75 -- 1 = the right sensor detects a black line
76 -- 2 = the left sensor detects a black line
77 -- 3 = both sensors detect a black line
78 readLineFollower :: Simulator -> IO Int
79 readLineFollower (Simulator m) = do world <- readMVar m
80                                return $ getLineStatus world
81
82 -- Calculates the line status
83 getLineStatus :: World -> Int
84 getLineStatus world@(World robot _ lns)
85   | isOnBlack left && isOnBlack right = 3
86   | isOnBlack right = 1
87   | isOnBlack left = 2
88   | otherwise = 0
89   where (x, y) = rPosition robot
90         angle = rAngle robot
91         left = rotateAround (x + 0.5, y + 0.5) angle (x + 1.0, y + 0.4)
92         right = rotateAround (x + 0.5, y + 0.5) angle (x + 1.0, y + 0.6)
93         isOnSegment point (start, end) = distance closest point <= 1 / 2
94                                     && abs (distance start end - (distance start closest + distance closest end)) < 0.001
95         where closest = closestPointOnline start end point
96         lns' = map adjustLine lns
97         isOnBlack point = any (isOnSegment point) lns'
98
99 -- Translates the coordinates so that they will be in the middle of a square instead of the beginning.
100 adjustLine :: Line -> Line
101 adjustLine ((x0, y0), (x1, y1)) = ( (x0 + sin angle / 2, y0 + cos angle / 2)
102                                     , (x1 + sin angle / 2, y1 + cos angle / 2) )
103   where angle = argV (abs (x1 - x0), abs (y1 - y0))

```

Simulator.hs

```

1 module Simulator where
2
3 import Control.Concurrent (threadDelay)
4 import Evaluator
5 import Initialize (initialize)
6 import qualified SimulatorInterface as S
7
8 main = do s <- S.openSimulator
9         let mDevice = simulatorDevice s
10         initialize mDevice

```

```

11 |
12 | simulatorDevice s = Device {
13 |     sleep      = \x -> threadDelay (x * 1000),
14 |     setRGB     = \l r g b -> threadDelay 10 >> S.sendCommand s (S.setRGB l r g b),
15 |     setMotor   = \l r -> threadDelay 10 >> S.sendCommand s (S.setMotor (l `quote` 5) (r `quote` 5)),
16 |     readDistance = do val <- S.readUltraSonic s
17 |                     threadDelay 10
18 |                     return (round val * 40),
19 |     readLine   = do line <- S.readLineFollower s
20 |                 threadDelay 10
21 |                 return line
22 | }

```

Interpreter.hs

```

1 | module Interpreter where
2 |
3 | import           Control.Concurrent (threadDelay)
4 | import           Evaluator
5 | import           Initialize
6 | import qualified MBot
7 |
8 | main = do d <- MBot.openMBot
9 |         let mDevice = botDevice d
10 |         initialize mDevice
11 |         MBot.closeMBot d
12 |
13 | -- A device that forwards commands to the physical mBot.
14 | botDevice d = Device {
15 |     sleep      = threadDelay . (*1000),
16 |     setRGB     = \l r g b -> MBot.sendCommand d $ MBot.setRGB l r g b,
17 |     setMotor   = \l r -> MBot.sendCommand d $ MBot.setMotor l r,
18 |     readDistance = fmap round (MBot.readUltraSonic d),
19 |     readLine   = fmap lineToInt (MBot.readLineFollower d)
20 | }
21 |
22 | -- Converts the line sensor reading of the physical mBot to an integer.
23 | lineToInt :: MBot.Line -> Int
24 | lineToInt MBot.BOTHW  = 0
25 | lineToInt MBot.RIGHTB = 1
26 | lineToInt MBot.LEFTB  = 2
27 | lineToInt MBot.BOTHB  = 3

```