Functioneel Programmeren

Pieter De Clercq

Inhoudsopgave

1 Inleiding **4**

2 Syntaxis **5**
2.1 Getallen . . . 5
2.2 Expressies . . . 5
2.3 MBot . . . 5
2.4 Statements . . . 6

3 Semantische constructies **7**
3.1 Cook . . . 7
3.2 Eating . . . 7
3.3 Hungry . . . 7
3.4 Order . . . 7
3.5 Puke . . . 7
3.6 Review . . . 8
3.7 Drive (MBot) . . . 8
3.8 Leds (MBot) . . . 8
3.9 Sensoren (MBot) . . . 8

4 Programma's **9**
4.1 Fibonacci . . . 9
4.2 MBot: Politiewagen . . . 9
4.3 MBot: Lijnvolger . . . 9
4.4 MBot: Obstakels ontwijken . . . 10

5 Implementatie **12**
5.1 Alle getallen zijn Doubles . . . 12
5.2 Whitespace . . . 12
5.3 Modulariteit . . . 12
5.4 Booleans zijn Aritmetische expressies . . . 12
5.5 Maybe.fromJust . . . 13
5.6 Tests . . . 13

6 Conclusie **14**
6.1 Gebruikte bronnen . . . 14

7 Appendix Broncode **15**
7.1 Evaluator.hs . . . 15
7.2 Lexer.hs . . . 18
7.3 Main.hs . . . 23
7.4 MBotPlus.hs . . . 24
7.5 Parser.hs . . . 27
7.6 Types.hs . . . 30

7.7 Utils.hs 33
7.8 runtests.hs 34

Hoofdstuk 1

Inleiding

De syntax van deze taal vindt zijn oorsprong in een schematisch idee op papier. Aangezien mijn schema's doorgaans uitpuilen van de pijltjes, zijn deze blijven staan in de effectieve implementatie van de taal. De naam M3NU is afkomstig van het woord Menu en alle identifiers en keywords zijn gebaseerd op culinaire termen. De statements met betrekking tot de MBot echter, vormen een uitzondering op deze regel. Inspiratie voor de werking van de taal heb ik geput uit de bestaande talen waarin ik het liefst programmeer, namelijk C, PHP en Python. Een resultaat hiervan is dat booleans geïmplementeerd zijn als getallen 1 en 0, respectievelijk true en false. Elke aritmetische expressie die niet naar 0 evalueert, wordt booleaans true, equivalent aan de implementatie in Python. Als bestandsextensie werd voor .course gekozen, het Engelse woord voor een "gang" in een menu.

Hoofdstuk 2

Syntaxis

arrow ::= $-$ >
end ::= {';'}$_{1+}$

Getallen

digit ::= 0|1|2|...|9
commasep ::= .
neg ::= $-$
number ::= $\{digit\}_{1+}$ | $\{digit\}_{1+}\{commasep\}_1\{digit\}_{1+}$
sgnnumber ::= $\{number\}_1$ | $\{neg\}_1\{number\}_1$

Expressies

lowercase ::= $a|b|c|...|z$
uppercase ::= $A|B|C|...|Z$
letter ::= $\{lowercase\}_1|\{uppercase\}_1$
variable ::= $\{letter|digit\}_{1+}$
bool ::= *tasty* | *disguisting*
opa ::= + | $-$ | $*$ | /
opb ::= *and* | *or*
opr ::=> | >= | == | <= | <
operator ::= $\{opa\}_1|\{opb\}_1|\{opr\}_1$
exppart ::= $\{sgnnumber\}_1|\{variable\}_1|\{bool\}_1|\{linesensor\}_1|\{ultrason\}_1|\{(exppart)\}_1$
unexp ::='|' $\{exp\}_1$ '|' |!$\{exp\}_1|(\{unexp\}_1)$
binexp ::= $(\{exp\}_1\{operator\}_1\{exp\}_1)$
exp ::= $\{exppart\}_1|\{binexp\}_1|\{(exp)\}_1$

MBot

knowncol ::= *red* | *green* | *blue* | *cyan* | *magenta* | *yellow* | *white*
rgbcol ::= $\{exp\}_1, \{exp\}_1, \{exp\}_1$
color ::= $\{knowncol\}_1|\{rgbcol\}_1$
direction ::= $forward$ | $left$ | *right* | *backward* | $backwardleft$ | *backwardright* | *brake*
leds ::= $left$ | *right*

mbotLed ::= $led\{leds\}_1\{arrow\}\{color\}_1$

Statements

Hoofdstuk 3

Semantische constructies

Cook

Dit statement biedt de mogelijkheid om de uitvoering van het programma te pauzeren voor een gegeven tijdsspanne, uitgedrukt in seconden. De naamgeving voor deze methode komt van de eenvoudige logica, dat er iemand niet kan eten terwijl die aan het koken is, dus de thread waarop het programma draait ook niet.

Eating

While-lussen kunnen gebruikt worden door middel van "Eating"-statements. De werking is niet anders dan bij andere programmeertalen; een blok code wordt uitgevoerd zolang een conditie naar "true", tasty, evalueert.

Hungry

De traditionele "if-else"-controlestructuur werd geïmplementeerd door het "hungry-stuffed"-statement. Als de expressie naar "tasty" evalueert wordt de "hungry"-tak uitgevoerd, anders de "stuffed"-tak. Een noemenswaardig verschil met Haskell is dat de "else/stuffed"-tak van het statement hier optioneel is, terwijl dit in Haskell verplicht is.

Order

Assignment gebeurt met het "Order"-statement. Variabelen worden gestockeerd in de environment en kunnen later in expressies weer gebruikt worden. Scoping van variabelen is niet mogelijk, alles zit in de global scope. Mocht ik functies willen implementeren, zou ik er hoogstwaarschijnlijk voor opteren dit zo te houden aangezien de code een heel stuk gecompliceerder zou worden met ondersteuning voor scope.

Puke

Met het "Puke"-statement kan uitvoer naar de console, met andere woorden naar "standaard-uit" weggeschreven worden. Zowel booleaanse als aritmetische expressies, alsook sensorwaarden kunnen worden getoond. De naamgeving van dit statement is vrij voordehandliggend, maar toch is er nog een diepere betekenis. Het woord "Puke" heeft namelijk een eerder negatieve connotatie; dit in overeenstemming met het feit dat I/O over het algemeen traag en bijgevolg negatief is.

Review

Commentaar is mogelijk via het “Review” statement. Ondersteuning voor commentaar op willekeurige plaatsen in de code is niet ingebouwd. De ondersteunde vormen van commentaar zijn lijncommentaar en commentaar over meerdere lijnen. Commentaar over meerdere lijnen is een logisch resultaat van 5.2. en vereiste geen verdere aanpassing aan de code.

Drive (MBot)

Met dit commando worden de motoren van de MBot aangestuurd. De motoren individueel aansturen is niet mogelijk, wel kan er eenvoudig gestuurd worden door een richting mee te geven aan dit statement. Ondersteunde richtingen zijn “forward”, “left”, “right”, “backward”, “backwardleft”, “backwardright”. De motoren stoppen, remmen, is ook mogelijk met de richting “brake”.

Leds (MBot)

Om de LED’s van de MBot te controleren werd het “leds” statement ingevoerd. Dit statement vergt 2 argumenten, namelijk de linkse of rechtse led, gevolgd door de gewenste kleur. Deze kleur kan in RGB-waarden worden opgegeven, maar er zijn ook enkele basiskleuren ingebouwd. Deze zijn “red”, “green”, “blue”, “cyan”, “magenta”, “yellow”, “white”.

Sensoren (MBot)

De waarden lijnsensor en ultrasonische sensor kunnen worden opgevraagd met de expressies “linesensor” en “ultrason”. Deze worden gewoon als expressies gezien voor de evaluator, waardoor bijvoorbeeld het dubbel van de waarde van de lijnsensor eenvoudig kan worden berekend met

```
1  order x (2*linesensor);
```

Hoofdstuk 4

Programma's

Fibonacci

Dit programma print de Fibonacci getallen tot het afgesloten wordt. Aangezien het zo snel ging en na één enkele seconde reeds Infinity bereikte, heb ik het vertraagd met een cook-statement. COURSES/PROGRAMS/FIBONACCI.COURSE

```
order x -> 0;
order y -> 1;
puke x;
puke y;
eating (tasty) ->
  order z -> (x+y);
  order x -> y;
  order y -> z;
  puke (z);
  cook 1;
enough;
```

MBot: Politiewagen

Laat de LEDs afwisselend rood en blauw knipperen, zoals een zwaailicht van een politiewagen of een brandweerauto. Als dit te snel zou gaan zou dit niet aangenaam zijn en tevens aanleiding kunnen geven tot epilepsie, waardoor ik hier ook een vertraging heb ingebouwd. COURSES/ALGORITHMS/POLICECAR.COURSE

```
order x -> tasty;
eating (tasty) ->
  order x -> (!x);
  led left  -> (255-(255*x)), 0, (255*x);
  led right -> (255*x),       0, (255-(255*x));
enough;
```

MBot: Lijnvolger

De lijnvolger werkt volgens een zeer eenvoudig principe. Zolang beide sensoren zwart zien, rijdt de robot rechtdoor. Wanneer de linkse of rechtse sensor wit ziet, wordt respectievelijk naar rechts of links gereden. Wanneer beide sensoren wit zien en de robot dus de weg

kwijt is, wordt achteruit gereden en gedraaid naar de kant waar het het laatst wit was. COURSES/ALGORITHMS/LINEFOLLOW.COURSE

```
order prev -> linesensor;
eating (tasty) ->
  order x -> linesensor;
  hungry (x == 3) ->
    drive forward;
    order prev -> x;
  satisfied;

  hungry (x == 2) ->
    drive right;
    order prev -> x;
  satisfied;

  hungry (x == 1) ->
    drive left;
    order prev -> x;
  satisfied;

  hungry (x == 0) ->
    hungry (prev == 1) ->
      drive backwardright;
    stuffed ->
      hungry (prev == 2) ->
        drive backwardleft;
      stuffed ->
        drive backward;
      satisfied;
    satisfied;
  satisfied;
enough;
```

MBot: Obstakels ontwijken

De obstakel ontwijker is zeer intuïtief geprogrammeerd. Er wordt namelijk altijd rechtdoor gereden, tot dit niet meer mogelijk is (de ultrasone sensor detecteert een obstakel), waarna telkens naar dezelfde kant wordt gedraaid, namelijk rechts. Om zeker te zijn dat er altijd kan gedraaid worden, zal de robot eerst een stuk achteruit rijden als deze te dicht bij het obstakel zou staan. COURSES/ALGORITHMS/OBSTACLE.COURSE

```
eating (tasty) ->
  hungry (ultrason < 10) ->
    hungry (ultrason < 6) ->
      drive backward;
    stuffed ->
      drive right;
```

```
7       satisfied;
8     stuffed ->
9       drive forward;
10    satisfied;
11  enough;
```

Hoofdstuk 5

Implementatie

Alle getallen zijn Doubles

Alle getallen worden voorgesteld als Doubles. Dit zorgt ervoor dat ik nergens rekening moet houden met gehele getallen of kommagetallen, aangezien elk geheel getal kan worden voorgesteld als een kommagetal maar niet omgekeerd. LEXER.HS: 162-166.

Whitespace

Voor aanvang van het parsen wordt alle whitespace, dit zijn zowel spaties, tabs alsook newlines, verwijderd uit de code die geparset moet worden. Dit maakt het niet alleen sneller, het maakt ook de code een stuk aangenamer en duidelijker om te lezen. Bovendien vormde dit toch geen probleem aangezien strings niet ondersteund worden. LEXER.HS: 100-102.

```
1  -- |Preprocesses a file for parsing, removing all whitespace.
2  preprocess :: String -> String
3  preprocess = filter (`notElem` " \t\n\r")
```

Modulariteit

Om het project wat overzichtelijk te houden heb ik besloten niet alles in één groot bestand te plaatsen, maar het op te splitsen naar verschillende bestanden. Vooraleer ik de State monad geïmplementeerd had, had ik een simpele environment gemaakt bestaande uit een Map. Hiervoor had ik nog een bestand Environment.hs, maar dat is nu volledig weg aangezien het maar een lijn of 10 meer bevatte.

Booleans zijn Aritmetische expressies

Booleans worden bij het parsen onmiddellijk omgezet van tasty en disguisting naar respectievelijk de getallen 1 en 0. Dit naar analogie met de implementatie van booleans in C. De omgekeerde logica geldt ook, namelijk elk getal dat niet 0 is heeft booleaanse waarde tasty (true) en 0 wordt disguisting (false). Dit is geïnspireerd op de programmeertaal Python. LEXER.HS: 107-110.

```
1  -- |Tokenizes a boolean expression.
2  bool :: Parser Exp
3  bool = true <|> false where
4    true = ident "tasty" >> return (Constant 1)
```

```
5     false = ident "disguisting" >> return (Constant 0)
```

Maybe.fromJust

Op een aantal plaatsen, bijvoorbeeld bij het ophalen van variabelen uit de environment, wordt gebruik gemaakt van een Maybe constructie. Als deze Nothing zou geven, zou het programma verder niet kunnen werken, omdat bijvoorbeeld een benodigde variabele niet bestaat. Hierdoor heb ik besloten geen Maybe's te returnen, maar met de fromJust functie te werken. Deze zal het programma doen stoppen als er een Nothing teruggegeven wordt, waardoor ik zelf geen error afhandeling moet doen. Dit maakt de code ook een stuk leesbaarder. VOORBEELD – TYPES.HS: 116-118.

Tests

Voor elk stuk functionaliteit werden tests geschreven, deze zijn te vinden als .course bestanden in de map courses/tests/.

Hoofdstuk 6

Conclusie

De eerlijkheid gebied mij te vermelden dat ik nog niet alle concepten van Haskell volledig onder de knie heb. Ik ben er dan ook van overtuigd dat bepaalde zaken veel korter en efficiënter geïmplementeerd zouden kunnen worden, maar wegens projecten van andere vakken en de examens had ik hier helaas geen voldoende tijd meer voor.

Graag had ik nog ondersteuning voor strings toegevoegd. In het begin had ik hier rekening mee gehouden, maar uiteindelijk kwam ik hierdoor in de problemen en om het mezelf niet onnodig moeilijk te maken heb ik besloten dit te verwijderen.

Alle methoden zijn gedocumenteerd volgens de Haddock-stijl, documentatie gegenereerd door Haddock werd ook bijgevoegd in de docs/ map.

Gebruikte bronnen

- De slides over Monads: voor het implementeren van de Parser monad.
- De Haskell-library Text.Parsec: inspiratie voor de hulpfuncties in de Lexer.
- `https://wiki.haskell.org/Parsing_a_simple_imperative_language`: om een basisidee te krijgen over de structuur van Parsers en Evaluators.
- `http://stackoverflow.com/questions/16970431/implementing-a-language-interpreter-in-haskell`: verdere uitwerking van evalueren.
- `https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/12-State-Monad`: zeer goede uitleg over het gebruik van de State monad.
- `http://pointfree.io/`: het inkorten van code achteraf.

Hoofdstuk 7

Appendix Broncode

Evaluator.hs

```haskell
{-|
Module      : Evaluator
Description : Evaluates parsed expressions.
Copyright   : (c) Pieter De Clercq, 2016
License     : MIT
Maintainer  : piedcler.declercq@ugent.be
-}
module Evaluator(eval) where

import Control.Concurrent(threadDelay)
import Control.Monad
import Control.Monad.IO.Class
import qualified MBotPlus as Bot

import Types
import Utils

-- |Evaluates any statement.
eval :: Statement -> Environment ()
eval (Cook a) = cook a
eval (Eating cond s) = eating cond s
eval (Hungry cond ifc elsec) = hungry cond ifc elsec
eval (Order val var) = order val var
eval (Puke v) = puke v
eval Review = return ()
eval (RobotDrive d) = robotDrive d
eval (RobotLeds l c) = robotLed l c
eval (Seq s) = sq s

-- |Evaluates a binary expression.
binExpr :: BinaryOp -> Exp -> Exp -> Environment Double
binExpr Add x y = (+) <$> expr x <*> expr y;
binExpr Minus x y = (-) <$> expr x <*> expr y;
binExpr Multiply x y = (*) <$> expr x <*> expr y;
binExpr Divide x y = (/) <$> expr x <*> expr y;

-- |Evaluates a boolean expression.
```

```
38 boolExpr :: BooleanOp -> Exp -> Exp -> Environment Double
39 boolExpr And x y = do {xe <- expr x; ye <- expr y;
40                       return $ boolDouble $ xe /= 0 && ye /= 0}
41 boolExpr Or x y = do {xe <- expr x; ye <- expr y;
42                       return $ boolDouble $ xe /= 0 || ye /= 0}
43
44 -- |Evaluates a sleep statement.
45 cook :: Exp -> Environment ()
46 cook e = liftIO . threadDelay . round . (1000000 *) =<< expr e
47
48 -- |Evaluates a while loop.
49 eating :: Exp -> Statement -> Environment ()
50 eating cond task = do
51   loopcond <- expr cond
52   when (doubleBool loopcond) (eval task >> eating cond task)
53
54 -- |Evaluates any expression.
55 expr :: Exp -> Environment Double
56 expr (Constant c) = return c
57 expr (Variable v) = environmentGet v
58 expr (Binary op x y) = binExpr op x y
59 expr (Boolean op x y) = boolExpr op x y;
60 expr (Unary op x) = unaryExpr op x
61 expr (Relational op x y) = relExpr op x y;
62 expr RobotLineSensor = robotLineSensor
63 expr RobotUltrason = robotUltrason
64
65 -- |Evaluates an if/else conditional statement.
66 hungry :: Exp -> Statement -> Statement -> Environment ()
67 hungry cond i e = expr cond >>= \c -> if doubleBool c then eval
      i else eval e
68
69 -- |Evaluates an assignment statement.
70 order :: String -> Exp -> Environment ()
71 order var v = void . environmentSet var =<< expr v
72
73 -- |Evaluates a print statement.
74 puke :: Exp -> Environment ()
75 puke e = liftIO . void . print =<< expr e
76
77 -- |Evaluates a relational expression.
78 relExpr :: RelationalOp -> Exp -> Exp -> Environment Double
79 relExpr Greater x y = fmap boolDouble ((>) <$> expr x <*> expr
      y)
80 relExpr GrEquals x y = fmap boolDouble ((>=) <$> expr x <*>
      expr y)
81 relExpr Equals x y = fmap boolDouble ((==) <$> expr x <*> expr
      y)
```

```
82 relExpr Less x y = fmap boolDouble ((<) <$> expr x <*> expr y)
83 relExpr LtEquals x y = fmap boolDouble ((<=) <$> expr x <*>
       expr y)
84
85 -- |Evaluates a MBot motor command.
86 robotDrive :: Bot.Direction -> Environment ()
87 robotDrive dir = do {d <- liftIO Bot.connect; liftIO $
       Bot.motorDirection dir d;
88                  liftIO $ Bot.close d;}
89
90 -- |Evaluates a MBot LED command.
91 robotLed :: Bot.Led -> Color -> Environment ()
92 robotLed l col = do {rv <- expr r; gv <- expr g; bv <- expr b;
93                 d <- liftIO Bot.connect; liftIO $ Bot.led d l rv
       gv bv;
94                 liftIO $ Bot.close d} where (r, g, b) = col
95
96 -- |Evaluates a MBOT linesensor command.
97 robotLineSensor :: Environment Double
98 robotLineSensor = do {d <- liftIO Bot.connect; v <- liftIO $
       Bot.lineSensor d;
99                  liftIO $ Bot.close d; return v}
100
101 -- |Evaluates a MBOT ultrasonic sensor command.
102 robotUltrason :: Environment Double
103 robotUltrason = do {d <- liftIO Bot.connect; v <- liftIO $
       Bot.ultrason d;
104                  liftIO $ Bot.close d; return v}
105
106 -- |Evaluates a sequence of statements.
107 sq :: [Statement] -> Environment ()
108 sq = foldr ((>>) . eval) (return ())
109
110 -- |Evaluates a unary expression.
111 unaryExpr :: UnaryOp -> Exp -> Environment Double
112 unaryExpr Abs x = fmap abs (expr x)
113 unaryExpr Not x = expr x >>= \e -> return $ if e == 0 then 1
       else 0
```

Lexer.hs

```haskell
{-|
Module      : Lexer
Description : Contains tokenizing functions and functions for lexical analysis.
Copyright   : (c) Pieter De Clercq, 2016
License     : MIT
Maintainer  : piedcler.declercq@ugent.be
-}
module Lexer(module Lexer, module Data.Char) where

import Data.Char

import qualified MBotPlus as Bot
import Types hiding(optional)

-- | LEXICOGRAPHIC HELP FUNCTIONS | --

-- |Runs a parser between two delimiters.
between :: Char -> Char -> Parser a -> Parser a
between l r p = do { token l; ret <- p; token r; return ret}

-- |Runs a parser between two brackets { }.
brackets :: Parser a -> Parser a
brackets = between '{' '}'

{-|
  Parses one character, returning the character and the rest of the unparsed
  string.
-}
char :: Parser Char
char = Parser f where
  f [] = []
  f (c:s) = [(c,s)]

-- |Parses a digit.
digit :: Parser Char
digit = spot isDigit

-- |Parses multiple digits, returning them as a string.
digits :: Parser String
digits = some digit

-- |Parses the end of an instruction, denoted by a semicolon. Discards result.
end :: Parser ()
```

```haskell
end = void (some semicolon)

-- |Parses an identifier. Discards the result.
ident :: String -> Parser ()
ident k = void (string k)

-- |Parses an identifier followed by a semicolon. Discards the result.
keyword :: String -> Parser ()
keyword k = string k >> end >> return ()

-- |Parses a letter, returning the letter.
letter :: Parser Char
letter = spot isAlpha

-- |Parses multiple letters, returning them as a string.
letters :: Parser String
letters = some letter

-- |Parses a lowercase letter.
lower :: Parser Char
lower = spot isLower

-- |Parses a string if it is found, returning the empty string otherwise.
optional :: Parser String -> Parser String
optional a = a <|> return []

-- |Runs a parser between parentheses ( ).
parens :: Parser a -> Parser a
parens = between '(' ')'

-- |Parses a semicolon. Discards the result.
semicolon :: Parser ()
semicolon = token ';'

-- |Parses characters as long as a given condition holds true.
skipUntil :: Parser a -> Parser String
skipUntil cond = done <|> oncemore where
  done = cond >> return []
  oncemore = do { c <- char; b <- skipUntil cond; return $ c : b}

-- |Parses a character satisfying a given predicate.
spot :: (Char -> Bool) -> Parser Char
spot p = do { c <- char; guard (p c); return c}

-- |Parses a string.
```

```
89 string :: String -> Parser String
90 string = mapM (spot . (==))
91
92 -- |Parses a given character. Discards the result.
93 token :: Char -> Parser ()
94 token c = void (spot (c ==))
95
96 -- |Parses an uppercase letter.
97 upper :: Parser Char
98 upper = spot isUpper
99
100 -- |Preprocesses a file for parsing, removing all whitespace.
101 preprocess :: String -> String
102 preprocess = filter (`notElem` " \t\n\r")
103
104 -- [ TOKENIZERS ] --
105
106 -- |Tokenizes a boolean expression.
107 bool :: Parser Exp
108 bool = true <|> false where
109   true = ident "tasty" >> return (Constant 1)
110   false = ident "disguisting" >> return (Constant 0)
111
112 -- |Tokenizes a binary expression.
113 binaryExpr :: Parser Exp
114 binaryExpr = add <|> sub <|> mul <|> dvd
115             <|> booland <|> boolor <|> gteq <|> lteq <|> gt <|>
     lt <|> eq where
116   binpart = constant <|> unaryExpr <|> binaryExpr
117   bin tk = do { x <- binpart; ident tk; y <- binpart; return
     (x,y)}
118   aritexp tk op = do { (x,y) <- parens $ bin tk; return $
     Binary op x y}
119   add = aritexp "+" Add;
120   sub = aritexp "-" Minus;
121   mul = aritexp "*" Multiply;
122   dvd = aritexp "/" Divide;
123   boolexp tk op = do { (x,y) <- parens $ bin tk; return $
     Boolean op x y}
124   booland = boolexp "and" And;
125   boolor = boolexp "or" Or;
126   relexp tk op = do { (x,y) <- parens $ bin tk; return $
     Relational op x y}
127   gteq = relexp ">=" GrEquals;
128   lteq = relexp "<=" LtEquals;
129   gt = relexp ">" Greater;
130   lt = relexp "<" Less;
131   eq = relexp "==" Equals;
```

```
132
133 -- |Tokenizes a color.
134 color :: Parser Color
135 color = rgb <|> off <|> white <|> red <|> green <|> blue
136                 <|> cyan <|> yellow <|> magenta where
137   rgbpart = do{ret <- expr; token ','; return ret}
138   rgb = do {r <- rgbpart; g <- rgbpart; b <- expr; return (r,
      g, b)}
139   off = ident "off" >> return (Constant 0, Constant 0, Constant
      0)
140   red = ident "red" >> return (Constant 255, Constant 0,
      Constant 0)
141   green = ident "green" >> return (Constant 0, Constant 255,
      Constant 0)
142   blue = ident "blue" >> return (Constant 0, Constant 0,
      Constant 255)
143   cyan = ident "cyan" >> return (Constant 0, Constant 255,
      Constant 255)
144   magenta = ident "magenta" >> return (Constant 255, Constant
      0, Constant 255)
145   yellow = ident "yellow" >> return (Constant 255, Constant
      255, Constant 0)
146   white = ident "white" >> return (Constant 255, Constant 255,
      Constant 255)
147
148 -- |Tokenizes a constant/variable/robot expression.
149 constant :: Parser Exp
150 constant = parens constant
151          <|> num <|> bool <|> robotline <|> robotultrason <|>
     var where
152   num = fmap Constant number
153   var = fmap Variable (some (spot isAlphaNum))
154   robotline = ident "linesensor" >> return RobotLineSensor
155   robotultrason = ident "ultrason" >> return RobotUltrason
156
157 -- |Tokenizes any expression.
158 expr :: Parser Exp
159 expr = constant <|> unaryExpr <|> binaryExpr
160
161 -- |Tokenizes a number expression.
162 number :: Parser Double
163 number = float <|> nat where
164   float = do {s <- optional $ string "-"; n <- digits; token
      '.';
165           f <- digits; return $ read (s ++ n ++ "." ++ f)}
166   nat = do { s <- optional $ string "-"; n <- digits; return $
     read (s ++ n) }
167
```

```
168 -- |Tokenizes an MBot direction.
169 robotDirection :: Parser Bot.Direction
170 robotDirection = parens robotDirection <|> brake <|>forward <|>
      left <|> right
171                 <|> backwardleft <|> backwardright<|> backward
     where
172   forward = ident "forward" >> return Bot.DirForward
173   left = ident "left" >> return Bot.DirLeft
174   right = ident "right" >> return Bot.DirRight
175   backward = ident "backward" >> return Bot.DirBackward
176   backwardleft = ident "backwardleft" >> return
      Bot.DirBackwardLeft
177   backwardright = ident "backwardright" >> return
      Bot.DirBackwardRight
178   brake = ident "brake" >> return Bot.Brake
179
180 -- |Tokenizes an MBot LED identifier.
181 robotLed :: Parser Bot.Led
182 robotLed = left <|> right where
183   left = ident "left" >> return Bot.LeftLed
184   right = ident "right" >> return Bot.RightLed
185
186 -- |Tokenizes a unary expression.
187 unaryExpr :: Parser Exp
188 unaryExpr = parens unaryExpr <|> absval <|> notval where
189   absval = fmap (Unary Abs) (between '|' '|' expr)
190   notval = fmap (Unary Not) (token '!' >> expr)
```

Main.hs

```haskell
{-|
Module      : Main
Description : Entrypoint of the parser.
Copyright   : (c) Pieter De Clercq, 2016
License     : MIT
Maintainer  : piedcler.declercq@ugent.be
-}
module Main where

import Control.Monad.Trans.State.Lazy
import qualified Data.Map as Map
import System.Environment

import Evaluator
import Parser
import Types

-- |The main entrypoint of the parser.
main :: IO ((), EnvironmentVar)
main = do
  args <- getArgs
  if length args /= 1 then error "Usage: ./Main path_to_course.course: "
  else do
    ast <- parseFile $ head args
    runStateT (eval ast) (Map.fromList [])
```

MBotPlus.hs

```haskell
{-|
Module      : MBotPlus
Description : Provides functions to work with the MBot.
Copyright   : (c) Pieter De Clercq, 2016
License     : MIT
Maintainer  : piedcler.declercq@ugent.be
-}
module MBotPlus (module MBotPlus) where

import Data.Bits
import Data.Maybe(fromJust)
import MBot
import System.HIDAPI as HID(Device)

import Utils

-- |Driving directions of the motors.
data Direction = DirForward -- ^ Drive forward
               | DirLeft -- ^ Turn left
               | DirRight -- ^ Turn right
               | Brake -- Stop all motors
               | DirBackward -- ^ Drive backwards
               | DirBackwardLeft -- ^ Drive backwards and turn left
               | DirBackwardRight -- ^ Drive backwards and turn right
               deriving (Eq, Show)

-- |Led identifiers.
data Led = LeftLed -- ^ The left led
         | RightLed -- ^ The right led
         deriving (Eq, Ord, Show)

-- |Motor identifiers.
data Motor = LeftMotor -- ^ The left motor
           | RightMotor -- ^ The right motor
           deriving (Eq, Ord, Show)

-- |Motor command: drive backwards and turn left.
backwardsLeft :: Device -> IO ()
backwardsLeft d = do
  sendCommand d $ setMotor (motorId LeftMotor) 0 0
  sendCommand d $ setMotor (motorId RightMotor) (complement 80) (complement 0)

-- |Motor command: drive backwards and turn right.
```

```
44 backwardsRight :: Device -> IO()
45 backwardsRight d = do
46   sendCommand d $ setMotor (motorId LeftMotor) 80 0
47   sendCommand d $ setMotor (motorId RightMotor) 0 0
48
49 -- |Close the connection to the MBot.
50 close :: Device -> IO ()
51 close = closeMBot
52
53 -- |Open a connection to the MBot.
54 connect :: (IO Device)
55 connect = openMBot
56
57 -- |Sends a comment to the MBot.
58 command :: Device -> Command -> IO ()
59 command = sendCommand
60
61 -- |Retrieves a LED int by its identifier.
62 ledId :: Led -> Int
63 ledId x = 1 + index [LeftLed, RightLed] x
64
65 -- |Sets the color on a given LED.
66 led :: Device -> Led -> Double -> Double -> Double -> IO ()
67 led d l r g b = command d $ setRGB (ledId l) ri gi bi where
68   (ri, gi, bi) = (doubleInt r, doubleInt g, doubleInt b)
69
70 {-|
71   Converts the linesensor value to a Double.
72  BOTHW = 0, LEFTB = 1, RIGHTB = 2, BOTHB = 3
73 -}
74 lineDouble :: Line -> Double
75 lineDouble x = intDouble $ index [BOTHW, LEFTB, RIGHTB, BOTHB] x
76
77 -- |Gets the value of the linesensor as a Double.
78 lineSensor :: Device -> IO Double
79 lineSensor d = readLineFollower d >>= \l -> return $ lineDouble
     l
80
81 -- |Gets the appropriate motor instructions for a given
     Direction.
82 motorDirection :: Direction -> (Device -> IO())
83 motorDirection DirForward = goAhead
84 motorDirection DirLeft = goLeft
85 motorDirection DirRight = goRight
86 motorDirection DirBackward = goBackwards
87 motorDirection DirBackwardLeft = backwardsLeft
88 motorDirection DirBackwardRight = backwardsRight
89 motorDirection Brake = stop
```

```
90
91 -- |Gets the motor int by its identifier.
92 motorId :: Motor -> Int
93 motorId r = fromJust $ mapLookup [(LeftMotor, 0x9),
       (RightMotor, 0xa)] r
94
95 -- |Gets the value of the ultrasonic sensor as a Double.
96 ultrason :: Device -> IO Double
97 ultrason d = readUltraSonic d >>= \u -> return $ floatDouble u
```

Parser.hs

```
1  {-|
2  Module      : Parser
3  Description : Parses code into an Abstract Syntax Tree.
4  Copyright   : (c) Pieter De Clercq, 2016
5  License     : MIT
6  Maintainer  : piedcler.declercq@ugent.be
7  -}
8  module Parser(parseFile, parseString) where
9
10 import Lexer
11 import Types
12
13 -- |Parses a statement
14 parse :: Parser Statement
15 parse = parens parse <|> multipleStatementsParser
16
17 {-|
18   Runs a parser and creates an Abstract Syntax Tree.
19   The entire input string must be consumed.
20 -}
21 doParse :: (Show a) => Parser a -> String -> a
22 doParse m s = one [x | (x,t) <- apply m s, t == "" ] where
23   one [x] = x
24   one [] = error "Parse not completed."
25   one xs | length xs > 1 = error ("Multiple parses found:\n"
     ++ show xs)
26   one _ = error "Unknown parse error."
27
28 -- |Parses multiple statements.
29 multipleStatementsParser :: Parser Statement
30 multipleStatementsParser = do {mult <- many statementParser;
     return $ Seq mult}
31
32 -- |Parses a statement.
33 statementParser :: Parser Statement
34 statementParser = reviewParser
35                 <|> eatingParser
36                 <|> hungryParser
37                 <|> orderParser
38                 <|> pukeParser
39                 <|> cookParser
40                 <|> robotDriveParser
41                 <|> robotLedParser
42
43 -- |Parses a sleep statement.
44 cookParser :: Parser Statement
```

```
45 cookParser = do {ident "cook"; amt <- expr; end; return $ Cook
     amt}
46
47 -- |Parses a while loop statement.
48 eatingParser :: Parser Statement
49 eatingParser = do
50   ident "eating"
51   cond <- expr
52   ident "->"
53   action <- parse
54   keyword "enough"
55   return $ Eating cond action
56
57 -- |Parses a if/else conditional statement.
58 hungryParser :: Parser Statement
59 hungryParser = do
60   ident "hungry"
61   cond <- expr
62   ident "->"
63   ifClause <- parse
64   elseClause <- ifelse <|> none
65   keyword "satisfied"
66   return $ Hungry cond ifClause elseClause where
67     ifelse = do {ident "stuffed"; ident "->"; parse }
68     none = return Review
69
70 -- |Parses an assignment statement.
71 orderParser :: Parser Statement
72 orderParser = do
73   ident "order"
74   var <- many (spot isAlphaNum)
75   ident "->"
76   val <- expr
77   end
78   return $ Order var val
79
80 -- |Parses a print statement.
81 pukeParser :: Parser Statement
82 pukeParser = do {ident "puke"; var <- expr; end; return $ Puke
     var}
83
84 -- |Parses comments (ignoring them).
85 reviewParser :: Parser Statement
86 reviewParser = string "review" >> skipUntil end >> return Review
87
88 -- |Parses an MBot motor command.
89 robotDriveParser :: Parser Statement
```

```
90 robotDriveParser = do {ident "drive"; dir <- robotDirection;
      end;
91                     return $ RobotDrive dir}
92
93 -- |Parses an MBot LED command.
94 robotLedParser :: Parser Statement
95 robotLedParser = do {ident "led"; l <- robotLed; ident "->";
      col <- color; end;
96                   return $ RobotLeds l col}
97
98 -- |Builds an Abstract Syntax Tree from a String.
99 parseString :: String -> IO Statement
100 parseString code = return $ doParse parse $ preprocess code
101
102 -- |Builds an Abstract Syntax Tree from a file.
103 parseFile :: String -> IO Statement
104 parseFile file = do {code <- readFile file;
105                   return $ doParse parse $ preprocess code }
```

Types.hs

```haskell
{-|
Module      : Types
Description : Contains all datatypes and classes used in the project.
Copyright   : (c) Pieter De Clercq, 2016
License     : MIT
Maintainer  : piedcler.declercq@ugent.be
-}
module Types(module Types, module X) where

import Control.Applicative as X
import Control.Monad as X
import Control.Monad.Trans.State.Lazy as X
import qualified Data.Map as Map
import Data.Maybe

import qualified MBotPlus as Bot
import Utils

-- |The type for Parsers.
newtype Parser a = Parser (String -> [(a, String)])

-- |Applies a parser.
apply :: Parser a -- ^ The parser to apply
      -> String -- ^ The string to apply the parser on
      -> [(a, String)] -- ^ The parsed expression and the remaining string
apply (Parser f) = f

-- |The functor instance for a Parser.
instance Functor Parser where
  fmap = liftM

-- |The Applicative instance for a Parser.
instance Applicative Parser where
  pure = return
  (<*>) = ap

-- |The Monad instance for a Parser.
instance Monad Parser where
  return x = Parser (\s -> [(x, s)])
  m >>= k = Parser (\s -> [(y, u) |
                           (x, t) <- apply m s,
                           (y, u) <- apply (k x) t])

-- |The Alternative instance for a Parser.
```

```
45 instance Alternative Parser where
46   empty = mzero
47   (<|>) p q = Parser (\s ->
48                       case apply p s of
49                         [] -> apply q s
50                         res -> res)
51   some p = do { x <- p; xs <- many p; return (x:xs)}
52   many p = some p `mplus` return []
53
54 -- |The MonadPlus instance for a Parser.
55 instance MonadPlus Parser where
56   mzero = Parser (const [])
57   mplus m n = Parser (ap ((++) . apply m) (apply n))
58
59 -- |Expressions.
60 data Exp = Constant Double -- ^ A literal/constant number
61          | Variable String -- ^ A variable name
62          | Binary BinaryOp Exp Exp -- ^ A binary expression
63          | Unary UnaryOp Exp -- ^ A unary expression
64          | Boolean BooleanOp Exp Exp -- ^ A boolean expression
65          | Relational RelationalOp Exp Exp -- ^ A relational
    binary expression
66          | RobotLineSensor -- ^ A statement to read te MBot
    linesensor
67          | RobotUltrason -- ^ A statement to read the MBot
    ultrasonic sensor
68          deriving (Eq, Show)
69
70 -- |Binary operators.
71 data BinaryOp = Add -- ^ Add 2 expressions
72               | Minus -- ^ Substract 2 expressions
73               | Multiply -- ^ Multiply 2 expressions
74               | Divide -- ^ Divide 2 expressions
75              deriving (Eq, Show)
76
77 -- |Boolean operators.
78 data BooleanOp = And -- ^ Logic AND operator
79                | Or -- ^ Logic OR operator
80                deriving (Eq, Show)
81
82 -- |Colors consist of 3 expressions: red, green, blue
    respectively.
83 type Color = (Exp, Exp, Exp)
84
85 -- |Unary operators.
86 data UnaryOp = Abs -- ^ Absolute value of an expression
87              | Not -- ^ Boolean negate an expression
88             deriving (Eq, Show)
```

```
89
90 -- |Relational operators.
91 data RelationalOp = Greater -- ^ Strict greater than (>)
92                   | GrEquals -- ^ Greater than or equals (>=)
93                   | Equals -- ^ Equals (==)
94                   | LtEquals -- ^ Less than or equals (<=)
95                   | Less -- ^ Strict less than (<)
96                   deriving (Eq, Show)
97
98 -- |Statements.
99 data Statement = Cook Exp -- ^ A sleep statement
100               | Eating Exp Statement -- ^ A while statement
101               | Hungry Exp Statement Statement -- ^ A
    conditional statement
102               | Order String Exp -- ^ An assignment statement
103               | Puke Exp -- ^ A print statement
104               | Review -- ^ Comments
105               | RobotDrive Bot.Direction -- ^ MBot command:
    motors
106               | RobotLeds Bot.Led Color -- ^ MBot command: LEDs
107               | Seq [Statement] -- ^ A sequence of multiple
    statements
108               deriving Show
109
110 -- |An environment variable.
111 type EnvironmentVar = Map.Map String Double
112
113 -- |An environment; built upon a State/IO Transformer.
114 type Environment a = StateT EnvironmentVar IO a
115
116 -- |Retrieves an environment variable.
117 environmentGet :: String -> Environment Double
118 environmentGet k = fmap (fromJust . Map.lookup k) get
119
120 -- |Sets an environment variable.
121 environmentSet :: String -> Double -> Environment ()
122 environmentSet k v = put . Map.insert k v =<< get
```

Utils.hs

```haskell
{-|
Module      : Utils
Description : Provides simple utility functions.
Copyright   : (c) Pieter De Clercq, 2016
License     : MIT
Maintainer  : piedcler.declercq@ugent.be
-}
module Utils(module Utils) where

import qualified Data.List as List
import qualified Data.Maybe as Maybe
import qualified Data.Map as Map

-- |Converts a Boolean to a Double value.
boolDouble :: Bool -> Double
boolDouble True = 1
boolDouble False = 0

{-|
  Converts a Double to a Boolean value. Every number is converted to True except
  for zero.
-}
doubleBool :: Double -> Bool
doubleBool = (/= 0)

-- |Converts a Double to an Integer.
doubleInt :: Double -> Int
doubleInt = round

-- |Converts a floating point number to a Double.
floatDouble :: Float -> Double
floatDouble = realToFrac

-- |Retrieves the index of an element in a list, erroring if not found.
index :: (Eq a) => [a] -> a -> Int
index xs x = Maybe.fromJust (List.elemIndex x xs)

-- |Converts an Integer to a Double.
intDouble :: Int -> Double
intDouble x = fromIntegral x :: Double

-- |Retrieves the value corresponding to a key in a list of tuples.
mapLookup :: (Ord a) => [(a,b)] -> a -> Maybe b
```

```
44 mapLookup xs x = Map.lookup x (Map.fromList xs)
```

runtests.hs

Dit is een klein bash-script, geschreven om alle tests sequentieel uit te voeren. Dit heb ik telkens gedaan bij een aanpassing aan de code, om te garanderen dat alles bleef werken.

```
1 #!/bin/sh
2 TESTS=courses/tests/*.course
3 for t in $TESTS
4 do
5   echo -n "Testing: $t "
6   runhaskell Main.hs $t > /dev/null
7   echo ""
8 done
```