Functioneel Programmeren: Haskell MBot Project

Quinten Lootens

2 januari 2017

1 Introductie

De opdracht van dit project is een programmeertaal te ontwikkelen die een robot aanstuurt. Deze taal moet uitvoerbaar zijn door het gebruik van een Haskell programma dat zelf geschreven moet worden.
Mijn taal draagt de naam *KDL*, **Kill the DeadLine**, en is voornamelijk gebaseerd op *JavaScript* en *Java*. Eenvoudig gesteld is het een taal die vergelijkbaar is met een Nederlandse versie van Java. Er zijn drie programma's in *KDL* geschreven die verder in het verslag worden toegelicht: **Politie.kdl**, **VolgDeLijn.kdl** en **JarvisOntwijk.kdl**.

2 KDL Syntax

⟨*Program*⟩ ::= ⟨*Statement*⟩*

⟨*BlockStatement*⟩ ::= '{' ⟨*Statement*⟩* '}'

⟨*Statement*⟩ ::= ⟨*If-Else*⟩ | ⟨*While*⟩ | ⟨*Expression*⟩ | ⟨*Comment*⟩

⟨*Expression*⟩ ::= ⟨*Binaryop*⟩ | ⟨*Assignment*⟩ | ⟨*Apply*⟩ | ⟨*Jarvis*⟩ | ⟨*Print*⟩ + ' ;'

⟨*Value*⟩ ::= ⟨*Int*⟩ | ⟨*Double*⟩ | ⟨*String*⟩ | ⟨*Bool*⟩

⟨*Int*⟩ ::= {⟨*Digit*⟩}1+ | '-' {⟨*Digit*⟩}1+

⟨*Double*⟩ ::= ⟨*Int*⟩ '.' ⟨*Int*⟩

⟨*Char*⟩ ::= " {⟨*Letter*⟩ | ⟨*Digit*⟩ | ⟨*Underscore*⟩ | ⟨*Space*⟩}1 "

⟨*String*⟩ ::= " {⟨*Char*⟩}2+ "

⟨*Bool*⟩ ::= `WAAR` | `VALS`

⟨*Binaryop*⟩ ::= '(' ⟨*Space*⟩ ⟨*Value*⟩ ⟨*Space*⟩ ⟨*Op*⟩ ⟨*Space*⟩ ⟨*Value*⟩ ⟨*Space*⟩ ')' ∗

⟨*Op*⟩ ::= '`+`' | '`-`' | '`*`' | '`==`' | '`!=`' | '`>`' | '`>=`' | '`<`' | '`<=`' | '`&&`' | '`||`'

⟨*Assignment*⟩ ::= `laat` ⟨*Space*⟩ {⟨*Identifier*⟩ | ⟨*System Identifier*⟩} ⟨*Space*⟩ '`=`' ⟨*Space*⟩ {⟨*Expression*⟩ | ⟨*Value*⟩}

⟨*Letter*⟩ ::= ⟨*Lowercase_ letter*⟩ | ⟨*Uppercase_ letter*⟩

⟨*Underscore*⟩ ::= '_'

⟨*Lowercase_ letter*⟩ ::= a|b|c|d|...|z

⟨*Uppercase_ letter*⟩ ::= A|B|C|D|...|Z

⟨*Digit*⟩ ::= 0|1|2|3|4|5|6|7|8|9

⟨*Identifier*⟩ ::= {⟨*Letter*⟩}1 {⟨*letter*⟩ | ⟨*Underscore*⟩ | ⟨*Digit*⟩}0+

⟨*System Identifier*⟩ ::= ⟨*Underscore*⟩⟨*Identifier*⟩

⟨*Apply*⟩ ::= '(' ⟨*Expression*⟩ '(' ⟨*Expression*⟩ ')' ')'

⟨*Empty*⟩ ::= ''

⟨*Space*⟩ ::= ' '

⟨*While*⟩ ::= `TERWIJL` '(' ⟨*Expression*⟩ ')' ⟨*BlockStatement*⟩

⟨*If-Else*⟩ ::= `ALS` '(' ⟨*Expression*⟩ ')' ⟨*BlockStatement*⟩ {`ANDERS` ⟨*BlockStatement*⟩}opt

⟨*Jarvis*⟩ ::= ⟨*Sensor*⟩ | `WACHT` | `LINKS` | `RECHTS` | `VOORUIT` | `ACHTERUIT` | `LICHT1` '(' ⟨*Int*⟩, ⟨*Int*⟩, ⟨*Int*⟩ ')' | `LICHT2` '(' ⟨*Int*⟩, ⟨*Int*⟩, ⟨*Int*⟩ ')'

⟨*Sensor*⟩ ::= `laat` ⟨*Space*⟩ ⟨*Identifier*⟩ ⟨*Space*⟩ '=' ⟨*Space*⟩ {`JARVIS_LIJN` | `JARVIS_AFSTAND`}

⟨*Comment*⟩ ::= `***`⟨*Empty*⟩{⟨*Letter*⟩ | ⟨*Digit*⟩}1⟨*String*⟩

⟨*Print*⟩ ::= `PRINT` '(' ⟨*Expression*⟩ ')'

∗ : <Value> afhankelijk van de ondersteunde operatie.

3 Semantiek van KDL

Een programma bestaat uit een opeenvolging van statements. Elk statement wordt afgesloten met een puntkomma. De volledige semantiek is geïnspireerd op Java; in KDL zijn de statements evenwel in het Nederlands in plaats van het Engels.

- `Jarvis`: Alles wat over Jarvis gaat, gaat over de MBot van Makeblock.
- `Assignment`: Dit wordt gebruikt om variabelen te declareren alsook om een nieuwe waarde toe te kennen aan een variabele. Het linker lid bevat eerst "laat" en dan een naam van de variabele. De "laat" geeft aan dat het over een variabele gaat. Het rechterlid bevat een expressie.
- `LICHT1(Int, Int, Int)`: Dit is een ingebouwde functie die toelaat van de LED's op de robot te bedienen. Ze neemt drie argumenten, namelijk de kleur in RGB-waarde. LICHT2 is analoog aan LICHT1.
- `WACHT`: Jarvis stopt.
- `VOORUIT`: Dit laat Jarvis vooruit rijden.
- `ACHTERUIT`: Dit laat Jarvis achteruit rijden.
- `LINKS`: Dit draait Jarvis naar links.
- `RECHTS`: Dit draait Jarvis naar rechts.
- `JARVIS_LIJN`: De sensor van Jarvis gaat een waarde teruggeven. Deze waarde is een integer en geeft aan naar welke kant Jarvis zal moeten uitwijken.
- `JARVIS_AFSTAND`: De sensor van Jarvis gaat een waarde teruggeven. Deze is een integer en geeft de afstand tussen zichzelf en een object aan.
- `TERWIJL`: Dit herhaalt een reeks statements zolang de conditie naar `WAAR` evalueert. De hantering van dit commando staat beschreven in de vorige sectie.
- `ALS ANDERS`: Dit evalueert een Booleanse Expressie waarna het dan de betreffende StatementBlock zal uitvoeren. De hantering van dit commando staat beschreven in de vorige sectie.
- `Comment`: De commentaar van de programmeer kan geschreven worden door op een nieuwe lijn eerst *** te schrijven. Het eerste karakter na *** mag geen spatie zijn.

4 KDL Programma's

4.1 Politie.kdl

Dit is een eenvoudig programma dat de lichten van Jarvis constant laat veranderen door middel van het commando `LICHT1(0, 0, 255)` en `LICHT2(0, 0, 0)`. Door de waarden van de commando's in een `TERWIJL` lus constant te laten wisselen, simuleren we hiermee politielichten.

4.2 VolgDeLijn.kdl

Om Jarvis een zwarte lijn te laten volgen, laten we een programma lopen dat constant de sensor van Jarvis leest en verwerkt.
We gebruiken een `TERWIJL` lus met daarin `ALS ANDERS` statements. We beginnen met een variabele aan te maken die direct ook aan Jarvis zegt dat we de LINE sensor willen gebruiken. Dit doen we door middel van `JARVIS_LIJN`.
Daarna gaan we telkens na wat de output van dat commando is. Aan de hand daarvan kunnen we zeggen aan Jarvis of hij `VOORUIT`, `ACHTERUIT`, `LINKS` of `RECHTS` moet gaan.

4.3 JarvisOntwijk.kdl

Dit programma is analoog geschreven aan **VolgDeLijn.kdl**. We beginnen eerst met een variabele te declareren om zo te kunnen afwisselen in welke richting Jarvis zou moeten uitwijken.
We gebruiken een `TERWIJL` lus met daarin `ALS ANDERS` statements. We beginnen met een variabele aan te maken die direct ook aan Jarvis zegt dat we de DISTANCE sensor willen gebruiken. Dit doen we door middel van `JARVIS_AFSTAND`.
Daarna gaan we telkens na wat de output van dat commando is. Aan de hand daarvan kunnen we zeggen aan Jarvis of hij `VOORUIT`, `ACHTERUIT`, `LINKS` of `RECHTS` moet gaan.
Wanneer Jarvis naar links of naar rechts gaat, laten we de variabele die we in het begin hebben gedeclareerd, veranderen van waarde.

5 Implementatie KDL

Om de implementatie van de code toe te lichten, bespreken we chronologisch de modules. We brengen ze op zo'n manier dat ze telkens verder bouwen op het vorige.

In de module *Parser.hs* initialiseer ik een Monad Parser. Ik schrijf hier ook nog een aantal hulpfuncties.

In de module *DataParser.hs* ga ik alles van Strings, Characters, Integers, etc... parsen. Het is vooral een oplijsting van functies die tal van karakters waarnemen en Parsen zodat de taal zijn basisvorm kan krijgen. Een voorbeeld hiervan zijn de functies met **bracket** van lijn 42 tot lijn 52. Ook is er overal rekening gehouden met spaties, deze probeer ik zoveel mogelijk te negeren. In dat geval vind ik **whitespace** op lijn 54 een belangrijke functie. Deze wordt in vele andere functies toegepast.

In de module *Expressions.hs* ga ik *Expressions*, *Values* en *KDLValues* definiëren. Deze liggen aan de basis om operaties op uit te voeren dan. De functie **parseExp** op lijn 83 is eigenlijk de omvattende functie die de expressies zal Parsen.

In de module *Evaluator.hs* definieer ik een Map waar ik een String en Value aan meegeef. Kort gezegd de container van wat bijgehouden moet worden tijdens het verwerken van een programma. In de vorige paragraaf beschreef ik wat we allemaal kunnen parsen, in deze module gaan we die Expressies verwerken naar Values, naar KDLValues (Either String Value).

In de module *Statements.hs* parse ik de statements die ik in sectie 2 heb besproken. Zoals ik in *Expressions.hs* expressies parse, doe ik dit nu met statements. De functie **parseStatement** op lijn 46 parset alle basis Statements zoals we die kennen in Java en andere imperatieve programmeertalen. Op lijn 54 definieer ik **parseEXEC**; deze gaat een blok statements parsen. Dit

vind ik hier wel een belangrijke functie. De Jarvis Statements parse ik eronder op een analoge wijze.

In de module *Jarvis.hs* implementeer ik de MBot functies. Ik maak hier gebruik van threadDelay om ervoor te zorgen dat er geen overvloed van commando's naar Jarvis kunnen gestuurd worden.

In de module *RunKDL.hs* laat ik de Statements uitvoeren. Het is te vergelijken met de *Evaluator.hs* voor Expressies. De functie **run** op lijn 24 is de basis van heel de module. Ik overloop alle mogelijke Statements en verwerk die volgens hun definitie.

6 Conclusie

In de KDL programmeertaal zijn we in staat om de meest primitieve statements die gekend zijn in imperatieve programmeertalen te implementeren en uit te voeren. Eenvoudige programma's kunnen geschreven worden op een overzichtelijke manier. Omdat de taal, niet zoals de meeste programmeertalen, in het Nederlands is, komt het voor Nederlandstaligen helder en begrijpbaar over.

Om de programmeertaal verder uit te breiden zodat hij meerdere functies ondersteunt, is het belangrijk een aantal zaken in de Haskell code aan te passen. Op dit moment heb ik vrij expliciet de expressies beschreven. Ik zou deze generischer kunnen schrijven door de expressies verder te ontbinden. (Volgende code kan gevonden worden op lijn 34 in **Expressions.hs**.)

```
data Exp = Lit Value | Assign Name Exp | Variable Name | Apply Exp Exp | Assist Name Exp
         | Exp :+: Exp | Exp :-: Exp | Exp :*: Exp | Exp :=: Exp | Exp :==: Exp | Exp :/=: Exp
         | Exp :>: Exp | Exp :>=: Exp | Exp :<: Exp | Exp :<=: Exp | Exp :&&: Exp
         | Exp :||: Exp
```

Voorgaande code zou ik dan schrijven als volgt:

```
data Exp = Lit Value | BinOp Exp Op Exp | Variable Name | Assist Name Exp | Apply Exp Exp
data Op = (:+:) | (:-:) | (:*:) | (:/:) | (:&&:) | (:||:) | (:>:) | (:>=:) | (:<:)
         | (:<=:) | (:==:) | (:!=:)
```

Dit zou toelaten om bij de **Evaluator.hs** een betere manier te hanteren die dan ook meerdere KDLValues (types) kan ondersteunen. Daarnaast kunnen ook nog *Statements* toegevoegd worden, waaronder: For-loops, Case, etc...

De KDL taal is ideaal om in het Nederlands kennis te maken met een imperatieve programmeertaal.

7 Appendix broncode

Main.hs

```
---------------------------------------------------------
---------------------------------------------------------
-- MAIN KDL LANGUAGE
---------------------------------------------------------
---------------------------------------------------------
import           System.Environment
import           Statements
import           Parser
import           RunKDL
import           Control.Monad.State
import qualified Data.Map
import           Evaluator


main :: IO ()
main = do
  args <- getArgs
```

```
  let filename = head args
   file <- readFile filename
  let kdlpar = parse parseKDL file
  print kdlpar
  let kdlm = Data.Map.empty :: KDLMap
  _ <- runStateT (run kdlpar) kdlm
  return ()
```

Parser.hs

```
-------------------------------------------------------------
-------------------------------------------------------------
-- Monad Parser
-------------------------------------------------------------
-------------------------------------------------------------
module Parser where

import          Control.Applicative
import          Control.Monad

-------------------------------------------------------------
-- Initialize Parser
-------------------------------------------------------------
-- The type of parsers, create a new tpe
newtype Parser a = Parser (String -> [(a, String)])

-- Apply a parser
apply :: Parser a -> String -> [(a, String)]
apply (Parser f) = f

-- Return parsed value, assuming at least one successful parse
parse :: Parser a -> String -> a
parse m s = one [ x | (x,t) <- apply m s, t == "" ]
  where
    one [] = error "no parse"
    one [x] = x
    one xs | length xs > 1 = error "ambiguous parse"
    one _ = error "something want wrong in the parse"

-- Add Functor and Applicative Instances for making Parser a Monad
instance Functor Parser where
  fmap = liftM

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

instance MonadPlus Parser where
  mzero = Parser $ const []
  mplus m n = Parser (\s -> apply m s ++ apply n s)

instance Alternative Parser where
  (<|>) = mplus
  empty = mzero

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

-------------------------------------------------------------
```

```haskell
-- Utilities
-------------------------------------------------------
-- Match zero or more occurrences
star :: Parser a -> Parser [a]
star p = plus p `mplus` return []

-- Match one or more occurrences
plus :: Parser a -> Parser [a]
plus p = do{
  x <- p;
  xs <- star p;
  return (x:xs);
}

-- parse a sequence of n time parser p seperated by parser sep
sepByN :: Int -> Parser a -> Parser b -> Parser [a]
sepByN n p sep = do
                 x  <- p
                 xs <- replicateM (n - 1) (sep >> p)
                 return (x:xs)
```

DataParser.hs

```haskell
-------------------------------------------------------
-------------------------------------------------------
-- Parse Data (Strings, Characters, Numbers, etc)
-------------------------------------------------------
-------------------------------------------------------
module DataParser (
char, spot, token, notToken, bracket, bracket', roundBracket, whitespace,
addWhitespace, whiteToken, match, whiteMatch, whiteIdentifier , parseString ,
parseLine , checkLegalIdentifier , parseNat, parseNeg, parseDouble, parseInt
) where

import      Parser
import      Data.Char
import      Control.Monad

-------------------------------------------------------
-- Parse Strings and Characters
-------------------------------------------------------
-- Parse one character
char :: Parser Char
char = Parser f
  where
    f [] = []
    f (c:s) = [(c,s)]

-- Parse a character satisfying a predicate (e.g., isDigit)
spot :: (Char -> Bool) -> Parser Char
spot p = do
  c <- char
  guard (p c)
  return c

-- Match a given Character
token :: Char -> Parser Char
token c = spot (== c)

-- Match a Character not equal to a given Character
notToken :: Char -> Parser Char
notToken c = spot (/= c)

```

```
-- Match something between brackets
bracket :: Parser a -> Parser b -> Parser c -> Parser b
bracket o p c = o >> p >>= \x -> c >> return x

-- Match something between brackets, last parser's value is returned
bracket' :: Parser a -> Parser b -> Parser c -> Parser c
bracket' o c p = bracket o p c

-- Match something between round brackets
roundBracket :: Parser a -> Parser a
roundBracket p = bracket (whiteToken '(') p (whiteToken ')')

-- Parse as much whitespace as possible
whitespace :: Parser String
whitespace = star $ spot isSpace

-- Change a parser so it parses optional whitespace to the right
addWhitespace :: Parser a -> Parser a
addWhitespace p = p >>= \m -> whitespace >> return m

-- Parse a token with optional whitespace to the right
whiteToken :: Char -> Parser Char
whiteToken t = addWhitespace (token t)

-- Match a given String
match :: String -> Parser String
match = mapM token

-- Match a given string with optional whitespace to the right
whiteMatch :: String -> Parser String
whiteMatch s = addWhitespace (match s)

-- Match an identifier with optional whitespace to the right
 whiteIdentifier  :: Parser String
 whiteIdentifier  = addWhitespace parseString

-- Match an identifier (starts with a lowercase character and contains
-- only alphanum characters).
 parseString  :: Parser String
 parseString  = do
   x    <- spot isLower
   xs   <- star $ spot isAlphaNum
   let name = x:xs
   guard $ checkLegalIdentifier name
   return name

-- Parse until a newline character is matched
 parseLine  :: Parser String
 parseLine  = do
     c <- char;
     if c == '\n'
       then whitespace >> return [c]
       else parseLine >>= \s -> return (c:s)

-- Check if given String matches KDL Language material
 checkLegalIdentifier  :: String -> Bool
 checkLegalIdentifier  name =
   let identies = ["WAAR", "VALS", "laat", "ALS", "ANDERS", "TERWIJL"]
   in  name `notElem` identies

-------------------------------------------------------------
-- Parse Numbers
-------------------------------------------------------------
```

```haskell
-- Match a natural number
parseNat :: Parser Int
parseNat = do
  s <- plus (spot isDigit)
  return (read s)

-- Match a negative number
parseNeg :: Parser Int
parseNeg = do
  _ <- token '-'
  n <- parseNat
  return (-n)

-- Match an integer
parseInt :: Parser Int
parseInt = parseNat `mplus` parseNeg

-- Match a Double
parseDouble :: Parser Double
parseDouble = do
  n <- parseInt
  _ <- token '.'
  c <- parseNat
  return $ read (show n ++ "." ++ show c)
```

Expressions.hs

```haskell
-----------------------------------------------------------
-----------------------------------------------------------
-- Expressions
-----------------------------------------------------------
-----------------------------------------------------------
module Expressions where

import          Parser
import          DataParser
import          Control.Monad
import          System.HIDAPI

-----------------------------------------------------------
-- Define Values, KDLValues and Expressions
-----------------------------------------------------------
type Name = String
type KDLValue = Either String Value

data Value = KDLInt      Int
           | KDLBool     Bool
           | KDLFunction (Value -> KDLValue)
           | KDLString   String
           | JARVIS      Device
           | KDLDouble   Double

instance Show Value where
    show (KDLInt i)      = show i
    show (KDLBool s)     = show s
    show (KDLFunction _) = "(Function)"
    show (JARVIS _)      = "JARVIS"
    show (KDLString s)   = show s
    show (KDLDouble d)   = show d

data Exp = Lit Value
         | Assign  Name Exp
         | Variable  Name
```

```
          | Apply Exp Exp
          | Assist  Name Exp
          | Exp :+:  Exp
          | Exp :-:  Exp
          | Exp :*:  Exp
          | Exp :=:  Exp
          | Exp :==: Exp
          | Exp :/=: Exp
          | Exp :>:  Exp
          | Exp :>=: Exp
          | Exp :<:  Exp
          | Exp :<=: Exp
          | Exp :&&: Exp
          | Exp :||:  Exp
        deriving (Show)

-------------------------------------------------------
-- Parse KDLValues to Expressions
-------------------------------------------------------
-- Parse a KDLValue
parseKDLValueIntoExp :: Parser Exp
parseKDLValueIntoExp = parseKDLInt `mplus` parseKDLBool `mplus` parseString'
                       `mplus` parseDouble'

-- Parse a KDLInt Value
parseKDLInt :: Parser Exp
parseKDLInt = fmap (Lit . KDLInt) (addWhitespace parseInt)

-- Parse a KDLBool Value
parseKDLBool :: Parser Exp
parseKDLBool = do
                s <- whiteMatch "WAAR" `mplus` whiteMatch "VALS"
                return $ Lit $ KDLBool (s == "WAAR")

-- Parse a Double value
parseDouble' :: Parser Exp
parseDouble' = (Lit . KDLDouble) <$> addWhitespace DataParser.parseDouble

--Parse a string
parseString' :: Parser Exp
parseString' = bracket' (token '"') (whiteToken '"') $
               (Lit . KDLString) <$> star (notToken '"')

-------------------------------------------------------
-- Parse Expressions
-------------------------------------------------------
parseExp :: Parser Exp
parseExp = parseLit  `mplus` parseAdd   `mplus` parseMin   `mplus`
           parseMul  `mplus` parseLessS `mplus` parseLess  `mplus`
           parseGreat `mplus` parseGreatS `mplus` parseOr  `mplus`
           parseAnd  `mplus` parseAssign `mplus` parseEq   `mplus`
           parseNEq  `mplus` parseApply `mplus` parseVar
            where
    parseLit                = parseKDLValueIntoExp

    parseAdd                = do
                                (d, e) <- parseBinaryOp " + "
                                return (d :+: e)

    parseMin                = do
                                (d, e) <- parseBinaryOp " - "
                                return (d :-: e)

```

```
100     parseMul                = do
101                                 (d, e) <- parseBinaryOp " * "
102                                 return (d :*: e)
103
104     parseLessS              = do
105                                 (d, e) <- parseBinaryOp " < "
106                                 return (d :<: e)
107
108     parseLess               = do
109                                 (d, e) <- parseBinaryOp " <= "
110                                 return (d :<=: e)
111
112     parseGreat              = do
113                                 (d, e) <- parseBinaryOp " >= "
114                                 return (d :>=: e)
115
116     parseGreatS             = do
117                                 (d, e) <- parseBinaryOp " > "
118                                 return (d :>: e)
119
120     parseOr                 = do
121                                 (d, e) <- parseBinaryOp " || "
122                                 return (d  :||:  e)
123
124     parseAnd                = do
125                                 (d, e) <- parseBinaryOp " && "
126                                 return (d :&&: e)
127
128     parseAssign             = do
129                                 n <- parseName
130                                 _ <- match " = "
131                                 v <- parseExp
132                                 return (n :=: v)
133
134     parseVar                = Variable <$> whiteIdentifier
135
136     parseEq                 = do
137                                 (d, e) <- parseBinaryOp " == "
138                                 return (d :==: e)
139
140     parseNEq                = do
141                                 (d, e) <- parseBinaryOp " != "
142                                 return (d :/=: e)
143
144     parseApply              = roundBracket $ Apply
145                                 <$> parseExp
146                                 <*> roundBracket parseExp
147
148 ------------------------------------------------
149 -- Utility Parse functions
150 ------------------------------------------------
151 -- Utility function to make it cleaner
152 parseBinaryOp :: String -> Parser (Exp, Exp)
153 parseBinaryOp ope = do
154                     _ <- whiteMatch "( "
155                     f <- parseExp
156                     _ <- match ope
157                     e <- parseExp
158                     _ <- whiteMatch " )"
159                     return (f, e)
160
161 -- Parse an expression surrounded by round brackets
162 parseExpBracket :: Parser Exp
```

```
parseExpBracket = roundBracket parseExp

parseExpBracketN :: Int -> Parser [Exp]
parseExpBracketN n = roundBracket $ sepByN n parseExp (whiteToken ',')

-- Parse a variable name
parseName :: Parser Exp
parseName = fmap Variable whiteIdentifier
```

Evaluator.hs

```
-------------------------------------------------------------
-------------------------------------------------------------
-- Evaluator
-------------------------------------------------------------
-------------------------------------------------------------
module Evaluator(
evalExpression , KDLMap
) where

import qualified  Data.Map
import            Expressions

-------------------------------------------------------------
-- create KDLMap
-------------------------------------------------------------
type KDLMap = Data.Map.Map String Value

-------------------------------------------------------------
-- Evaluate Expressions
-------------------------------------------------------------
evalExpression  ::  KDLMap -> Exp -> KDLValue
evalExpression  kdlmap      (Lit  n)   = valueToKDLValue $ evalOperation kdlmap (Lit n)
evalExpression  kdlmap (Variable  n)   = searchVariable  n kdlmap
evalExpression  kdlmap (Assign n v)   =
  return $ KDLFunction (\x -> (evalExpression (Data.Map.insert n x kdlmap) v))
evalExpression  kdlmap (Assist  n v)   =
  return $ KDLFunction (\x -> evalExpression (Data.Map.insert n x kdlmap) v)
evalExpression  kdlmap (Apply  a b)   =
  mapM (evalExpression kdlmap) [a, b] >>= \[assis, arg]
        -> evalKDLFunction assis arg

evalExpression  kdlmap       expr      = case expr of
  (e :+:  f)    -> valueToKDLValue $ evalOperation kdlmap (e :+: f)
  (e :*:  f)    -> valueToKDLValue $ evalOperation kdlmap (e :*: f)
  (e :-:  f)    -> valueToKDLValue $ evalOperation kdlmap (e :-: f)
  (e :<:  f)    -> valueToKDLValue $ evalOperation kdlmap (e :<: f)
  (e :>:  f)    -> valueToKDLValue $ evalOperation kdlmap (e :>: f)
  (e :<=: f)    -> valueToKDLValue $ evalOperation kdlmap (e :<=: f)
  (e :>=: f)    -> valueToKDLValue $ evalOperation kdlmap (e :>=: f)
  (e :==: f)    -> valueToKDLValue $ evalOperation kdlmap (e :==: f)
  (e :/=: f)    -> valueToKDLValue $ evalOperation kdlmap (e :/=: f)
  (e :&&: f)    -> valueToKDLValue $ evalOperation kdlmap (e :&&: f)
  (e :||: f)    -> valueToKDLValue $ evalOperation kdlmap (e :||: f)
  _             -> Prelude.error "Something went wrong in evalExpression."


evalOperation  ::  KDLMap -> Exp -> Value
evalOperation kdlmap expr = case expr of
  (Lit  n)      -> n
  (e :+:  f)    -> intToKLDInt $  evalInt      kdlmap (e :+:  f)
  (e :*:  f)    -> intToKLDInt $  evalInt      kdlmap (e :*:  f)
  (e :-:  f)    -> intToKLDInt $  evalInt      kdlmap (e :-:  f)
```

```
  (e :<:  f)   -> boolToKDLBool $ evalIntBool kdlmap (e :<: f)
  (e :>:  f)   -> boolToKDLBool $ evalIntBool kdlmap (e :>: f)
  (e :<=: f)   -> boolToKDLBool $ evalIntBool kdlmap (e :<=: f)
  (e :>=: f)   -> boolToKDLBool $ evalIntBool kdlmap (e :>=: f)
  (e :==: f)   -> boolToKDLBool $ evalIntBool kdlmap (e :==: f)
  (e :/=: f)   -> boolToKDLBool $ evalIntBool kdlmap (e :/=: f)
  (e :&&: f)   -> boolToKDLBool $ evalBool  kdlmap (e :&&: f)
  (e :||: f)   -> boolToKDLBool $ evalBool  kdlmap (e :||: f)
  _            -> Prelude.error "Something went wrong in the evalOperation"

-------------------------------------------------------
-- Evaluate Expressions Utilities
-------------------------------------------------------
-- Evaluate Integer Arithmetic operations
evalInt :: KDLMap -> Exp -> Int
evalInt _ (Lit n)          = kdlintToInt n
evalInt k (Variable n)     = kdlintToInt $ searchVariable' n k
evalInt k (e :+: f)        = evalInt k e + evalInt k f
evalInt k (e :*: f)        = evalInt k e * evalInt k f
evalInt k (e :-: f)        = evalInt k e - evalInt k f
evalInt _ _                = Prelude.error "Unable to perform operation."

-- Evaluate Boolean operations with Integers
evalIntBool :: KDLMap -> Exp -> Bool
evalIntBool k (e :<:  f)       = evalInt k e <  evalInt k f
evalIntBool k (e :>:  f)       = evalInt k e >  evalInt k f
evalIntBool k (e :<=: f)       = evalInt k e <= evalInt k f
evalIntBool k (e :>=: f)       = evalInt k e >= evalInt k f
evalIntBool k (e :/=: f)       = evalInt k e /= evalInt k f
evalIntBool k (e :==: f)       = evalInt k e == evalInt k f
evalIntBool _ _                = Prelude.error "Unable to perform operation."

-- Evaluate Boolean operations with Booleans
evalBool :: KDLMap -> Exp -> Bool
evalBool _ (Lit n)          = kdlboolToBool n
evalBool k (Variable n)     = kdlboolToBool $ searchVariable' n k
evalBool k (e :&&: f)       = evalBool k e && evalBool k f
evalBool k (e :||: f)       = evalBool k e || evalBool k f
evalBool _ _                = Prelude.error "Unable to perform operation."


-- Evaluate a KDLFunction
evalKDLFunction :: Value -> Value -> KDLValue
evalKDLFunction (KDLFunction k) a = k a
evalKDLFunction a b = Left $ "Internal error: " ++ show a ++
                             " with argument" ++ show b ++ "."

-------------------------------------------------------
-- Utility functions
-------------------------------------------------------
kdlintToInt :: Value -> Int
kdlintToInt (KDLInt n)          = n
kdlintToInt _                   = Prelude.error $ "Internal error" ++
                                                  " KDLInt to Int"

kdlboolToBool :: Value -> Bool
kdlboolToBool (KDLBool n)       = n
kdlboolToBool _                 = Prelude.error $ "Internal error" ++
                                                  " KDLBool to Bool"

intToKLDInt :: Int -> Value
intToKLDInt                     = KDLInt

```

```
116 boolToKDLBool :: Bool -> Value
117 boolToKDLBool                = KDLBool
118
119 valueToKDLValue :: Value -> KDLValue
120 valueToKDLValue = return
121
122 -- Find a variable in the KDLMap and return a KDLValue
123 searchVariable  :: Name -> KDLMap -> KDLValue
124 searchVariable x m = case Data.Map.lookup x m of
125   Just v  -> return v
126   Nothing -> Left ("Unbound variable: " ++ x)
127
128 -- Find a variable in the KDLMap and return a Value
129 searchVariable' :: Name -> KDLMap -> Value
130 searchVariable' x m = case Data.Map.lookup x m of
131   Just v  -> v
132   _       -> Prelude.error "Something went wrong in searchVariable'"
```

Statements.hs

```
1  ------------------------------------------------------
2  ------------------------------------------------------
3  -- Parse Statements
4  ------------------------------------------------------
5  ------------------------------------------------------
6  module Statements where
7
8  import          Expressions
9  import          Parser
10 import          Control.Monad
11 import          DataParser
12 import          JarvisStmt
13
14 ------------------------------------------------------
15 -- Create Statements
16 ------------------------------------------------------
17 data Statement = EXEC Statement
18                | IF Exp Statement
19                | IFELSE Exp Statement Statement
20                | WHILE Exp Statement
21                | SEQUENCE [Statement]
22                | Name := Exp -- Variables
23                | COMMENT String
24                | NOTHING
25                | GO Direction
26                | SETLIGHT Light Exp Exp Exp
27                | STOP
28                | READ Sensor Name
29                | PRINT Exp
30                  deriving (Show)
31
32 ------------------------------------------------------
33 -- Parse Statements
34 ------------------------------------------------------
35 --Parse a KDL Program
36 parseKDL :: Parser Statement
37 parseKDL = whitespace >> parseSequence
38
39 -- Parse a SEQUENCE of Statements
40 parseSequence :: Parser Statement
41 parseSequence   = fmap SEQUENCE (star parseStatement)
42
43 ------------------------------------------------------
```

```
-- Parse Basic Statements
-----------------------------------------------------
parseStatement :: Parser Statement
parseStatement = parseAssign   `mplus` parseEXEC `mplus` parseIf      `mplus`
                 parseIfElse   `mplus` parseWhile `mplus` parseComm  `mplus`
                 parseWait     `mplus` parseLight `mplus` parseJarvis `mplus`
                 parseSensors  `mplus` parsePrint

                 where
    -- Parse a block, a sequence of Statements, wrapped in '{' '}'
    parseEXEC     =
      fmap EXEC (bracket (whiteToken '{') parseSequence (whiteToken '}'))

    -- Parse an IF Statement
    parseIf       =
      IF <$> (whiteMatch "ALS" >> parseExpBracket) <*> parseEXEC

    -- Parse an IF ELSE Statement
    parseIfElse   = do
                    (IF expr ifStmt) <- parseIf
                    _                <- whiteMatch "ANDERS"
                    elseStmt         <- parseEXEC
                    return $ IFELSE expr ifStmt elseStmt

    -- Parse a WHILE loop
    parseWhile    =
      WHILE <$> (whiteMatch "TERWIJL" >> parseExpBracket) <*> parseEXEC

    -- Parse a commentline
    parseComm     = COMMENT <$> (whiteMatch "***" >> parseLine)

    -- Parse a variable declaration
    parseAssign   =
      uncurry (:=) <$> (whiteMatch "laat" >> assignThis)
      >>= \s -> endChar >> return s
      where
        assignThis = makeT <$> whiteIdentifier
                           <*> (whiteToken '=' >> parseExp)

    -- Parse a PRINT statement
    parsePrint    = PRINT <$> (whiteMatch "PRINT" >> parseExpBracket)
                      >>= \s -> endChar >> return s

-----------------------------------------------------
-- Parse Jarvis Statements
-----------------------------------------------------
parseJarvis :: Parser Statement
parseJarvis = parseDir "VOORUIT" FORWARD `mplus`
              parseDir "ACHTERUIT" BACKWARD `mplus`
              parseDir "LINKS"    ToLEFT  `mplus`
              parseDir "RECHTS"   ToRIGHT


parseLight :: Parser Statement
parseLight =  parseL "LICHT1" LIGHT1 `mplus`
              parseL "LICHT2" LIGHT2

parseWait :: Parser Statement
parseWait =  do
             _ <- whiteMatch "WACHT"
             return STOP >>= \s -> endChar >> return s

parseSensors :: Parser Statement
```

```
parseSensors = parseSensor "JARVIS_LIJN" LINE
               `mplus`
               parseSensor "JARVIS_AFSTAND" DISTANCE

-------------------------------------------------
-- Utility functions
-------------------------------------------------
endChar :: Parser Char
endChar = addWhitespace (token ';')

makeT :: a -> b -> (a, b)
makeT a b = (a, b)

parseL :: Name -> Light -> Parser Statement
parseL name light = do
                    _          <- whiteMatch name
                    [r, g, b] <- parseExpBracketN 3
                    return (SETLIGHT light r g b) >>= \s -> endChar >> return s

parseSensor :: Name -> Sensor -> Parser Statement
parseSensor n s = do
        name <- whiteMatch "laat" >> whiteIdentifier
        _    <- whiteToken '='    >> whiteMatch n
        return (READ s name) >>= \l -> endChar >> return l

parseDir :: Name -> Direction -> Parser Statement
parseDir name dir = do
  _          <- whiteMatch name
  return (GO dir) >>= \s -> endChar >> return s
```

JarvisStmt.hs

```
-------------------------------------------------
-------------------------------------------------
-- Jarvis (MBot) Statements
-------------------------------------------------
-------------------------------------------------
module JarvisStmt(
Light(LIGHT1, LIGHT2), Sensor(LINE, DISTANCE),
Direction(FORWARD, BACKWARD, ToLEFT, ToRIGHT)
) where

data Light     = LIGHT1
               | LIGHT2
               deriving (Show)

data Sensor    = LINE
               | DISTANCE
               deriving (Show)

data Direction = FORWARD
               | BACKWARD
               | ToLEFT
               | ToRIGHT
                deriving (Show)
```

Jarvis.hs

```
-------------------------------------------------
-------------------------------------------------
-- JARVIS Engine Commands
-------------------------------------------------
```

```
5  ------------------------------------------------------------
6  module Jarvis (
7  getDistance, getLineData, leftLight , rightLight , goForward, goBackward,
8  turnRight, turnLeft, stopMotor
9  ) where
10
11 import qualified MBot as JARVIS
12 import System.HIDAPI
13 import Control.Concurrent
14
15 ------------------------------------------------------------
16 -- JARVIS Sensors
17 ------------------------------------------------------------
18 -- Get the distance using the ultrasonic sensor
19 getDistance :: Device -> IO Int
20 getDistance d = do
21   f <- JARVIS.readUltraSonic d;
22   return $ truncate f
23
24 -- Read the line linesensor
25 getLineData :: Device -> IO Int
26 getLineData d = do
27     status <- JARVIS.readLineFollower d
28     case status of
29       JARVIS.LEFTB -> return 1
30       JARVIS.RIGHTB -> return 2
31       JARVIS.BOTHB -> return 3
32       JARVIS.BOTHW -> return 4
33
34 ------------------------------------------------------------
35 -- JARVIS Lights
36 ------------------------------------------------------------
37 -- Set the left led color using rgb notation
38 leftLight :: Int -> Int -> Int -> Device -> IO ()
39 leftLight r g b d = do
40   threadDelay 500000;
41   setLed 1 r g b d;
42
43 -- Set the right led color using rgb notation
44 rightLight :: Int -> Int -> Int -> Device -> IO ()
45 rightLight r g b d = do
46   threadDelay 500000;
47   setLed 2 r g b d;
48
49 -- Set the led (index) color using rgb notation
50 setLed :: Int -> Int -> Int -> Int -> Device -> IO ()
51 setLed index r g b d = JARVIS.sendCommand d $ JARVIS.setRGB index r g b
52
53 ------------------------------------------------------------
54 -- JARVIS Driving Skills
55 ------------------------------------------------------------
56 -- Move the motor forward at the given speed
57 goForward :: Device -> IO ()
58 goForward d = do
59   threadDelay 1000000;
60   JARVIS.goBackwards d;
61
62 -- Move the motor backward at the given speed
63 goBackward :: Device -> IO ()
64 goBackward d = do
65   threadDelay 1000000;
66   JARVIS.goAhead d;
67
```

```
-- Turn right at the given speed
turnRight :: Device -> IO ()
turnRight d = do
  threadDelay 1000000;
  JARVIS.goRight d;
  threadDelay 3000000;
  JARVIS.stop d;

-- Turn left at the given speed
turnLeft :: Device -> IO ()
turnLeft d = do
  threadDelay 1000000;
  JARVIS.goLeft d;
  threadDelay 3000000;
  JARVIS.stop d;

-- Stop the motor
stopMotor :: Device -> IO ()
stopMotor d = do
  threadDelay 500000;
  JARVIS.stop d
```

RunKDL.hs

```
-----------------------------------------------------------
-----------------------------------------------------------
-- Run KDL Language
-----------------------------------------------------------
-----------------------------------------------------------
module RunKDL(
run
) where

import              Expressions
import qualified    Data.Map
import              Statements
import              Control.Monad.Trans.State
import              Control.Monad.Trans.Class
import              Jarvis
import              Evaluator
import              MBot
import              JarvisStmt
import              System.HIDAPI

-----------------------------------------------------------
-- Run KDL
-----------------------------------------------------------
run :: Statement -> StateT KDLMap IO()

-----------------------------------------------------------
-- Run JARVIS Statements
-----------------------------------------------------------
run (GO dir) =
   lift $ do jar <- openMBot
             moveJarvis dir jar
             closeMBot jar
               where moveJarvis :: Direction -> Device -> IO()
                     moveJarvis FORWARD d = goForward  d
                     moveJarvis BACKWARD d = goBackward d
                     moveJarvis ToLEFT d   = goLeft     d
                     moveJarvis ToRIGHT d  = goRight    d

run (SETLIGHT light ex1 ex2 ex3) = do
```

```
  kdlm       <- get
  jarvis     <- getJarvis
  let    values = mapM (evalExpression kdlm) [ex1, ex2, ex3]
  case  values of
    Right [KDLInt r, KDLInt g, KDLInt b] -> lift $ setLight  light  r g b  jarvis
    Right _ -> raiseError "Invalid types in light-statement"
    Left  s -> raiseError s
  where setLight :: Light -> Int -> Int -> Int -> Device -> IO ()
        setLight  LIGHT1 = leftLight
        setLight  LIGHT2 = rightLight

run STOP                          = do
                     jarvis   <- getJarvis
                     lift  $ stopMotor  jarvis

run (READ LINE n)                 = do
                     jarvis   <- getJarvis
                     kdlm     <- get
                     line     <- lift $ getLineData  jarvis
                     put $ Data.Map.insert n (KDLInt line) kdlm

run (READ DISTANCE n)             = do
                     jarvis   <- getJarvis
                     kdlm     <- get
                     dist     <- lift $ getDistance  jarvis
                     put $ Data.Map.insert n (KDLInt dist) kdlm

-------------------------------------------------------
-- Run KDL Statements
-------------------------------------------------------
run (SEQUENCE statements)       = sequence_ $ fmap run statements

run NOTHING                     = skip

run (COMMENT _)                 = run NOTHING

run (n := expression)           = do
        kdlm <- get
        let value = evalExpression  kdlm expression  in
              case value of
                 Right a -> put $ Data.Map.insert n a kdlm
                 Left  s -> raiseError s

run (IF  expression  statement)     = run $ IFELSE expression statement NOTHING

run (IFELSE expression ifStatement elseStatement) = do
        kdlm <- get
        let    predicate  = evalExpression  kdlm expression
        case  predicate  of
          Right (KDLBool True) -> run ifStatement
          Right (KDLBool False) -> run elseStatement
          Left  s               -> raiseError s
          _                     -> raiseError "Invalid  type in  if-statement"

run (WHILE expression statement) = do
  kdlm <- get
  let    predicate  = evalExpression  kdlm expression
  case  predicate  of
    Right (KDLBool True) -> run statement >> run (WHILE expression statement)
    Right (KDLBool False) -> skip
    Left  s               -> raiseError s
    Right a               -> raiseError ("Invalid  type in  while-statement: "
                               ++ show a)
```

```
103
104 run (PRINT expression)              = do
105             kdlm <- get
106             let value = evalExpression kdlm expression
107              lift $ putStrLn (case value of
108                Right a -> show a
109                Left  s -> s)
110
111 run (EXEC statement)                 = do
112             startKDLM <- get
113             run statement
114             endKDLM   <- get
115             let changed = Data.Map.intersection endKDLM startKDLM
116             let globals = Data.Map.filterWithKey (\k _ -> isJARVIS k) endKDLM
117             let newKDLM = changed `Data.Map.union` startKDLM `Data.Map.union` globals
118             put newKDLM
119
120 ---------------------------------------------------------------
121 -- Utilities
122 ---------------------------------------------------------------
123 getJarvis :: StateT KDLMap IO Device
124 getJarvis = do
125   kdlm <- get
126   case Data.Map.lookup "_JARVIS" kdlm of
127     Just (JARVIS m) -> return m
128     _               -> do
129       jar <- lift openMBot
130       put $ Data.Map.insert "_JARVIS" (JARVIS jar) kdlm
131       return jar
132
133 --Show an error message
134 raiseError :: String -> StateT KDLMap IO ()
135 raiseError = lift . fail . ("RUNTIME ERROR: " ++)
136
137 --Do nothing
138 skip :: StateT KDLMap IO ()
139 skip =  lift $ return ()
140
141 --Test if a variable name represents a global variable
142 isJARVIS :: String -> Bool
143 isJARVIS s = head s == '_'
```