

**M 3 N U**

---

# Functioneel Programmeren

Pieter De Clercq



# Inhoudsopgave

<b>1</b>	<b>Inleiding</b>	<b>3</b>
<b>2</b>	<b>Syntaxis</b>	<b>4</b>
<b>3</b>	<b>Semantische constructies</b>	<b>5</b>
<b>4</b>	<b>Programma's</b>	<b>6</b>
4.1	Fibonacci . . . . .	6
4.2	MBot: Politiewagen . . . . .	6
4.3	MBot: Lijnvolger . . . . .	6
4.4	MBot: Obstakels ontwijken . . . . .	6
<b>5</b>	<b>Implementatie</b>	<b>7</b>
5.1	-Alle getallen zijn Doubles . . . . .	7
5.2	Whitespace . . . . .	7
5.3	Modulariteit . . . . .	7
5.4	Booleans zijn Aritmetische expressies . . . . .	7
5.5	Maybe.fromJust . . . . .	8
5.6	Tests . . . . .	8
<b>6</b>	<b>Conclusie</b>	<b>9</b>
6.1	Gebruikte bronnen . . . . .	9
<b>7</b>	<b>Appendix Broncode</b>	<b>10</b>
7.1	- Evaluator.hs . . . . .	10
7.2	- Lexer.hs . . . . .	13
7.3	- Main.hs . . . . .	18
7.4	- MBotPlus.hs . . . . .	19
7.5	- Parser.hs . . . . .	22
7.6	- Types.hs . . . . .	25
7.7	- Utils.hs . . . . .	28
7.8	- runtests.hs . . . . .	29

# Hoofdstuk 1

## Inleiding

De syntax van deze taal vertrok van 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 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" een menu.

## Hoofdstuk 2

### Syntaxis

## Hoofdstuk 3

### Semantische constructies

# 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`

### 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`

### 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`

### 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_AVOID.COURSE`

# 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](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>: 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

```
1 {-|
2 Module      : Evaluator
3 Description  : Evaluates parsed expressions.
4 Copyright   : (c) Pieter De Clercq, 2016
5 License     : MIT
6 Maintainer  : piedcler.declercq@ugent.be
7 -}
8 module Evaluator(eval) where
9
10 import Control.Concurrent(threadDelay)
11 import Control.Monad
12 import Control.Monad.IO.Class
13 import qualified MBotPlus as Bot
14
15 import Types
16 import Utils
17
18 — |Evaluates any statement.
19 eval :: Statement -> Environment ()
20 eval (Cook a) = cook a
21 eval (Eating cond s) = eating cond s
22 eval (Hungry cond ifc elsec) = hungry cond ifc elsec
23 eval (Order val var) = order val var
24 eval (Puke v) = puke v
25 eval Review = return ()
26 eval (RobotDrive d) = robotDrive d
27 eval (RobotLeds l c) = robotLed l c
28 eval (Seq s) = sq s
29
30 — |Evaluates a binary expression.
31 binExpr :: BinaryOp -> Exp -> Exp -> Environment Double
32 binExpr Add x y = (+) <$> expr x <*> expr y;
33 binExpr Minus x y = (-) <$> expr x <*> expr y;
34 binExpr Multiply x y = (*) <$> expr x <*> expr y;
35 binExpr Divide x y = (/) <$> expr x <*> expr y;
36
37 — |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
68 |   i else eval e
69 |
70 | — |Evaluates an assignment statement.
71 | order :: String -> Exp -> Environment ()
72 | order var v = void . environmentSet var =<< expr v
73 |
74 | — |Evaluates a print statement.
75 | puke :: Exp -> Environment ()
76 | puke e = liftIO . void . print =<< expr e
77 |
78 | — |Evaluates a relational expression.
79 | relExpr :: RelationalOp -> Exp -> Exp -> Environment Double
80 | relExpr Greater x y = fmap boolDouble ((>) <$> expr x <*> expr
81 |   y)
82 | relExpr GrEquals x y = fmap boolDouble ((>=) <$> expr x <*>
83 |   expr y)
84 | relExpr Equals x y = fmap boolDouble ((==) <$> expr x <*> expr
85 |   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

```
1 {-|
2 Module      : Lexer
3 Description  : Contains tokenizing functions and functions for
4               lexical analysis.
5 Copyright   : (c) Pieter De Clercq, 2016
6 License     : MIT
7 Maintainer  : piedcler.declercq@ugent.be
8 -}
9
10 module Lexer(module Lexer, module Data.Char) where
11
12 import Data.Char
13
14 import qualified MBotPlus as Bot
15 import Types hiding(optional)
16
17 — [ LEXICOGRAPHIC HELP FUNCTIONS ] —
18
19 — |Runs a parser between two delimiters.
20 between :: Char -> Char -> Parser a -> Parser a
21 between l r p = do { token l; ret <- p; token r; return ret}
22
23 — |Runs a parser between two brackets { }.
24 brackets :: Parser a -> Parser a
25 brackets = between '{' '}'
26
27 {-|
28 Parses one character, returning the character and the rest of
29 the unparsed
30 string.
31 -}
32 char :: Parser Char
33 char = Parser f where
34   f [] = []
35   f (c:s) = [(c,s)]
36
37 — |Parses a digit.
38 digit :: Parser Char
39 digit = spot isDigit
40
41 — |Parses multiple digits, returning them as a string.
42 digits :: Parser String
43 digits = some digit
44
45 — |Parses the end of an instruction, denoted by a semicolon.
46 Discards result.
47 end :: Parser ()
```

```

44 | end = void (some semicolon)
45 |
46 | — |Parses an identifier. Discards the result.
47 | ident :: String -> Parser ()
48 | ident k = void (string k)
49 |
50 | — |Parses an identifier followed by a semicolon. Discards the
    | result.
51 | keyword :: String -> Parser ()
52 | keyword k = string k >> end >> return ()
53 |
54 | — |Parses a letter, returning the letter.
55 | letter :: Parser Char
56 | letter = spot isAlpha
57 |
58 | — |Parses multiple letters, returning them as a string.
59 | letters :: Parser String
60 | letters = some letter
61 |
62 | — |Parses a lowercase letter.
63 | lower :: Parser Char
64 | lower = spot isLower
65 |
66 | — |Parses a string if it is found, returning the empty string
    | otherwise.
67 | optional :: Parser String -> Parser String
68 | optional a = a <|> return []
69 |
70 | — |Runs a parser between parentheses ( ).
71 | parens :: Parser a -> Parser a
72 | parens = between '(' ')'
73 |
74 | — |Parses a semicolon. Discards the result.
75 | semicolon :: Parser ()
76 | semicolon = token ';'
77 |
78 | — |Parses characters as long as a given condition holds true.
79 | skipUntil :: Parser a -> Parser String
80 | skipUntil cond = done <|> oncemore where
81 |   done = cond >> return []
82 |   oncemore = do { c <- char; b <- skipUntil cond; return $ c :
    |     b}
83 |
84 | — |Parses a character satisfying a given predicate.
85 | spot :: (Char -> Bool) -> Parser Char
86 | spot p = do { c <- char; guard (p c); return c}
87 |
88 | — |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 <|>
116 |               lt <|> eq where
117 |   binpart = constant <|> unaryExpr <|> binaryExpr
118 |   bin tk = do { x <- binpart; ident tk; y <- binpart; return
119 |                 (x,y) }
120 |   aritexp tk op = do { (x,y) <- parens $ bin tk; return $
121 |                         Binary op x y }
122 |   add = aritexp "+" Add;
123 |   sub = aritexp "-" Minus;
124 |   mul = aritexp "*" Multiply;
125 |   dvd = aritexp "/" Divide;
126 |   boolexp tk op = do { (x,y) <- parens $ bin tk; return $
127 |                         Boolean op x y }
128 |   booland = boolexp "and" And;
129 |   boolor = boolexp "or" Or;
130 |   relexp tk op = do { (x,y) <- parens $ bin tk; return $
131 |                         Relational op x y }
132 |   gteq = relexp ">=" GrEquals;
133 |   lteq = relexp "<=" LtEquals;
134 |   gt = relexp ">" Greater;
135 |   lt = relexp "<" Less;
136 |   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,
139         g, b)}
139     off = ident "off" >> return (Constant 0, Constant 0, Constant
140         0)
140     red = ident "red" >> return (Constant 255, Constant 0,
141         Constant 0)
141     green = ident "green" >> return (Constant 0, Constant 255,
142         Constant 0)
142     blue = ident "blue" >> return (Constant 0, Constant 0,
143         Constant 255)
143     cyan = ident "cyan" >> return (Constant 0, Constant 255,
144         Constant 255)
144     magenta = ident "magenta" >> return (Constant 255, Constant
145         0, Constant 255)
145     yellow = ident "yellow" >> return (Constant 255, Constant
146         255, Constant 0)
146     white = ident "white" >> return (Constant 255, Constant 255,
147         Constant 255)
147
148 — |Tokenizes a constant/variable/robot expression.
149 constant :: Parser Exp
150 constant = parens constant
151         <|> num <|> bool <|> robotline <|> robotultrason <|>
152         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         '.';
166         f <- digits; return $ read (s ++ n ++ "." ++ f)}
165     nat = do { s <- optional $ string "-"; n <- digits; return $
166         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

```
1 {-|
2 Module      : Main
3 Description  : Entrypoint of the parser.
4 Copyright   : (c) Pieter De Clercq, 2016
5 License     : MIT
6 Maintainer  : piedcler.declercq@ugent.be
7 -}
8 module Main where
9
10 import Control.Monad.Trans.State.Lazy
11 import qualified Data.Map as Map
12 import System.Environment
13
14 import Evaluator
15 import Parser
16 import Types
17
18 — |The main entrypoint of the parser.
19 main :: IO ((), EnvironmentVar)
20 main = do
21     args <- getArgs
22     if length args /= 1 then error "Usage: _./Main_
23         path_to_course.course:_"
24     else do
25         ast <- parseFile $ head args
26         runStateT (eval ast) (Map.fromList [])
```

## - MBotPlus.hs

```
1 {-|
2 Module      : MBotPlus
3 Description  : Provides functions to work with the MBot.
4 Copyright   : (c) Pieter De Clercq, 2016
5 License     : MIT
6 Maintainer  : piedcler.declercq@ugent.be
7 -}
8 module MBotPlus (module MBotPlus) where
9
10 import Data.Bits
11 import Data.Maybe(fromJust)
12 import MBot
13 import System.HIDAPI as HID(Device)
14
15 import Utils
16
17 -- |Driving directions of the motors.
18 data Direction = DirForward -- ^ Drive forward
19                | DirLeft -- ^ Turn left
20                | DirRight -- ^ Turn right
21                | Brake -- Stop all motors
22                | DirBackward -- ^ Drive backwards
23                | DirBackwardLeft -- ^ Drive backwards and turn
24                | DirBackwardRight -- ^ Drive backwards and turn
25                left
26                right
27                deriving (Eq, Show)
28
29 -- |Led identifiers.
30 data Led = LeftLed -- ^ The left led
31          | RightLed -- ^ The right led
32          deriving (Eq, Ord, Show)
33
34 -- |Motor identifiers.
35 data Motor = LeftMotor -- ^ The left motor
36            | RightMotor -- ^ The right motor
37            deriving (Eq, Ord, Show)
38
39 -- |Motor command: drive backwards and turn left.
40 backwardsLeft :: Device -> IO ()
41 backwardsLeft d = do
42   sendCommand d $ setMotor (motorId LeftMotor) 0 0
43   sendCommand d $ setMotor (motorId RightMotor) (complement 80)
44   (complement 0)
45
46 -- |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
80   l
81
82 — |Gets the appropriate motor instructions for a given
83   Direction.
84 motorDirection :: Direction -> (Device -> IO())
85 motorDirection DirForward = goAhead
86 motorDirection DirLeft = goLeft
87 motorDirection DirRight = goRight
88 motorDirection DirBackward = goBackwards
89 motorDirection DirBackwardLeft = backwardsLeft
90 motorDirection DirBackwardRight = backwardsRight
91 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_pares_found:\n_"
26     ++ show xs)
27   one _ = error "Unknown_parse_error."
28
29 -- |Parses multiple statements.
30 multipleStatementsParser :: Parser Statement
31 multipleStatementsParser = do {mult <- many statementParser;
32   return $ Seq mult}
33
34 -- |Parses a statement.
35 statementParser :: Parser Statement
36 statementParser = reviewParser
37   <|> eatingParser
38   <|> hungryParser
39   <|> orderParser
40   <|> pukeParser
41   <|> cookParser
42   <|> robotDriveParser
43   <|> robotLedParser
44
45 -- |Parses a sleep statement.
46 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

```
1 {-|
2 Module      : Types
3 Description  : Contains all datatypes and classes used in the
   project.
4 Copyright   : (c) Pieter De Clercq, 2016
5 License     : MIT
6 Maintainer  : piedcler.declercq@ugent.be
7 -}
8 module Types(module Types, module X) where
9
10 import Control.Applicative as X
11 import Control.Monad as X
12 import Control.Monad.Trans.State.Lazy as X
13 import qualified Data.Map as Map
14 import Data.Maybe
15
16 import qualified MBotPlus as Bot
17 import Utils
18
19 — |The type for Parsers.
20 newtype Parser a = Parser (String -> [(a, String)])
21
22 — |Applies a parser.
23 apply :: Parser a — ^ The parser to apply
24         -> String — ^ The string to apply the parser on
25         -> [(a, String)] — ^ The parsed expression and the
   remaining string
26 apply (Parser f) = f
27
28 — |The functor instance for a Parser.
29 instance Functor Parser where
30     fmap = liftM
31
32 — |The Applicative instance for a Parser.
33 instance Applicative Parser where
34     pure = return
35     (<*>) = ap
36
37 — |The Monad instance for a Parser.
38 instance Monad Parser where
39     return x = Parser (\s -> [(x, s)])
40     m >>= k = Parser (\s -> [(y, u) |
41                             (x, t) <- apply m s,
42                             (y, u) <- apply (k x) t])
43
44 — |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
66           binary expression
67         | RobotLineSensor — ^ A statement to read the MBot
68           linesensor
69         | RobotUltrason — ^ A statement to read the MBot
70           ultrasonic sensor
71         deriving (Eq, Show)
72
73 — |Binary operators.
74 data BinaryOp = Add — ^ Add 2 expressions
75               | Minus — ^ Subtract 2 expressions
76               | Multiply — ^ Multiply 2 expressions
77               | Divide — ^ Divide 2 expressions
78               deriving (Eq, Show)
79
80 — |Boolean operators.
81 data BooleanOp = And — ^ Logic AND operator
82               | Or — ^ Logic OR operator
83               deriving (Eq, Show)
84
85 — |Colors consist of 3 expressions: red, green, blue
86   respectively.
87 type Color = (Exp, Exp, Exp)
88
89 — |Unary operators.
90 data UnaryOp = Abs — ^ Absolute value of an expression
91               | Not — ^ Boolean negate an expression
92               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

```
1 {-|
2 Module      : Utils
3 Description  : Provides simple utility functions.
4 Copyright   : (c) Pieter De Clercq, 2016
5 License     : MIT
6 Maintainer  : piedcler.declercq@ugent.be
7 -}
8 module Utils(module Utils) where
9
10 import qualified Data.List as List
11 import qualified Data.Maybe as Maybe
12 import qualified Data.Map as Map
13
14 -- | Converts a Boolean to a Double value.
15 boolDouble :: Bool -> Double
16 boolDouble True = 1
17 boolDouble False = 0
18
19 {-|
20   Converts a Double to a Boolean value. Every number is
21   converted to True except
22   for zero.
23 -}
24 doubleBool :: Double -> Bool
25 doubleBool = (/= 0)
26
27 -- | Converts a Double to an Integer.
28 doubleInt :: Double -> Int
29 doubleInt = round
30
31 -- | Converts a floating point number to a Double.
32 floatDouble :: Float -> Double
33 floatDouble = realToFrac
34
35 -- | Retrieves the index of an element in a list, erroring if
36   not found.
37 index :: (Eq a) => [a] -> a -> Int
38 index xs x = Maybe.fromJust (List.elemIndex x xs)
39
40 -- | Converts an Integer to a Double.
41 intDouble :: Int -> Double
42 intDouble x = fromIntegral x :: Double
43
44 -- | Retrieves the value corresponding to a key in a list of
45   tuples.
46 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
```