Alphi

Tobiah Lissens

14-01-2017

Contents

1 Inleiding

In dit project wordt de eenvoudige programmeertaal Alphi opgesteld. Hierbij is het de bedoeling verschillende basiselementen van een imperatieve programmeertaal te implementeren: zoals toekenning, variabelen en volgorde van bewerkingen. De taal die hieronder wordt uitgewerkt heet Alphi, wat staat voor alphanumerical. Deze taal maakt enkel gebruik van alphanumerische karakters met de uitzondering dat whitespace ook is toegestaan. Eerst zal de syntax worden vastgelegd. Vervolgens wordt de semantiek vastgelegd en worden voorbeeldprogramma's gegeven. Hierna worden de implementatie-aspecten besproken. Als laatste worden nog enkele mogelijke verbeteringen of aanpassingen voorgesteld.

2 Syntax

BNF notatie van de Alphi taal.
note: Soms word de [0-9] notatie gebruikt om ranges aan te duiden.

```
Pgm ::= Stmt

Stmt ::= <Var> "Is" <Exp> "Stop"
       |"Command" <Output> <Exp> "Stop"
       |<Stmt> <Stmt>
       | "If" <Exp> "Begin" Stmt "End"
       | "While" <Exp> "Begin" Stmt

Exp    ::= <BExp>
         | <NExp>
         | "Command" <Input>

NExp   ::= <Num>
         | <NVar>
         | <NExp> "Add" <NExp>
         | <NExp> "Sub" <NExp>
         | <NExp> "Mul" <NExp>
         | <NExp> "Div" <NExp>
         | "Open" NExp "Close"

BExp   ::= <Bool>
         | <BVar>
         |  "Not" <BExp>
         | <NExp> "Gt" <NExp>
         | <NExp> "Lt" <NExp>
         | <NExp> "Eq" <NExp>
         | <BExp> "And" <BExp>
         | <BExp> "Or" <BExp>
         | "Open" BExp "Close"

Input  ::= "OpenMBot"
         | "CloseMBot"
         | "SensorR"
         | "SensorL"
         | "Ultra"

Output ::= "Print"
         | "MotorR"
         | "MotorL"
         | "Led1"
         | "Led2"

Bool   ::= "True" | "False"
Num    ::=  Int | Float
Int    ::= ["0"-"9"]+
Float  ::= <Int>"Point"<Int>
Var    ::= NVar | BVar
NVar   ::= "N"<Letter>+
BVar   ::= "B"<Letter>+
Letter ::= ["a"-"Z"]
```

3 Semantiek

qsdfqsdf

4 Programma's

Korte beschrijvingen van het programma

4.1 demo_police.alp

(zie Appendix Broncode)
Start teller.
Indien teller even zet Led1 op rood en led 2 op blauw.
Indien teller oneven zet led2 op rood en led1 op blauw.
Verhoog Teller met 1
Begin bij stap 2.

4.2 demo_police.alp

(zie Appendix Broncode)
Lees beide lichtsensoren uit.
Indien beide sensoren Zwart zien rij de robot rechtdoor.
Indien links wit ziet en rechts zwart draai alleen de linker motor.
Indien rechts wit ziet en links zwart draai alleen de rechter motor.
Indien Beide wit zien rij achteruit.
Begin terug bij stap 1.

4.3 demo_ultra.alp

(zie Appendix Broncode)
Lees Ultrasonesensor uit.
Indien afstand Groter dan 40 rij rechtdoor.
Indien afstand Kleiner dan 40 draai de linkermotor vooruit en de rechtermotor achteruit.
Begin terug bij stap 1.

5 Implementatie

Hier worden kort de interessante functies aangeraakt.

5.1 Parsen

5.1.1 Base.hs

Hier werd de Parser monad geimplementeerd. Het grootste deel van de code komt uit de slides over monads. Wel belangrijk te noteren dat de option uit alternative en de mplus uit de monadplus anders geïmplementeerd zijn. Hier volgt een omschrijving van hun implementatie.

```
Monadplus:
Zie Parser.Base line ?
mzero = gefaalde parser.
mplus p1 p2 = probeer parser 1 en ook parser2.


Alternative:
Zie Parser.Base line ?
empty = gefaalde parser.
option p1 p2 = indien parser 1 faalt probeer parser2.
```

5.1.2 Util.hs

Hier werden alle Parser functies geimplementeerd die nergens anders een plaats hadden. Twee interessante functies zijn matchStr en chainl1.

```
matchStr:
Zie Parser.Util line ?
matchStr s = match een bepaalde string en minstens een whitespace karakter
             of commentaar.


chainl1:
Zie Parser.Uitl line ?
chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainl1 p op = neemt 2 parsers binnen 1 gewone parser en 1 operator parser
               en zal de operator parser met laagste prioriteit toepassen.
```

5.1.3 NumericalParser.hs/BooleanParser.h/StatementParser.hs

Hier staan alle taal parsers. Deze brengen echter weinig nieuwigheden en combineren gewoon parsers uit base of util om aan hun speciefieke resultaat te komen.

5.2 Evalueren

Bij het evalueren wordt er gebruik gemaakt van een StateT monad transformer waarin een IO monad zit. Hierdoor kunnen we bij het evalueren statefull werken. Deze State zal de variabelen mappen naar waarden, Zodat we later de waardes van variabelen kunnen opvragen. De State zal ook return waardes van expressies teruggeven.

5.2.1 NumericEval.hs / BoolEval.hs

Het idee hierbij is te pattern matchen op de datastructuur. Op deze manier kunnen we elk geval apart behandelen. Voor functies die veel voorkomen wordt een abstractere versie aangemaakt in Evaluator.Util.hs. Een mooi voorbeeld hiervan is de functie EvalBOp zie Evaluator.Util.hs line ?. EvalBOp is een functie die 6 argumenten neemt.

1. Functie die 2 a's binnen neemt en een a teruggeeft
2. Expressie1 een expressie
3. Expressie2 een expressie
4. Evaluator1
5. Unwrapper (m a -> a) functie die een return value unwrapped
6. Constructor Wrapper constructor

Deze functie zal de 2 expressies uitrekenen. Vervolgens zal deze de return waarden daarvan uitpakken en tenslotte de functie erop toepassen en als laatste deze terug wrappen en in een MyState steken.

5.2.2 StatementEval.hs

De StatementEval werkt op dezelfde manier als NumericEval en BoolEval. Maar bij het evalueren moet in onze statemonad aan IO gedaan worden. Dit omdat we met onze robot/standaardout willen kunnen communiceren. De functies evalInput'(line ?), evalPrint (line ?) en evalRobotFunction (line ?) maken om dit te kunnen doen gebruik van liftIO.

5.3 RobotLib

5.4 Robot.Base.hs

Hier werd verder gewerkt op de gegeven library zodat er intuïtiever gewerkt kan worden met de MBot En zodat er een mooie scheiding kan blijven bestaan tussen de robotaansturing en de taal Alphi. Een Interessante functie is de move functie. Hierbij wordt een device, snelheid en motor meegegeven zodat de motor makkelijker kan aangestuurd worden. De Implementatie kan gevonden worden onder Robot.Base.hs line ? .

6 Mogelijke Verbeteringen

6.1 Taalaspecten

Het gebruik van enkel alphanumerical karakters is niet aangeraden. Het is zowel voor de leesbaarheid als voor de moeilijkheid van het parsen beter om wel gebruik te maken van speciale karakters. Verder zou het leuk zijn moest de taal werkelijk statisch zijn. In de plaats van voor elke identifier een ID te zetten zoals (Numerical) of B(Boolean). Dit kan gebeuren door tijdens het parsen ook een state van een environment bij te houden waarin elke variable gemapped zou worden op het type. Er zijn nog enkele kleine aspecten die de programmeertaal niet helemaal statisch maken zoals numerical expressions die kunnen staan als boolean in If/While Statement. Ook zou het leuk zijn moesten integers niet achterliggend worden geconverteerd naar doubles zoals nu het geval is. Als laatste Verbetering zouden multiple parameters voor bijvoorbeeld het aansturen van de leds van de robot ook handig zijn.

6.2 Programmeeraspecten

In de programmeertaal zit in de implementatie van de volgorde van bewerkingen werkelijk ook volgorde van bewerkingen. Wat ik hiermee bedoel is dat het omwisselen van 2 parsers hier er voor kan zorgen dat heel de parser niet meer werkt. Dit komt omdat er gebruik wordt gemaakt van de chain bewerking en mplus bewerking om de parsers te combineren. Het omwisselen van 2 parsers kan hierbij echter werkelijk heel de parser breken helpen. Hierdoor zou het misschien beter zijn de manier waarop de volgorde van bewerkingen gedaan wordt te herzien en robuuster te schrijven.

7 Conclusie

7.1 Algemeen

Een alphanumerical taal maken leek in het begin leuk. Dit bracht echter enkele nadelen met zich mee. Het grote nadeel hierbij is dat je geen speciale karakters hebt die kunnen instaan voor bijvoorbeeld het einde van een statement, haakjes etc. Verder wordt de taal ook enorm rap onduidelijk en onleesbaar doordat er weinig tot geen onderscheid gemaakt kan worden tussen keywords en Expressies of variabelen.

7.2 Syntax definitie

Hierbij zijn er soms onlogische samenstellingen mogelijk zoals Numerical Expression in de IfConditie of WhileStatement Hierdoor moet dit opgevangen worden tijdens het evalueren. Dit is ongewild.

7.3 Implementatie

De parseLibrary is vrij onduidelijk geschreven. Er onbreekt een mooie volgbare hierachie die bijvoorbeeld wel aanwezig is bij het evalueren. Dit komt voornamelijk doordat er geen eenduidige manier is om dingen te parsen en er op ieder moment rekening met whitespace en commentaar gehouden moet worden.

8 Appendix Broncode

8.1 AlphiExamples

8.1.1 demo_police.alp

```
commentOpen

A simple police sirene program

commentClose


Command OpenMBot Stop

Ncount Is 0 Stop
While True  Begin

  If Ncount Mod 2 Eq 0 Begin
    Command Led1 1 Stop
    Command Led2 3 Stop
  End

  If Ncount Mod 2 Eq 1 Begin
    Command Led1 3 Stop
    Command Led2 1 Stop
  End

  Ncount Is Ncount Add 1 Stop

End
Command CloseMBot Stop
```

8.1.2 demo_line.alp

```
commentOpen

A simple linefollowing program

commentClose


Command OpenMBot Stop
While True Begin

  Bleft Is Command SensorL Stop
  Bright Is Command SensorR Stop

  If Bleft And Bright Begin
    Command MotorL 70 Stop
    Command MotorR 70 Stop
  End
```

```
18
19    If Bleft And Not Open Bright Close Begin
20        Command MotorL 0 Stop
21        Command MotorR 80 Stop
22    End
23
24    If Bright And Not Open Bleft Close Begin
25        Command MotorL 80 Stop
26        Command MotorR 0  Stop
27    End
28
29    If Not Open Bright Or Bleft Close Begin
30      Command MotorL 0 Sub 60 Stop
31      Command MotorR 0 Sub 60 Stop
32    End
33  End
34  Command CloseMBot Stop
```

8.1.3 demo_ultra.alp

```
1   commentOpen
2
3   A simple wall evade program
4
5   commentClose
6
7
8   Command OpenMBot Stop
9
10  While True Begin
11    Ndistance Is Command Ultra Stop
12
13    If Ndistance Gt 40 Begin
14      Command MotorL 70 Stop
15      Command MotorR 70 Stop
16    End
17
18    If Ndistance Lt 39 Begin
19      Command MotorL 70 Stop
20      Command MotorR 0 Sub 70 Stop
21    End
22    Command Print Ndistance Stop
23  End
```

8.2 SRC

8.2.1 Main.hs

```
1  module Main where
2
3  import Data.Char
```

```haskell
import Data.Base
import Parser.Base
import Control.Monad.State
import Parser.StatementParser
import Evaluator.StatementEval
import Parser.Util
import Control.Applicative
import System.Environment


-- parses a string to a Statement
parseAll :: String -> Statement
parseAll = parseResult finalParse
        where finalParse = parseLeadingSpace >> parseStatement

-- parses and evaluates a given string
eval :: String -> IO (ReturnValue, Env ReturnValue)
eval = flip (runStateT . evalStatement . parseAll) emptyEnv

main :: IO (ReturnValue, Env ReturnValue)
main = getArgs >>= readFile . head >>= eval
```

8.2.2 Parser.Base.hs

```haskell
module Parser.Base where
import Control.Applicative
import Control.Monad
import Data.Base
import Data.List

-- Define new parser type
newtype Parser a = Parser (String -> [(a, String)])

-- Functor of a parser
instance Functor Parser where
  fmap = liftM

-- Applicative of a parser
instance Applicative Parser where
  pure  = return
  (<*>) = ap

-- monad Defenition 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 ])

-- MonadPlus Defenition parser
instance MonadPlus Parser where
  mzero     = Parser $ const []
```

```
27    mplus m n = Parser (\s -> apply m s ++ apply n s)
28
29  -- Alternative of a parser
30  instance Alternative Parser where
31     empty      = mzero
32     (<|>)      = option
33
34
35  -- Apply a parser.
36  apply :: Parser a -> String -> [(a, String)]
37  apply (Parser f) = f
38
39  -- Implement option for an alternative
40  option :: Parser a -> Parser a -> Parser a
41  option p q = Parser $ \s -> case apply p s of
42                                []  -> apply q s
43                                xs -> xs
44
45
46  -- Return the parse from a parser
47  parse :: Parser a -> String -> [(a, String)]
48  parse m s = [ (x,t) | (x,t) <- apply m s, t == "" ]
49
50  -- Return parsed value, assuming at least one successful parse
51  parseResult :: Parser a -> String -> a
52  parseResult p s          = one $ parse p s
53    where
54    one []                 = error noParse
55    one [x]                = fst x
56    one _                  = error ambiguousParse
```

8.2.3 Parser.NumericParser.hs

```
1   module Parser.NumericParser (parseNumberExp) where
2   import Control.Applicative
3   import Control.Monad
4   import Parser.Base
5   import Data.Char
6   import Parser.Util
7   import Data.Base
8
9   -- Parser for a series of digits
10  parseDigits :: Parser String
11  parseDigits = plus $ spot isDigit
12
13  -- Parser for an integer
14  parseInt :: Parser Int
15  parseInt = fmap read parseDigits
16
17  -- Parser for doubles
```

```
18 parseDouble :: Parser Double
19 parseDouble = do  x <- parseDigits;
20                   parseString floatSep;
21                   y <- parseDigits;
22                   return (read (x ++ "." ++ y) :: Double)
23
24 -- Parser for an numerical variable
25 parseNumVar :: Parser NumericExp
26 parseNumVar = token num >> fmap NVar parseAlpha
27
28 -- Parser for an numeral
29 parseNumLiteral :: Parser NumericExp
30 parseNumLiteral = parseTrailingSpace $ fmap LitInteger parseInt
31                                 `mplus` fmap LitDouble parseDouble
32
33
34 -- Function to make order of expressions easier
35 chainExp:: Parser NumericExp -> [(String, NumericBinaryOp)] -> Parser NumericExp
36 chainExp acc xs = chainl1 acc $ parseFromTuple' f xs
37                     where f (s, cons) = createP1' s BinaryNumericOp cons
38
39 -- Parser for all numerical expressions
40 parseNumberExp :: Parser NumericExp
41 parseNumberExp = foldl chainExp base orderBNumOp
42                     where base =          parseNumVar
43                                   `mplus` parseNumLiteral
44                                   `mplus` parseParens parseNumberExp
```

8.2.4 Parser.BoolParser.hs

```
1  module Parser.BoolParser (parseBoolExp) where
2  import Control.Applicative
3  import Parser.NumericParser
4  import Control.Monad
5  import Parser.Base
6  import Parser.Util
7  import Data.Base
8
9  -- Parser for all boolean expressions
10 parseBoolExp :: Parser BooleanExp
11 parseBoolExp =  base `chainl1` parseBinOPBool binaryBoolOp
12             where base =       parseLitBool
13                         `mplus` parseBoolVar
14                         `mplus` parseParens parseBoolExp
15                         `mplus` parseAltBinOPBool binaryAltBoolOp
16                         `mplus` parseUOPBool uBoolOp
17
18 -- Parser for a literal boolean
19 parseLitBool :: Parser BooleanExp
20 parseLitBool =     createP1' true LitBool True
```

```
21             `mplus` createP1' false LitBool False
22
23 -- Parser for a boolean variable
24 parseBoolVar :: Parser BooleanExp
25 parseBoolVar = token bool >> fmap BVar parseAlpha
26
27 --parser for an Unary operator
28 parseUOPBool :: [(String, UnaryBoolOp)] -> Parser BooleanExp
29 parseUOPBool     = parseFromTuple' parseU
30 parseU (s, cons) = fmap (UnaryBoolOp cons)(matchStr s >>
31                                            parseParens parseBoolExp
32                                            `mplus` parseLitBool
33                                            `mplus` parseBoolVar)
34
35
36 -- Parser for a binary boolean operator
37 parseBinOPBool :: [(String, BinaryBoolOp)] -> Parser (BooleanExp -> BooleanExp -> Boolea
38 parseBinOPBool     = parseFromTuple' parseBin
39 parseBin (s, cons) = createP1' s BinaryBoolOp cons
40
41 -- Parser for a numeral expresion boolean operator
42 parseAltBinOPBool :: [(String, BinaryAltBoolOp)] -> Parser BooleanExp
43 parseAltBinOPBool     = parseFromTuple' parseAltBin
44 parseAltBin (s, cons) = do  x <- parseNumberExp
45                            matchStr s
46                            y <- parseNumberExp
47                            return (BinaryAltBoolOp cons x y)
```

8.2.5 Parser.StatementParser.hs

```
1  module Parser.StatementParser (parseStatement) where
2  import Control.Applicative
3  import Parser.NumericParser
4  import Control.Monad
5  import Parser.Base
6  import Parser.Util
7  import Data.Base
8  import Parser.BoolParser
9
10 -- Parser for an Expression
11 parseExp :: Parser Exp
12 parseExp =        fmap BExp parseBoolExp
13           `mplus` fmap NExp parseNumberExp
14           `mplus` parseINCommand sensorL  LineLeft
15           `mplus` parseINCommand sensorR  LineRight
16           `mplus` parseINCommand ultra    ReadUltra
17           `mplus` parseINCommand openBot  OpenBotConnection
18           `mplus` parseINCommand closeBot CloseBotConnection
19
20
```

```
21 -- Parser for multiple Statements
22 parseStatementExp :: Parser Statement
23 parseStatementExp = matchEnd $ ExpStatement <$> parseExp
24
25 -- Parser to easyfy structures like while loops and if expressions
26 parseStruct :: String -> (Exp -> Statement -> Statement) -> Parser Statement
27 parseStruct s c = do matchStr s
28                      x <- parseExp
29                      y <- parseBrackets $ parseStatement `mplus` return Empty
30                      return $ c x y
31
32 -- Parser for an output command
33 parseOUTCommand :: String -> OUTCommand -> Parser Statement
34 parseOUTCommand s c = matchEnd $ fmap (Output c) (matchStr command >> matchStr s >> pars
35
36 -- Parser for an input command
37 parseINCommand :: String -> INCommand -> Parser Exp
38 parseINCommand s c = matchStr command >> matchStr s >> (return . Input) c
39
40 -- Parser for assignment of an expression
41 parseAssign :: Parser Statement
42 parseAssign = matchEnd $ assign' bool `mplus` assign' num
43       where assign' t = do token t
44                            s <- parseAlpha
45                            matchStr assign
46                            x <- parseExp
47                            return (Assign s x)
48
49 -- Parser for a Statement
50 parseStatement :: Parser Statement
51 parseStatement = base `chainl1` return Statements
52         where base = parseStatementExp
53                      `mplus` parseAssign
54                      `mplus` parseStruct       if'       If
55                      `mplus` parseStruct       while     While
56                      `mplus` parseOUTCommand   print'    Print
57                      `mplus` parseOUTCommand   motorR    MotorRight
58                      `mplus` parseOUTCommand   motorL    MotorLeft
59                      `mplus` parseOUTCommand   led1      Led1
60                      `mplus` parseOUTCommand   led2      Led2
```

8.2.6 Parser.Util.hs

```
1 module Parser.Util where
2 import Control.Applicative
3 import Control.Monad
4 import Parser.Base
5 import Data.Base
6 import Data.Char
7
```

```
8  -- match zero or more occurrences
9  star :: Parser a -> Parser [a]
10 star p = plus p `mplus` return []
11
12 -- match one or more occurrences
13 plus :: Parser a -> Parser [a]
14 plus p = do x  <- p
15             xs <- star p
16             return (x:xs)
17
18 -- parse a character satisfying a predicate (e.g., isDigit)
19 spot :: (Char -> Bool) -> Parser Char
20 spot p = do c <- char
21             guard (p c)
22             return c
23
24 -- Match a given character
25 token :: Char -> Parser Char
26 token c = spot (== c)
27
28 -- parse exactly one character
29 char :: Parser Char
30 char = Parser f
31   where
32   f ""     = []
33   f (c:s)  = [(c,s)]
34
35
36 parseString :: String -> Parser String
37 parseString ""       = return ""
38 parseString (x:xs)   = do y <- token x
39                           parseString xs
40                           return (y:xs)
41
42 parseAlpha :: Parser String
43 parseAlpha = parseTrailingSpace $ plus (spot isAlpha) >>= isKeyword
44     where isKeyword x | x `elem` keywords = mzero
45                       |  otherwise        = return x
46
47 -- Parser that ignores whiteSpace and newlines
48 parseWhiteSpace :: Parser Char
49 parseWhiteSpace = spot isSpace <|> token '\n'
50
51 parseSpace :: Parser a -> Parser a
52 parseSpace = (=<<) $ \x ->  plus parseWhiteSpace >> return x
53
54 parseSpaceAndComments :: Parser String
55 parseSpaceAndComments = parseSpace parseComments
56
```

```
57 -- Creates a new parser that ignores newLines and whitespace
58 parseTrailingSpace :: Parser a -> Parser a
59 parseTrailingSpace p = parseSpace p `mplus` do  x <- parseSpace p
60                                                 parseSpaceAndComments
61                                                 return x
62
63
64 parseLeadingSpace ::  Parser String
65 parseLeadingSpace = star parseWhiteSpace `mplus` (star parseWhiteSpace >> parseSpaceAndC
66
67
68 -- Match a certain keyword
69 matchStr :: String -> Parser String
70 matchStr = parseTrailingSpace  . parseString
71
72 -- Match parentheses
73 parseParens :: Parser a -> Parser a
74 parseParens p   = do matchStr parOpen
75                      x <- p
76                      matchStr parClosed
77                      return x
78
79 -- Match
80 parseBrackets :: Parser a -> Parser a
81 parseBrackets p = do matchStr bracketsOpen
82                      x <- p
83                      matchStr bracketsClosed
84                      return x
85
86 matchEnd :: Parser a -> Parser a
87 matchEnd p =  do x <- p
88                  matchStr stop
89                  return x
90
91 createP1 :: Parser a -> (b -> c) -> b -> Parser c
92 createP1 p c a1 = p >> (return . c) a1
93
94 createP1' :: String -> (a -> b) -> a -> Parser b
95 createP1' = createP1 . matchStr
96
97 -- parse comments
98 parseComments :: Parser String
99 parseComments = do parseString commentOpen
100                    findclose
101                where findclose = parseString commentClose
102                                  <|> (spot (const True) >> findclose)
103
104
105
```

```
106 parseFromTuple' :: (Functor t, Foldable t, MonadPlus m) => (a1 -> m a) -> t a1 -> m a
107 parseFromTuple' f xs = foldl1 mplus $ fmap f xs
108
109 -- creates a new parser from a parser and a parser of an operator
110 chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
111 chainl1 p op = p >>= rest
112   where rest a = do f <- op
113                     b <- p
114                     rest (f a b)
115                     `mplus` return a
```

8.2.7 Evaluator.NumericEval.hs

```
1  module Evaluator.NumericEval (evalNumExp) where
2  import Control.Monad.State
3  import Control.Monad
4  import Data.Maybe
5  import Parser.Base
6  import Control.Monad.Trans.Maybe
7  import Parser.NumericParser
8  import Parser.BoolParser
9  import Data.Base
10 import Parser.Util
11 import Parser.StatementParser
12 import Evaluator.Util
13
14 -- evaluate boolean expressions
15 -- Uses evalNumExp for binary operators this will evaluate both expressions
16 -- apply the given function and return a value wrapped in a MyState
17 evalNumExp :: NumericExp -> MyState
18 -- Just return the number wrapped in MyState
19 evalNumExp (LitDouble x)              = (return . Num) x
20 -- Convert to a number and wrap it in a Mystate
21 evalNumExp (BinaryNumericOp Add x y) = evalBOp (+)   x y evalNumExp getNum Num
22 evalNumExp (LitInteger x)             = (return . Num . fromIntegral) x
23 evalNumExp (BinaryNumericOp Sub x y) = evalBOp (-)   x y evalNumExp getNum Num
24 evalNumExp (BinaryNumericOp Mul x y) = evalBOp (*)   x y evalNumExp getNum Num
25 evalNumExp (BinaryNumericOp Div x y) = evalBOp (/)   x y evalNumExp getNum Num
26 evalNumExp (BinaryNumericOp Mod x y) = evalBOp mod'' x y evalNumExp getNum Num
27 --lookup the value in the environment and return it
28 evalNumExp (NVar x)                   = state $ \s -> (find x s,s)
29
30 -- modulo for doubles what a stupid hack
31 mod'' :: Double -> Double -> Double
32 mod'' x y  = fromIntegral $ mod (round x) (round y)
```

8.2.8 Evaluator.BoolEval.hs

```
1  module Evaluator.BoolEval (evalBoolExp) where
2
```

```
3  import Control.Monad.State
4  import Data.Base
5  import Evaluator.Util
6  import Evaluator.NumericEval
7
8
9  -- Evaluate boolean expressions
10 -- Uses evalNumExp for binary operators this will evaluate both expressions
11 -- apply the given function and return a value wrapped in a MyState
12 evalBoolExp :: BooleanExp -> MyState
13 -- Just return the boolean in a MyState
14 evalBoolExp (LitBool x)                              = return (Boolean x)
15 -- return the negation of the expression wrapped in MyState
16 evalBoolExp (UnaryBoolOp Not x)                     = fmap (Boolean . not . getBool) (evalE
17 evalBoolExp (BinaryBoolOp And x y)                  = evalBOp (&&) x y evalBoolExp getBool
18 evalBoolExp (BinaryBoolOp Or  x y)                  = evalBOp (||) x y evalBoolExp getBool
19 evalBoolExp (BinaryAltBoolOp GreaterThan x y)       = evalBOp (>)  x y evalNumExp getNum Bo
20 evalBoolExp (BinaryAltBoolOp SmallerThan x y)       = evalBOp (<)  x y evalNumExp getNum Bo
21 evalBoolExp (BinaryAltBoolOp Equals x y )           = evalBOp (==) x y evalNumExp getNum Bo
22 --lookup the value in the environment and return it
23 evalBoolExp (BVar x)                                = state $ \s -> (find x s, s)
```

8.2.9 Evaluator.StatementEval.hs

```
1  module Evaluator.StatementEval (evalStatement) where
2  import Data.Base
3  import Control.Monad.State
4  import Evaluator.Util
5  import Evaluator.BoolEval
6  import Evaluator.NumericEval
7  import Robot.Base
8  import System.HIDAPI (Device)
9  import MBot hiding (Command)
10
11
12 -- Evaluate expressions
13 evalExp :: Exp -> MyState
14 evalExp (BExp  e) = evalBoolExp e
15 evalExp (NExp  e) = evalNumExp  e
16 evalExp (Input c) = evalInput   c
17
18 -- Evaluate input commands
19 evalInput :: INCommand -> MyState
20 evalInput ReadUltra          = evalInput' readUltra Num
21 evalInput LineLeft           = evalInput' (readLine SensorL) Boolean
22 evalInput LineRight          = evalInput' (readLine SensorR) Boolean
23 evalInput OpenBotConnection  = fmap Dev (liftIO openMBot) >>= insert device
24 evalInput CloseBotConnection = returnDevice >>= liftIO . closeMBot . getDevice >> return
25
26 -- Evaluate statements
```

```
27 evalStatement :: Statement -> MyState
28 evalStatement (ExpStatement e)    = evalExp e
29 evalStatement (Statements s1 s2) = evalStatement s1 >> evalStatement s2
30 evalStatement (If b s)           = evalStruct b s $ evalStatement s
31 evalStatement (While b s)        = evalStruct b s $ evalStatement s >> evalStatement (W
32 evalStatement (Assign s e)       = evalAssign s e
33 evalStatement (Output t e)       = evalCommand t e
34 evalStatement Empty              = return Void
35
36 -- Evaluate output commands
37 evalCommand :: OUTCommand -> Exp -> MyState
38 evalCommand Print (BExp e)        = evalPrint (evalBoolExp e) getBool
39 evalCommand Print (NExp e)        = evalPrint (evalNumExp e)  getNum
40 evalCommand MotorRight e          = evalRobotFunction e MotorR move
41 evalCommand MotorLeft  e          = evalRobotFunction e MotorL move
42 evalCommand Data.Base.Led1 e      = evalRobotFunction e Robot.Base.Led1 led
43 evalCommand Data.Base.Led2 e      = evalRobotFunction e Robot.Base.Led2 led
44
45
46 -- Evaluate input
47 evalInput' :: (Device -> IO a) -> (a -> ReturnValue) -> MyState
48 evalInput' f c = fmap c (returnDevice >>= liftIO . f . getDevice)
49 -- Evaluate print commands
50 evalPrint :: (Show a) => MyState -> (ReturnValue -> a) -> MyState
51 evalPrint e f = e >>= (liftIO . print . f) >> return Void
52
53 -- evaluates a robot instruction
54 -- e is the expression
55 -- f is a function that returns an IO
56 evalRobotFunction :: Integral c => Exp -> t -> (c -> t -> Device -> IO a) -> StateT (Env
57 evalRobotFunction e l f = do x <- evalExp e
58                              d <- returnDevice;
59                              liftIO (f ((floor . getNum) x) l (getDevice d))
60                              return Void
61
62 -- Evaluate structures like while and if
63 evalStruct :: Exp -> Statement -> MyState -> MyState
64 evalStruct b s f = evalExp b >>= check
65   where check (Boolean True)  = f
66         check (Boolean False) = return Void
67         check x               = error impossibleState
68
69 -- Evaluate assignment
70 evalAssign :: String -> Exp -> MyState
71 evalAssign st e = evalExp e >>= insert st
```

8.2.10 Evaluator.Util.hs

```
1 module Evaluator.Util where
2 import Control.Monad.State
```

```
3  import Data.Base
4  import Data.Maybe
5  import System.HIDAPI hiding (error)
6
7  -- evaluate 2 expressions apply a function and return a MyState with the evaluated expre
8  evalBOp :: Monad m => (t3 -> t3 -> t2) -> t -> t -> (t -> m t1) -> (t1 -> t3) -> (t2 ->
9  evalBOp f x y e g c = do x' <- e x
10                          y' <- e y
11                          return (c (f (g x') (g y')))
12
13 -- find a value in a environment
14 find ::(Eq a) => a -> [(a,b)] -> b
15 find x env = fromMaybe (error varNotFound) (lookup x env)
16
17
18 insertVar :: String -> a -> Env a -> Env a
19 insertVar s a env =  (s,a):remove s env
20   where remove s  = filter (\(s1,a) -> (s1 /= s))
21
22
23 -- Insert a return value
24 insert :: String -> ReturnValue ->  MyState
25 insert st x = state $ \s -> (x, insertVar st x s)
26
27 -- Get a variable
28 getVar :: String -> Env a -> (a -> b) -> b
29 getVar s env f = f $ find s env
30
31 -- Return the device from the state
32 returnDevice :: MyState
33 returnDevice =  state $ \s -> (find device s, s)
34
35
36 getDevice :: ReturnValue -> Device
37 getDevice (Dev d) = d
38 getDevice x       = error impossibleState
39
40 -- Extract a double from a return value
41 getNum :: ReturnValue -> Double
42 getNum (Num x)    = x
43 getNum x          = error impossibleState
44
45 -- Extract a boolean from a return value
46 getBool :: ReturnValue -> Bool
47 getBool (Boolean x) = x
48 getBool x           = error impossibleState
```

8.2.11 Data.Base.hs

```
1  module Data.Base where
```

```
2  import Data.Map
3  import System.HIDAPI hiding (error)
4  import Control.Monad.State
5
6  type Var               = String                                        -- typedef for variable
7  type Env a             = [(Var, a)]                                    -- environment declarati
8  type MyState           = StateT (Env ReturnValue) IO ReturnValue -- rename this long type
9  emptyEnv               = []                                            -- create empty environn
10
11 -- variables that can be stored in the environment and returned by a mystate
12 data ReturnValue       = Num     Double
13                        | Boolean Bool
14                        | Dev     Device
15                        | Void
16
17 -- little numerical language
18 data NumericExp        = LitInteger       Int
19                        | LitDouble        Double
20                        | NVar             Var
21                        | BinaryNumericOp NumericBinaryOp NumericExp NumericExp
22                        deriving (Show, Eq)
23
24 -- operations used in the numerical language
25 data NumericBinaryOp  = Add
26                        | Sub
27                        | Mul
28                        | Div
29                        | Mod
30                        deriving (Show, Eq)
31
32 -- little boolean language based using the numerical language
33 data BooleanExp        = LitBool          Bool
34                        | BVar             Var
35                        | UnaryBoolOp      UnaryBoolOp      BooleanExp
36                        | BinaryBoolOp     BinaryBoolOp     BooleanExp BooleanExp
37                        | BinaryAltBoolOp BinaryAltBoolOp NumericExp NumericExp
38                        deriving (Show, Eq)
39
40 -- unary operator for boolean expressions
41 data UnaryBoolOp      = Not deriving (Show, Eq)
42
43 -- binary operator for boolean expressions
44 data BinaryBoolOp     = And
45                        | Or
46                        deriving (Show, Eq)
47
48 data BinaryAltBoolOp = GreaterThan
49                        | SmallerThan
50                        | Equals
```

```
51                      deriving (Show, Eq)
52
53  -- combination of multiple expressions
54  data Exp            = BExp    BooleanExp
55                      | NExp    NumericExp
56                      | Input   INCommand
57                      deriving (Show, Eq)
58
59  -- statements
60  data Statement      = Empty
61                      | Assign        Var         Exp
62                      | Statements    Statement   Statement
63                      | If            Exp         Statement
64                      | While         Exp         Statement
65                      | ExpStatement  Exp
66                      | Output        OUTCommand  Exp
67                       deriving (Show, Eq)
68
69  -- Output -> results in an action
70  data OUTCommand     = Print
71                      | MotorRight
72                      | MotorLeft
73                      | Led1
74                      | Led2
75
76                       deriving (Show, Eq)
77  -- Input -> results in an
78  data INCommand      = LineLeft
79                      | LineRight
80                      | ReadUltra
81                      | OpenBotConnection
82                      | CloseBotConnection
83                      deriving(Show, Eq)
84
85  -- keywords --
86  parOpen             = "Open"          -- eq (      --
87  parClosed           = "Close"         -- eq )      --
88  bracketsOpen        = "Begin"         -- eq {      --
89  bracketsClosed      = "End"           -- eq }      --
90  commentOpen         = "commentOpen"   -- eq /*     --
91  commentClose        = "commentClose"  -- eq */     --
92  assign              = "Is"            -- eq =      --
93  floatSep            = "Point"         -- eq .      --
94  true                = "True"          -- eq true   --
95  false               = "False"         -- eq false  --
96  while               = "While"         -- eq while  --
97  if'                 = "If"            -- eq  if    --
98  stop                = "Stop"          -- eq ;      --
99  command             = "Command"
```

```
100 print'               = "Print"
101 motorR               = "MotorR"
102 motorL               = "MotorL"
103 sensorL              = "SensorL"
104 sensorR              = "SensorR"
105 led1                 = "Led1"
106 led2                 = "Led2"
107 ultra                = "Ultra"
108 device               = "Device"
109 openBot              = "OpenMBot"
110 closeBot             = "CloseMBot"
111
112 --types--
113 num                  = 'N' -- eq double
114 bool                 = 'B' -- eq bool
115
116 -- binary numeric operators --
117 add                  = "Add" --eq +
118 sub                  = "Sub" --eq -
119 mul                  = "Mul" --eq *
120 div'                 = "Div" --eq /
121 mod'                 = "Mod" --eq %
122
123 -- unary boolean operators --
124 not'                 = "Not" -- eq !
125
126 -- binary boolean operators --
127 and'                 = "And"
128 or'                  = "Or"
129 gt                   = "Gt"
130 lt                   = "Lt"
131 eq                   = "Eq"
132
133 -- keywords that cannont be used as variables
134 keywords = [parOpen, parClosed, bracketsOpen, bracketsClosed, assign, floatSep
135             , command, print', motorR, motorL, sensorL , sensorR, ultra, true
136             , false, if', add, sub, mul, div', mod', not', and', or', gt, lt
137             , eq]
138
139 orderBNumOp          =  [[(mul,  Mul),
140                           (div', Div),
141                           (mod', Mod)],
142                          [(add, Add),
143                           (sub, Sub)]]
144
145 boolLit              = [(true, True), (false, False)]
146 uBoolOp              = [(not', Not)]
147 binaryBoolOp         = [(and', And), (or', Or)]
148 binaryAltBoolOp      = [(gt, GreaterThan), (lt, SmallerThan), (eq, Equals)]
```

```
149
150 -- errors
151 noParse             = "No parse was found!!!!!!"
152 ambiguousParse      = "Parse is ambiguous!!!!!!"
153 evalerror           = "something went wrong when evaluating"
154 varNotFound         = "var was not found"
155 impossibleState     = "state not possible"
```

8.2.12 Robot.Base.hs

```
 1 -- A more intuitive library
 2 module Robot.Base where
 3 import System.HIDAPI hiding (error)
 4 import MBot
 5 import qualified Data.Base as Data
 6 import Data.Bits
 7 import Evaluator.Util
 8 import GHC.Float
 9
10 motors     = [(MotorR, 0xa), (MotorL, 0x9)]
11 linesensor = [(SensorL, LEFTB), (SensorR, RIGHTB )]
12 leds       = [(Led1, 1), (Led2, 2)]
13 stops      = 0
14
15 data Motor      = MotorL  | MotorR  deriving (Eq)
16 data LineSensor = SensorL | SensorR deriving (Eq)
17 data Led        = Led1    | Led2    deriving (Eq)
18
19
20 -- Note I have no clue how this works !!!!!!
21 -- A more generic and intuitive implementation to control the motors --
22 -- Speed range [-255,255]
23 move :: Int -> Motor -> Device -> IO ()
24 move s m d | m == MotorR && s > 0 = move' s stops
25           | m == MotorR          = move' (complement (-s)) (complement stops)
26           | m == MotorL && s > 0 = move' (complement s) (complement stops)
27           | m == MotorL          = move' (-s) stops
28           | otherwise            = error Data.impossibleState
29         where move' s x = sendCommand d $ setMotor (find m motors) s x
30
31 {-
32 1 -> Red
33 2 -> Green
34 3 -> Blue
35 -}
36
37 led :: Int -> Led -> Device -> IO()
38 led x l d | x == 0    = f 0   0   0
39          | x == 1    = f 100 0   0
40          | x == 2    = f 0   100 0
```

```
41            | x == 3     = f 0    0    100
42            | otherwise = error Data.impossibleState
43           where f r g b =  sendCommand d $ setRGB (find l leds) b g r
44
45 --Read out the ultrasonic sensor and convert it to a IO(Double) --
46 readUltra :: Device -> IO Double
47 readUltra d = fmap float2Double (readUltraSonic d)
48
49 -- check if the sensor sees white or black --
50 readLine :: LineSensor -> Device -> IO Bool
51 readLine s d = readLineFollower d >>= match
52              where match LEFTB  = return $ find s linesensor == LEFTB
53                    match RIGHTB = return $ find s linesensor == RIGHTB
54                    match BOTHB  = return True
55                    match BOTHW  = return False
```