
Master of Science in Computer Science Engineering

Academic year 2015–2016

# Functionele en Logische Programmeertalen
## Project MBot

Titouan Vervack

# 1 Inleiding

Voor dit project is een minimale programmeertaal ontworpen voor het besturen van een robot. De taal kreeg de naam T21 (lees als: Tee-Two-One). De inspiratie voor de taal komt uit de programmeertalen Java, Pascal, Bash en C.
Naast de taal, werd er ook een interpreter voor de taal geschreven. Deze laat toe van een programma, geschreven in T21, uit te voeren. De volledige broncode van de interpreter vindt u terug in Sectie 7.
Tenslotte zijn er een aantal voorbeeld programma's geschreven in T21. In deze programma's laat de robot zijn LED lichtjes flikkeren als een politiewagen, probeert hij obstakels te ontwijken of tracht hij een donkere lijn te volgen.

# 2 Syntax van de taal

Zodat de code proper oogt, is het toegelaten om witruimte aan het begin van een statement te plaatsen. Er zijn voor deze reden ook vaak verplichte spaties, bijvoorbeeld voor en na een binaire operator.
Om op het zicht te verstaan wat een statement doet moeten keywords in hoofdletters staan, variabelen beginnen met een kleine letter en beginnen functies met een hoofdletter. De syntax van T21 wordt gegeven door de volgende EBNF.
Hierbij is `string` een string van eender welke karakters, uitgesloten `--` en `\n` (alles dat matched met `not . isControl`).
De definitie `WS` bevat alle Unicode space karakters en de controle karakters '\t', '\n', '\r', '\f' en '\v' (alles wat matched met `Data.Char.isSpace`).

⟨*program*⟩ ::= ⟨*statement*⟩*

⟨*statement*⟩ ::= ⟨*WS*⟩* ⟨*stat*⟩ ⟨*NL*⟩
| ⟨*comment*⟩
| ⟨*empty*⟩

⟨*stat*⟩ ::= ⟨*assignment*⟩ | ⟨*rgb*⟩ | ⟨*wait*⟩ | ⟨*walk*⟩ | ⟨*moonwalk*⟩
| ⟨*hammertime*⟩| ⟨*turnleft*⟩ | ⟨*turnright*⟩ | ⟨*while*⟩ | ⟨*if*⟩

⟨*exp*⟩ ::= '`Feel`' | '`Look`' | ⟨*literal*⟩ | ⟨*name*⟩
| exp ⟨*binop*⟩ exp
| '`(`' exp '`)`'

⟨*literal*⟩ ::= ⟨*bliteral*⟩ | [0-9]+

⟨*bliteral*⟩ ::= '`True`' | '`False`'

⟨*name*⟩ ::= [a-z][a-zA-Z0-9]*

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

⟨*assignment*⟩ ::= ⟨*name*⟩ ⟨*S*⟩ '`=`' ⟨*S*⟩ exp

⟨*rgb*⟩ ::= '`Rgb`' '`(`' exp ⟨*C*⟩ exp ⟨*C*⟩ exp ⟨*C*⟩ exp ⟨*C*⟩ '`)`'

⟨*wait*⟩ ::= '`Wait`' '`(`' exp '`)`'

⟨*walk*⟩ ::= '`Walk`'

⟨*moonwalk*⟩ ::= '`Moonwalk`'

⟨*hammertime*⟩ ::= '`Hammertime`'

⟨*turnleft*⟩ ::= '`TurnLeft`'

⟨*turnright*⟩ ::= '`TurnRight`'

⟨*comment*⟩ ::= '`--`' ⟨*S*⟩ ⟨*string*⟩

⟨*while*⟩ ::= '`WHILE`' ⟨*S*⟩ ⟨*exp*⟩ '`ELIHW`'

⟨*if*⟩ ::= IF ⟨*S*⟩ ⟨*exp*⟩ '`FI`'

⟨*NL*⟩ ::= '`\r`'? '`\n`'

⟨*C*⟩ ::= '`,`' ⟨*S*⟩

⟨*S*⟩ ::= ' '

## 3 Semantiek van de taal

Een programma bestaat uit een opeenvolging van `statement`s. Deze kunnen beginnen met witruimte en hebben geen puntkomma aan het einde. Er kan ook slechts één statement per lijn staan. Hieronder worden alle statements individueel uitgelegd.

- `assignment`: Dit wordt gebruikt om variabelen te declareren alsook om een nieuwe waarde toe te kennen aan een variabele. Het linker lid bevat enkel een naam, het rechterlid bevat een expressie.
- `rgb`: Dit is een ingebouwde functie die toe laat van de LED's op de robot te bedienen. Ze neemt vier argumenten, de index van de LED en de kleur in RGB.

- `wait`: Dit laat toe om het programma te onderbreken gedurende een gegeven aantal milliseconden. Dit wordt gebruikt om te bepalen hoe lang bepaalde acties moeten uitgevoerd worden. Om gedurende 500 milliseconden te draaien voert u bijvoorbeeld de volgende reeks statements uit: `TurnLeft`; `Wait(500)`; `Hammertime`.
- `walk`: Dit laat de robot vooruit rijden aan een vooraf ingestelde snelheid.
- `moonwalk`: Dit laat de robot achteruit rijden aan dezelfde snelheid als hij vooruit rijdt.
- `hammertime`: Dit stopt beide motoren.
- `turnleft`: Dit draait de robot naar links.
- `turnright`: Dit draait de robot naar rechts.
- `while`: Dit herhaalt een reeks statements zolang de conditie naar `True` evalueert. Het statement begint met een `WHILE` gevolgd door een spatie en een expressie en ze eindigt met een `ELIHW`.
- `if`: Dit voert het blok tussen IF en ELSE uit indien de conditie evalueert naar `True`. In het andere geval wordt het blok tussen ELSE en FI uitgevoerd. Zowel het `True`- als `Falseblok` zijn dus verplicht.

# 4 Voorbeeld programma's

### Police

Dit programma vindt u terug in `police.t21`, u kan het uitvoeren met het volgende commando: `runhaskell Main.hs police.t21`.
In dit programma laten we de robot een politiewagen simuleren.

Het voorbeeld begint met het definiëren van een aantal variabelen. Een van deze variabelen, `x`, bepaalt hoeveel keer we de LED's aan en af zetten. Vervolgens zetten we de eerste let op rood en wachten we gedurende `wait` milliseconden, waarna we deze weer afzetten. Dan wachten we nog eens `wait` milliseconden en herhalen we het process voor de tweede LED, die we blauw kleuren. De lus wordt afgesloten door de variabele `x` met 1 te verminderen.
Tenslotte zetten we beide LED's terug uit na het aflopen van de lus.

### Obstacles

Dit programma vindt u terug in `obstacles.t21`, u kan het uitvoeren met het volgende commando: `runhaskell Main.hs obstacles.t21`.
In dit voorbeeld rijdt de robot vooruit en probeert hij obstakels te ontwijken.

Het programma begint opnieuw met een aantal variabele declaraties. Hier bepaalt `t` hoe dicht een obstakel mag staan eer we het proberen te ontwijken. Na de variabele declaraties starten we een oneindige lus. In deze lus slaan we de waarde van de ultrasone sensor op in de variabele `distance`. De waarde van de sensor vragen we op met de `Feel` expressie. Deze heet zo omdat het is alsof de robot voor zich aan het voelen is om zeker te zijn dat hij nergens tegen loopt.
Indien we geen obstakel tegenkomen binnen onze threshold `t`, lopen we gewoonweg vooruit. Indien we wel een obstakel tegenkomen, dan stoppen we de robot en rijden we achteruit gedurende `mwalk` milliseconden. Daarna draaien we gedurende `time` milliseconden naar rechts. Hierna eindigt deze iteratie van de while-lus en herhalen we deze operaties opnieuw.

### Line

Dit programma vindt u terug in `line.t21`, u kan het uitvoeren met het volgende commando: `runhaskell Main.hs line.t21`.
In dit voorbeeld probeert de robot een donkere lijn te volgen.

We definiëren opnieuw een aantal variabelen en starten dan terug een oneindige lus. In deze lus halen we eerst de waarde van onze lijn sensors op met de `Look` expressie. Dit slaan we op in de `curr` variabele. De `Look` expressie heet zo omdat het is alsof de robot naar de grond kijkt.

Vervolgens kijken we of beide sensoren een wit oppervlak zien. Indien dit zo is en dit in de vorige meting ook zo was, dan gaan we er van uit dat we onze lijn kwijt zijn en draaien we constant naar rechts op zoek naar een lijn.
Indien beide sensoren beiden een wit oppervlak zien en de vorige iteratie enkel de linkse sensor een zwart oppervlak zag, draaien we naar links zolang we niet met beide sensoren een zwart oppervlak zien.
Als de rechtse sensor in de vorige iteratie een zwart oppervlak zag, dan draaien we naar rechts tot beide sensoren een zwart oppervlak zien.
Indien één van de vorige twee gevallen voorkomt, passen we `curr` aan naar de nieuwe waarde en stoppen we de robot. Dit doen we om te vermijden dat hij terug onmiddellijk van de lijn afwijkt en om zeker te zijn over onze vorige meting (er kan geen meting gemist zijn indien de robot stil staat).

Indien één van de sensoren dus een zwart oppervlak ziet rijden we vooruit. Indien beide echter een wit oppervlak zien proberen we de robot terug op de lijn te krijgen a.h.v. van de vorige gemeten waarde.
We eindigen de lus door de huidige waarde op te slaan in `prev`, om deze in de volgende iteratie beschikbaar te hebben.

# 5 Implementatie

Op lijn 11 en 12 van `Main.hs` wordt een bestand aangemaakt. Dit bestand is een log bestand, na de afloop van een programma vindt u de trace van het programma terug in dit bestand. Deze twee lijnen gaan er van uit dat de naam van het programma eindigt met een extensie van exact drie karakters lang. Het is dus niet verplicht om een bestand de extensie `t21` te geven, maar ze moet wel drie karakters lang zijn.

Lijn 10 en 15 van `Main.hs` open en sluiten de connectie van de robot, dit gebeurt dus slechts één maal in het programma en niet bij elk commando.

Op lijn 22 van `Interpreter.hs` wordt de waarde van de ultrasone waarde opgehaald en afgekapt om er een integer van te maken. Dit doen we omdat we dan enkel rekening moeten houden met integers en omdat die precisie overbodig is (volgens eigen testen).

Op lijn 23 van `Interpreter.hs` wordt de waarde van de lijn sensor opgehaald en omgezet in waarde van 1 t.e.m. 4. Dit wordt opnieuw gedaan zodat we ons enkel zorgen hoeven te maken over integers.

De lijnen 48 t.e.m 64 in `Interpreter.hs` laten ons toe om eenvoudig binaire operators te evalueren. Om een booleaanse waarde te verkrijgen uit een binaire operatie (voor `&&` en `||`) gebruiken we de lijnen 53-59. Deze zetten eerst de integer waarden om in booleans (strikt positieve waarden zijn `True`, de andere `False`) en voeren dan de booleanse operator uit in Haskell. Tenslotte wordt deze boolean, alsook elke andere boolean in het programma, terug omgezet in een integer waarde. `True` wordt voorgesteld door de waarde 1 en `False` door de waarde 0.

De lijnen 82 t.e.m. 107 in `Interpreter.hs` maken gebruik van de `plog` functie (gedefiniëerd op lijn 161). Deze logs vormen de trace in `programma_naam.log` zoals eerder vermeld.

In `Parser.hs` worden de lijnen 155 t.e.m. 161 gebruikt om eenvoudig expressies met binaire operatoren te parsen.

De lijnen 164 t.e.m. 166 in `Parser.hs` laten toe om indentatie toe te voegen aan de T21 programma's.

Tenslotte laten de lijnen 256 t.e.m. 259 in `Parser.hs` toe van expressies te omringen met ronde haken.

# 6 Conclusie

Voor dit project hebben we een minimale programmeertaal, genaamd T21, ontworpen voor het besturen van een robot. De taal gebruikt strikte naamgevingen en is strikt over het gebruikt van witruimte in zijn programma's om de code beter te doen ogen. Ze gebruikt welbekende concepten uit bestaande programmeertalen, zoals `while` en `if` om eenvoud te behouden. Ze maakt intern ook enkel gebruik van integers, wat de taal zeer eenvoudig maakt en type declaraties overbodig maakt.

Met behulp van deze taal kunnen we de LED's en motoren van een mBot controleren. Dit laat ons toe om de robot om te vormen tot een politiewagen, hem obstakels te doen ontwijken en hem lijnen te doen volgen.

Tenslotte kunnen er nog enkele verbeteringen aangebracht worden aan de taal.
We gebruiken op dit ogenblik enkel integers, waardoor we dus geen strings kunnen gebruiken. Dit zou echter handig zijn voor logging, gebruikersinteractie en IO.
Al de functies zijn op dit ogenblik gedefiniëerd en geïmplementeerd in de compiler. In de toekomst zou de gebruiker dit zelf moeten kunnen om zijn code verder te kunnen opsplitsen en te hergebruiken.
Het zou ook nuttig zijn een multiline comment te voorzien. Dit is echter niet noodzakelijk.

# 7 Appendix broncode

```haskell
import           Control.Monad.State
import qualified Data.Map            as M
import           Interpreter
import           MBot
import           System.Environment

main =  do
  (args:_) <- getArgs
  contents <- readFile args
  d <- openMBot
  let name = reverse (drop 4 $ reverse args)
  writeFile (name ++ ".log") ""
  -- Evaluate the statements and intialize an empty state
  runStateT (evalStats d name $ lines contents) M.empty
  closeMBot d
```

Listing 1: Main.hs

```haskell
module Interpreter
( evalExp
, evalStats
) where

import           Commander           as C
import           Control.Monad.State
import qualified Data.Map            as M
import           MBot                (Line (..))
import           Parser
import           System.Directory    as D
import           System.HIDAPI

-- The State Map: it holds all of our variables
type SMap = M.Map String Int
-- The Program State: concists of a state (the map) and an IO action as value
type PState = StateT SMap IO ()

-- Evaluates an expression
evalExp :: Device -> SMap -> Exp -> IO Int
evalExp d s expr = case expr of
      Feel     -> do f <- C.ultraSonic d; return $ truncate f
      Look     -> do status <- C.lineF d
                     case status of
                       LEFTB  -> return 1
                       RIGHTB -> return 2
                       BOTHB  -> return 3
                       BOTHW  -> return 4
      Lit n    -> return n
      e :*: f  -> evalAriOp  (*)  e f
      e :+: f  -> evalAriOp  (+)  e f
      e :-: f  -> evalAriOp  (-)  e f
      e :==: f -> evalBinOp  (==) e f
      e :!=: f -> evalBinOp  (/=) e f
      e :>: f  -> evalBinOp  (>)  e f
      e :>=: f -> evalBinOp  (>=) e f
      e :<: f  -> evalBinOp  (<)  e f
      e :<=: f -> evalBinOp  (<=) e f
      e :&&: f -> evalBBinOp (&&) e f
      e :||: f -> evalBBinOp (||) e f
      Name n   -> case M.lookup n s of
                    Nothing -> Prelude.error $ "Using uninitialized variable: " ++ n
                    Just x  -> return x
      _        -> Prelude.error "Unknown expression found in evalExp"
      where
      -- Helper functions
            -- Evaluates an arithmetic operation
            evalAriOp :: (Int -> Int -> Int) -> Exp -> Exp -> IO Int
            evalAriOp o e f = o <$> evalExp d s e <*> evalExp d s f
            -- Evaluates an binary operation
            evalBinOp :: (Int -> Int -> Bool) -> Exp -> Exp -> IO Int
            evalBinOp o e f = getInt $ o <$> evalExp d s e <*> evalExp d s f
            -- Evaluates a boolean binary operation
            evalBBinOp :: (Bool -> Bool -> Bool) -> Exp -> Exp -> IO Int
            evalBBinOp o e f = getInt $ o <$> getBool (evalExp d s e) <*> getBool (evalExp d s f)
            -- Transforms an IO Int into an IO Bool
            getBool :: IO Int -> IO Bool
            getBool ion = do n <- ion
                             return (n > 0)
            -- Transforms an IO Bool into an IO Int
            getInt :: IO Bool -> IO Int
            getInt iob = do b <- iob
                            if b
                            then return 1
```

```haskell
                          else return 0

-- Evaluates multiple expressions
evalExps :: Device -> SMap -> [Exp] -> IO [Int]
evalExps d m = mapM (evalExp d m)

-- Represents a block of statements
type Block = [String]

-- Evaluates all the statements of the program
evalStats :: Device -> String -> Block -> PState
evalStats _ _ [] = return ()
evalStats d name (s:xs) =
      if null s -- Allow empty lines
      then evalStats d name xs
      else do m <- get
              case parse parseExpression s of
                n :=: v -> do plog name $ "Assignment " ++ show n ++ " " ++ show v
                              do value <- liftIO $ evalExp d m v
                                 setVar (getName n) value
                                 recall
                Rgb p -> do plog name $ "Rgb " ++ show p
                            lift $ do exps <- evalExps d m p
                                      if length exps /= 4
                                      then Prelude.error "RGB has 4 parameters!"
                                      else C.rgb d exps
                            recall
                Wait t -> do plog name $ "Wait " ++ show t
                             lift $ do time <- evalExp d m t; C.wait time
                             recall
                Walk -> do plog name "Walk"; lift $ C.walk d; recall
                Moonwalk -> do plog name "Moonwalk"; lift $ C.moonwalk d; recall
                Hammertime -> do plog name "Hammertime"; lift $ C.stop d; recall
                TurnLeft -> do plog name "Turn left"; lift $ C.left d; recall
                TurnRight -> do plog name "Turn right"; lift $ C.right d; recall
                Comment -> recall
                While bexp -> do plog name $ "While " ++ show bexp
                                 processWhile d name bexp wBlock
                                 -- +1 because wBlock does not contain the Elihw
                                 evalStats d name $ drop (length wBlock +1) xs
                                 where
                                   wBlock = reverse $ findBlock Elihw 0 xs []
                If bexp -> do plog name $ "If " ++ show bexp
                              processIf d name bexp tBlock fBlock
                              evalStats d name $ drop (length tBlock + length fBlock + 2) xs
                              where
                                tBlock = reverse $ findBlock Else 0 xs []
                                fBlock = reverse $ findBlock Fi   0 (drop (length tBlock + 1) xs) []
                e -> Prelude.error $ "Invalid statement: " ++ show e
                where
                      recall = evalStats d name xs

-- Processes a while loop
processWhile :: Device -> String -> Exp -> Block -> PState
processWhile d name e wBlock = do m <- get
                                  r <- lift $ evalExp d m e
                                  when (r > 0) $
                                    do evalStats d name wBlock
                                       processWhile d name e wBlock

-- Proccess an if statement
processIf :: Device -> String -> Exp -> Block -> Block -> PState
processIf d name e tBlock fBlock = do m <- get
                                      r <- lift $ evalExp d m e
                                      if r > 0
                                      then evalStats d name tBlock
                                      else evalStats d name fBlock
```

```haskell

-- Represents the level of conditional nesting
type Level = Int
-- Finds a block of a conditional, either a the block of a while loop or the
-- true/false block in an if statement
findBlock :: Exp -> Level -> Block -> Block -> Block
findBlock e _ [] _ = Prelude.error $ "Missing " ++ show e
findBlock e l (s:xs) c
      | l < 0 =  Prelude.error $ "Missing " ++ show e
      | e == r = if l == 0 then c else findBlock e (l-1) xs c++[s]
      | otherwise = case e of
            -- We're looking for a while block
            Elihw -> case r of
              -- Found another while, increase depth
              While _ -> findBlock e (l+1) xs c++[s]
              _ -> findBlock e l xs c++[s]
            -- We're looking for a true block
            Else -> case r of
              If _ -> findBlock e (l+1) xs c++[s]
              _ -> findBlock e l xs c++[s]
            -- We're looking for a false block
            Fi -> case r of
              If _ -> findBlock e (l+1) xs c++[s]
              _ -> findBlock e l xs c++[s]
            expr -> Prelude.error $ "Trying to find block of an unsupported expression: " ++ show expr
      where
        r = parse parseExpression s

-- Logs a string to a given file
plog :: String -> String -> PState
plog name s = liftIO $ appendFile (name ++ ".log") (s ++ "\n")

-- Adds or sets a variable in the State
setVar :: String -> Int -> PState
setVar k v = do
      m <- get
      put $ M.insert k v m
      return ()

-- Returns the name of a Name expression
getName :: Exp -> String
getName (Name n) = n
getName e        = Prelude.error "Trying to get an identifier out of an unsupported expression: " ++ show e
```

Listing 2: Interpreter.hs

```haskell
module Parser
( Exp(..)
, Params
, parseExpression
, parse
) where

import           Control.Applicative
import           Control.Monad
import           Data.Char

-- The type of parsers
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 == "" ]
```

```haskell
  where
  one [] = error "no parse"
  one [x] = x
  one xs | length xs > 1 = error "ambiguous parse"

-- Adding Applicative/Functor Instances for the Parser Monad according to the
-- Functor-Applicative-Monad Proposal
instance Functor Parser where
  fmap = liftM

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

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 ])

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

-- 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 given string
match :: String -> Parser String
match = mapM token

-- 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)

-- 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
```

```
89              return (-n)
90
91  -- Match an integer
92  parseInt :: Parser Int
93  parseInt = parseNat `mplus` parseNeg
94
95  -- Match a lower case char
96  parselChar :: Parser Char
97  parselChar = spot isLower
98
99  -- Match an alphanumeric string
100 parseString :: Parser String
101 parseString = star (spot isAlphaNum)
102
103 -- Match False
104 parseFalse :: Parser Int
105 parseFalse = do match "False"
106                 return 0
107
108 -- Match True
109 parseTrue :: Parser Int
110 parseTrue = do match "True"
111                return 1
112
113 -- Match a boolean
114 parseBoolean :: Parser Int
115 parseBoolean = parseFalse `mplus` parseTrue
116
117 -- Match any string
118 parseRandom :: Parser String
119 parseRandom = plus (spot $ not . isControl)
120
121 type Params = [Exp]
122
123 data Exp = Lit Int
124          | Name String
125          | Exp :+:  Exp
126          | Exp :-:  Exp
127          | Exp :*:  Exp
128          | Exp :=:  Exp
129          | Exp :==: Exp
130          | Exp :!=: Exp
131          | Exp :>:  Exp
132          | Exp :>=: Exp
133          | Exp :<:  Exp
134          | Exp :<=: Exp
135          | Exp :&&: Exp
136          | Exp :||: Exp
137          | While Exp
138          | Elihw
139          | If Exp
140          | Else
141          | Fi
142          | Rgb Params
143          | Wait Exp
144          | Look
145          | Feel
146          | Walk
147          | Moonwalk
148          | Hammertime
149          | TurnLeft
150          | TurnRight
151          | Comment
152         deriving (Eq,Show)
153
154 -- Parse a binary operation
155 parseBinOp :: String -> Parser (Exp, Exp)
```

```
156 parseBinOp o = do token '('
157                   f <- parseExp
158                   match $ " " ++ o ++ " "
159                   s <- parseExp
160                   token ')'
161                   return (f, s)
162
163 -- Allow for whitespace in front of a statement
164 parseExpression :: Parser Exp
165 parseExpression = do _ <- star (spot isSpace)
166                      parseExp
167
168 -- Actually parse an expression
169 parseExp :: Parser Exp
170 parseExp = parseLit `mplus` parseBool `mplus` parseWhile `mplus`
171            parseElihw `mplus `parseIf `mplus` parseFi `mplus` parseElse `mplus`
172            parseName `mplus` parseAdd `mplus` parseMin `mplus` parseMul `mplus`
173            parseAssign `mplus` parseEq `mplus` parseNeq `mplus`
174            parseGt `mplus` parseGte `mplus` parseLt `mplus` parseLte `mplus`
175            parseAnd `mplus` parseOr `mplus` parseRGB `mplus` parseWait `mplus`
176            parseLook `mplus`parseFeel `mplus` parseWalk `mplus`
177            parseMwalk `mplus` parseHTime `mplus` parseTLeft `mplus`
178            parseTRight `mplus` parseComm `mplus` parseBraces
179   where
180   parseLit    = do n <- parseInt
181                    return (Lit n)
182   parseBool   = do n <- parseBoolean;
183                    return (Lit n)
184   parseName   = do f <- parselChar
185                    r <- parseString
186                    return $ Name $ f:r
187   parseAdd    = do (d, e) <- parseBinOp "+"
188                    return (d :+: e)
189   parseMin    = do (d, e) <- parseBinOp "-"
190                    return (d :-: e)
191   parseMul    = do (d, e) <- parseBinOp "*"
192                    return (d :*: e)
193   parseAssign = do n <- parseName
194                    match " = "
195                    v <- parseExp
196                    return (n :=: v)
197   parseEq     = do (d, e) <- parseBinOp "=="
198                    return (d :==: e)
199   parseNeq    = do (d, e) <- parseBinOp "!="
200                    return (d :!=: e)
201   parseGt     = do (d, e) <- parseBinOp ">"
202                    return (d :>: e)
203   parseGte    = do (d, e) <- parseBinOp ">="
204                    return (d :>=: e)
205   parseLt     = do (d, e) <- parseBinOp "<"
206                    return (d :<: e)
207   parseLte    = do (d, e) <- parseBinOp "<="
208                    return (d :<=: e)
209   parseAnd    = do (d, e) <- parseBinOp "&&"
210                    return (d :&&: e)
211   parseOr     = do (d, e) <- parseBinOp "||"
212                    return (d :||: e)
213   parseWhile  = do match "WHILE "
214                    c <- parseExp
215                    return $ While c
216   parseElihw  = do match "ELIHW"
217                    return Elihw
218   parseIf     = do match "IF "
219                    c <- parseExp
220                    return $ If c
221   parseElse   = do match "ELSE"
222                    return Else
```

```
223   parseFi     = do match "FI"
224                    return Fi
225   parseRGB    = do match "Rgb("
226                    l <- parseExp
227                    match ", "
228                    r <- parseExp
229                    match ", "
230                    g <- parseExp
231                    match ", "
232                    b <- parseExp
233                    token ')'
234                    return (Rgb [l, r, g, b])
235   parseWait   = do match "Wait("
236                    t <- parseExp
237                    token ')'
238                    return (Wait t)
239   parseLook   = do match "Look"
240                    return Look
241   parseFeel   = do match "Feel"
242                    return Feel
243   parseWalk   = do match "Walk"
244                    return Walk
245   parseMwalk  = do match "Moonwalk"
246                    return Moonwalk
247   parseHTime  = do match "Hammertime"
248                    return Hammertime
249   parseTLeft  = do match "TurnLeft"
250                    return TurnLeft
251   parseTRight = do match "TurnRight"
252                    return TurnRight
253   parseComm   = do match "-- "
254                    parseRandom
255                    return Comment
256   parseBraces = do token '('
257                    e <- parseExp
258                    token ')'
259                    return e
```

Listing 3: Parser.hs

```
1  module Commander
2  ( walk
3  , moonwalk
4  , stop
5  , left
6  , right
7  , wait
8  , rgb
9  , ultraSonic
10 , lineF
11 ) where
12
13 import           Control.Concurrent
14 import           Data.Bits
15 import qualified MBot               as M
16 import           System.HIDAPI
17
18 -- ID's for the left and right motor
19 lm = 0x9
20 rm = 0xa
21 -- Defaults for motor speed
22 rightM = (80, 0)
23 leftM  = mapT complement rightM
24 turn = (80, 0)
25
26 -- Move the robot forwards
27 walk :: Device -> IO ()
```

```haskell
walk d = motors d leftM rightM

-- Move the robot backwards
moonwalk :: Device -> IO ()
moonwalk d = motors d (mapT complement leftM) (mapT complement rightM)

-- Stop the robot
stop :: Device -> IO ()
stop d = motors d (0, 0) (0, 0)

-- Make the robot turn left
left :: Device -> IO ()
left d = motors d turn (0, 0)

-- Make the robot turn right
right :: Device -> IO ()
right d = motors d (0, 0) (mapT complement turn)

-- Activate the motors with given speeds
motors :: Device -> (Int, Int) -> (Int, Int) -> IO ()
motors d (ls, lu) (rs, ru) = do send d $ M.setMotor rm rs ru
                                send d $ M.setMotor lm ls lu

-- Activate a given led with a given colour
rgb :: Device -> [Int] -> IO ()
rgb d p = send d $ M.setRGB (head p) (p !! 1) (p !! 2) (p !! 3)

-- Read the value from the ultra sonic sensor
ultraSonic :: Device -> IO Float
ultraSonic = M.readUltraSonic

-- Read the value from the line follower sensor
lineF :: Device -> IO M.Line
lineF = M.readLineFollower

-- Sleeps for t milliseconds
wait :: Int -> IO ()
wait t = threadDelay $ t*1000

-- Helper function

-- Send a command to the robot
send :: Device -> M.Command -> IO ()
send = M.sendCommand

-- The map function over a tuple
mapT :: (a -> a) -> (a, a) -> (a, a)
mapT f (l, r) = (f l, f r)
```

Listing 4: Commander.hs