UNIVERSITEIT GENT

FUNCTIONEEL PROGRAMMEREN

CHRISTOPHE SCHOLLIERS

Haskell interpreter: Komainski

Authors
BERT COMMEINE
MATTEO HOLVOET

21 januari 2018

Inhoudsopgave

1 Inleiding

In dit project heb ik gewerkt aan een interpreter voor mijn zelfverzonnen taal "Komainski". De bedoeling was om per twee deze opdracht te vervolledigen, maar we zouden graag hebben dat onze stukken apart beoordeeld worden.

De inspiratie heb ik grotendeels gehaald uit Java. Het starten af sluiten van blokken code met accolades, en eindigen van statements met een puntkomma maken het parsen een stuk makkelijker. Daarnaast zijn codewoorden vervangen door hun "Jommekes-Russische"varianten.

De taal bevat if-statements en while-loops als controlestructuren. Daarnaast is het mogelijk om waarden op te slaan in de environment. Deze waarden kunnen Booleaanse waarden zijn, Integers en Floats. Daarnaast is het ook mogelijk de MBot te besturen vanuit de taal. Daarvoor zijn er enkele commando's geïntegreerd zoals "driveskiën "ledski".

2 Syntax en semantiek

2.1 Syntax

⟨*Value*⟩ ::= Wrong
| Num
| #true | #false
| Dec

⟨*Direction*⟩ ::= leftski
| rightski

⟨*Expression*⟩ ::= ⟨*Empty*⟩
| Con
| Var
| #sensorski
| #lightski Direction
| BoolVal
| Decimal
| (Expression + Expression)
| (Expression * Expression)
| (Expression - Expression)
| (Expression / Expression)
| (Expression == Expression)
| (Expression > Expression)
| (Expression <Expression)
| ∼ Expression

⟨*Statement*⟩ ::= ⟨*Empty*⟩
| String := Expression
| whileski Expression doski { Statement }
| ifski Expression thenski { Statement } elski { Statement }

| Statement; Statement;
| printski(Expression);
| driveski Expression;
| turnski Direction Expression;
| ledski (Expression, Expression, Expression, Expression)
| – Comment –

2.2 Semantiek

2.2.1 Values

Values zijn de waarden die kunnen worden opgeslagen in de Environment. Deze bestaan ofwel uit een discrete waarde (Num), een booleaanse waarde of een float (Dec). Wanneer er wordt gezocht naar een waarde die niet opgeslagen is wordt een value met waarde "Wrong"teruggegeven.

2.2.2 Expression

Een expression is een term die kan worden geëvalueerd tot een Value. Deze kunnen bestaan uit Vars die worden opgezocht in de environment, Values zelf, Aritmethic operatoren, Booleaanse operatoren en leesbewerkingen op de mbot.

2.2.3 Statement

Een statement is een term die effectief kan worden uitgevoerd. Deze bestaan uit assignements, een if- en while-controlestructuren, een printcommando, mogelijkse besturingen van de mbot en een Seq of sequentie. Deze laatste is een opeenvolging van Statements. Wanneer een programma wordt geparsed zal dit programma waarschijnlijk de vorm van een Seq aannemen.

3 Voorbeelden

We beginnen met de politieauto. Allereerst gaan we een waarde opslaan onder de noemen "teller"Deze is nodig om ons programma te doen afsluiten. Daarna volgt een while-loop. We laten onze politieauto 20 cycli lopen. Daarna zetten we de eerste LED op rood en de tweede op blauw. Daarna wisselen we.

```
teller := 1;
    whileski(teller < 20)
    doski{
      ledski (1 , 0 , 0 , 100);
      ledski (2 , 100 , 0 , 0);
      teller := (teller + 1);
      ledski (1 , 100 , 0 , 0);
      ledski (2 , 0 , 0 , 100);
    }
    ledski (1 , 0 , 0 , 0);
    ledski (2 , 0 , 0 , 0);
```

Hierna bekijken we het voorbeeld met de lijn. In mijn implementatie ben ik ervan uitgegaan dat wit positief was en zwart negatief, dus we gaan overal onze not-operator toepassen. We beginnen terug met een teller, want we willen niet vastzitten in een oneindige lus.

```
teller := 1;
whileski(teller < 200)
doski{
  ifski ~ #lightski leftski
  thenski {
    ifski ~ #lightski rightski
    thenski {
      driveski 70;
    } elski {
      turnski leftski 70;
    }
  } elski {
    ifski ~ #lightski rightski
    thenski {
      turnski rightski 70;
    } elski {
      driveski -70;
    }
  }
  teller := (teller + 1);
}
driveski 0;
```

Daarna gaan we kijken als er links van de robot zwart te vinden is. Wanneer dit zo is, kijken we of er rechts ook zwart te vinden is. Is dit weeral het geval, rijden we rechtdoor. Als dit niet zo is draaien we naar links. Als er links geen zwart meer te vinden is testen we opnieuw of er rechts nog zwart te vinden is. In dit geval gaan we naar rechts, anders, als er langs beide kanten geen zwart meer is, rijden we achteruit tot we terug op de zwarte strook zitten. Na onze loop laten we de robot terug tot stilstand komen.

Als laatste bekijken we het script die de robot obstakels doet ontwijken. Deze ziet eruit als volgt:

```
teller := 1;
  whileski(teller < 60)
  doski{
    ifski (#sensorski > 15)
    thenski {
      driveski 100;
    } elski {
      turnski rightski 100;
    }
    teller := (teller + 1);
```

```
}
driveski 0;
```

We gaan weer op dezelfde manier tewerk als bij het vorige voorbeeld: We starten met een loop. In die loop gaan we testen als er iets voor ons ligt. De waarde 15 is arbitrair, maar bleek uit testen goed te werken. Wanneer er iets te zien is voor ons, gaan we uitwijken. Als er niets te zien is rijden we gewoon rechtdoor.

4 Implementatie

4.1 Datatypes

Voor datatypes ben heb ik gekozen voor de volgende oplijsing:

- **Value** : De waarden die aan een variabele kunnen worden toegewezen. Dit zijn Booleaanse waarden, Integers en Floats.
- **Expression** : Expressions zijn datatypes die kunnen worden vereenvoudigd naar Values. Dit gaat dan om bewerkingen of vergelijkingen.
- **Statement** Een statement is iets wat kan worden uitgevoerd. Dit geeft geen waarde terug. Voorbeelden zijn "Assignement", "Printën de meeste controlestructuren. Opmerkelijk is misschien de statement "Seq [Statement]". Deze verzameling van statements stelt een zogenaamde block code voor.

4.2 Parser

Deze module diende als vervanging voor een klassieke parser. Het is grotendeels gebaseerd op de parser die we in de les behandeld hebben. In de module wordt de Monad gedefinieerd, samen met zijn MonadPlus, en de functies äpplyën "parse".

4.3 Parser-extended

Hierin worden alle parser gedefinieerd die we specifiek voor dit project nodig hadden. Eerst worden simpele stukjes opgebouwd (parsers voor Integers, Strings, ...) om dan op te bouwen naar drie grote parsefuncties, nl. ParseExpression, ParseStatement en multipleStatementParser. In de eerste zitten dan voor elk soort Expression een unieke parser die via de monadplus bewerking worden gekoppeld. De volgende werkt op dezelfde manier, voor elk statement een parser en die samen laten werken. De laatste gaat meerdere keren die functie oproepen, en zo een "Seq [Statement]"teruggeven. Deze laatste is degene die we gebruiken wanneer we een bestand gaan parsen.

```
multipleStatementsParser = do { stmts <- many parseStatement;
                                return $ Seq stmts}
```

4.4 Evaluator

Mijn evaluator is veruit mijn grootste stuk, maar bevat in essentie maar twee functies (en hun hulpfuncties). De eerste functie is ëvalën zet een expression om naar een value. In geval van operaties wordt er rekening gehouden met het type van de deelexpressies. Bij operaties tussen Integers en Floats worden de integers eerst omgezet naar Floats.

De tweede functie is ëxecute". Deze voert een Statement uit, en geeft Env() terug. Hier werden opnieuw een aantal hulpfuncties voorzien voor onder de MBot-gerelateerde Statements. Merk op dat bij de execute van een "Seq"statement we deze operatie folden over onze lijst met statements.

4.5 State Monad

Deze module bevat de monad transformer. Initieel was het mijn plan om een simpele [(String, Value)] te gebruiken als omgeving, maar blijkbaar was het integreren van de IO makkelijker wanneer ik een generieke Map gebruikte als staat. Hier worden ook de twee functies voorzien om waarden op te halen en toe te voegen aan de map. Vanwege de werking van de map moest ik geen extra functie voorzien om een waarde te vervangen. Het type van onze centrale Env is trouwens *StateT Environment IO a*.

4.6 Komainski

Hier wordt de main beschreven. Deze gaat het filepath ophalen, de file inlezen, parsen en uitvoeren. Aangezien we in de parser zelf geen rekening houden met mogelijke spaties, newlines of tabs filteren we die eerst weg.

5 Conclusie

Met haskell ben ik erin geslaagd een eigen parser te bouwen, en het resultaat daarvan te interpreteren en uit te voeren. Ik had graag nog Strings in mijn values gestoken, ookal was dit geen vereiste voor de opgave. Daarnaast denk ik dat het uitdagender was geweest om mijn eigen Environment te gebruiken en die te integreren in een eigen StateMonad, maar de methode die ik nu toe paste bleek een pak sneller te gaan.

5.1 Appendix

Datatypes

```
1  module Datatypes where
2
3    type Words = String
4
5    data Direction     = Right
6                       | Left
7                       deriving(Eq, Show)
8
9    data Value         = Wrong
10                      | Num Int
11                      | Bool Bool
12                      | Dec Float
13                      deriving(Eq)
14
15   data Expression    = Var Words
16                      | Con Int
17                      | ReadSensor
18                      | LightSensor Direction
19                      | BoolVal Bool
20                      | Decimal Float
21                      -- Artithmic operators ----------------
22                      | Add Expression Expression
23                      | Mul Expression Expression
24                      | Sub Expression Expression
25                      | Div Expression Expression
26                      -- Comparing operators ----------------
27                      | Eq  Expression Expression
28                      | Greater Expression Expression
29                      | Lesser Expression Expression
30                      -- Boolean operators ------------------
31                      | And Expression Expression
32                      | Or  Expression Expression
33                      | Not Expression
34                      deriving( Show, Eq)
35
36   data Statement     = Assignement Words Expression
37                      | If Expression Statement Statement
38                      | While Expression Statement
39                      | Seq [Statement]
40                      | Print Expression
41                      | Comment
42                      | Drive Expression
43                      | Turn Direction Expression
44                      | Lights Expression Expression Expression Expression
45                      deriving(Eq)
46                      --Turn Expression
47
48   instance (Show Value) where
49     show Wrong     = "Wrong"
50     show (Num i)   = "Int " ++ show i
51     show (Bool i) = show i
52     show (Dec i)    = "Decimal " ++ show i
53
54   instance (Show Statement) where
```

```
55      show (Assignement e f) = "Assign " ++ show e ++ show f
56      show (If ex stmt1 stmt2)  = "If " ++ show ex ++ "then " ++ show stmt1 ++ "
            else " ++ show stmt2
57      show (While ex stmt) = "While " ++ show ex ++ "Do " ++ show stmt
58      show (Print i) = "Print " ++ show i
```

Evaluator

```
1  module Evaluator (eval, execute) where
2    import  Datatypes
3    import  StateMonad
4    import  Simulator
5
6    import  Control.Monad.IO.Class
7    import  Control.Monad (MonadPlus, mplus, mzero, guard, when, unless, void)
8
9    add :: Value -> Value -> Value
10   add (Num x) (Num y) = Num (x + y)
11   add (Dec x) (Dec y) = Dec (x + y)
12   add (Dec x) (Num y) = Dec (x + fromIntegral y)
13   add (Num x) (Dec y) = Dec (fromIntegral x + y)
14   add _       _       = Wrong
15
16   mul :: Value -> Value -> Value
17   mul (Num x) (Num y) = Num (x * y)
18   mul (Dec x) (Dec y) = Dec (x * y)
19   mul (Dec x) (Num y) = Dec (x * fromIntegral y)
20   mul (Num x) (Dec y) = Dec (fromIntegral x * y)
21   mul _       _       = Wrong
22
23   sub :: Value -> Value -> Value
24   sub (Num x) (Num y) = Num (x - y)
25   sub (Dec x) (Dec y) = Dec (x - y)
26   sub (Dec x) (Num y) = Dec (x - fromIntegral y)
27   sub (Num x) (Dec y) = Dec (fromIntegral x - y)
28   sub _       _       = Wrong
29
30   divide :: Value -> Value -> Value
31   divide (Num x) (Num y) = Num (x `div` y)
32   divide (Dec x) (Dec y) = Dec (x / y)
33   divide (Dec x) (Num y) = Dec (x / fromIntegral y)
34   divide (Num x) (Dec y) = Dec (fromIntegral x / y)
35   divide _       _       = Wrong
36
37   equal :: Value -> Value -> Value
38   equal (Num x)   (Num y)   = Bool (x == y)
39   equal (Bool x)  (Bool y)  = Bool (x == y)
40   equal (Dec x)   (Dec y)   = Bool (x == y)
41   equal (Dec x)   (Num y)   = Bool (x == fromIntegral y)
42   equal (Num x)   (Dec y)   = Bool (fromIntegral x == y)
43   equal _       _           = Wrong
44
45   greater :: Value -> Value -> Value
46   greater (Num x) (Num y)   = Bool (x > y)
47   greater (Dec x) (Dec y)   = Bool (x > y)
48   greater (Dec x) (Num y)   = Bool (x > fromIntegral y)
49   greater (Num x) (Dec y)   = Bool (fromIntegral x > y)
50   greater _       _         = Wrong
```

```
51
52  lesser :: Value -> Value -> Value
53  lesser (Num x) (Num y)    = Bool (x < y)
54  lesser (Dec x) (Dec y)    = Bool (x < y)
55  lesser (Dec x) (Num y)    = Bool (x < fromIntegral y)
56  lesser (Num x) (Dec y)    = Bool (fromIntegral x < y)
57  lesser _       _          = Wrong
58
59  andBool :: Value -> Value -> Value
60  andBool (Bool x)  (Bool y)  = Bool ( x && y)
61  andBool _         _         = Wrong
62
63  orBool :: Value -> Value -> Value
64  orBool (Bool x)  (Bool y)  = Bool ( x || y)
65  orBool _         _         = Wrong
66
67  notBool :: Value -> Value
68  notBool (Bool False)  = Bool True
69  notBool (Bool True )  = Bool False
70  notBool _             = Wrong
71
72  drive :: Value -> Env()
73  drive (Num v) = do {    d <- liftIO openMBot;
74                          liftIO $ sendCommand d $ setMotor v v;
75                          liftIO $ closeMBot d; }
76  drive _       = return ()
77
78  turn :: Direction -> Value -> Env()
79  turn Datatypes.Right (Num v) = do {    d <- liftIO openMBot;
80                                liftIO $ sendCommand d $ setMotor v 0;
81                                liftIO $ closeMBot d; }
82  turn Datatypes.Left (Num v) = do {    d <- liftIO openMBot;
83                                liftIO $ sendCommand d $ setMotor 0 v;
84                                liftIO $ closeMBot d; }
85  turn _     _      = return ()
86
87
88  lights :: Value -> Value -> Value -> Value -> Env()
89  lights (Num e1) (Num e2) (Num e3) (Num e4) = do {   d <- liftIO openMBot;
90                                                      liftIO $ sendCommand d $
                                                            setRGB e1 e2 e3 e4;
91                                                      liftIO $ closeMBot d; }
92  lights _        _        _        _       = return ()
93
94  processLine :: Line -> Direction -> Value
95  processLine LEFTB Datatypes.Left    = Bool False
96  processLine LEFTB Datatypes.Right   = Bool True
97  processLine RIGHTB Datatypes.Left   = Bool True
98  processLine RIGHTB Datatypes.Right  = Bool False
99  processLine BOTHB   _               = Bool False
100 processLine BOTHW   _               = Bool True
101
102
103
104 eval :: Expression -> Env Value
105 eval (Var x)              = fetch x
106 eval (Con x)              = return (Num x)
```

```
107  eval (BoolVal x)          = return (Bool x)
108  eval (Decimal x)          = return (Dec x)
109  eval (Add t1 t2)          = do {  a <- eval t1;
110                                    b <- eval t2;
111                                    return (add a b) }
112  eval (Mul t1 t2)          = do {  a <- eval t1;
113                                    b <- eval t2;
114                                    return (mul a b) }
115  eval (Div t1 t2)          = do {  a <- eval t1;
116                                    b <- eval t2;
117                                    return (divide a b) }
118  eval (Sub t1 t2)          = do {  a <- eval t1;
119                                    b <- eval t2;
120                                    return (sub a b) }
121  eval (Eq t1 t2)           = do {  a <- eval t1;
122                                    b <- eval t2;
123                                    return (equal a b) }
124  eval (Greater t1 t2)      = do {  a <- eval t1;
125                                    b <- eval t2;
126                                    return (greater a b) }
127  eval (Lesser t1 t2)       = do {  a <- eval t1;
128                                    b <- eval t2;
129                                    return (lesser a b) }
130  eval (And t1 t2)          = do {  a <- eval t1;
131                                    b <- eval t2;
132                                    return (andBool a b) }
133  eval (Or t1 t2)           = do {  a <- eval t1;
134                                    b <- eval t2;
135                                    return (orBool a b) }
136  eval (Not t1)             = do {  a <- eval t1;
137                                    return (notBool a) }
138  eval ReadSensor          = do {    d <- liftIO openMBot;
139                                      v <- liftIO $ readUltraSonic d;
140                                      liftIO $ closeMBot d;
141                                      return (Dec v)}
142  eval (LightSensor dir)  = do {    d <- liftIO openMBot;
143                                    line <- liftIO $ readLineFollower d;
144                                    return (processLine line dir)}
145  execute :: Statement -> Env()
146  execute Comment                  =  return ()
147  execute (Assignement name expr) =  do { v <- eval expr;
148                                          write name v}
149  execute (If expr1 stmt1 stmt2)   = do { c <- eval expr1;
150                                          if c == Bool True
151                                          then execute stmt1
152                                          else  when( c == Bool False ) $ execute
                                                stmt2}
153
154  execute (While expr stmt) = let loop() =  do {
155                                          c <- eval expr;
156                                          unless ( c == Bool False)
157                                          $ do {execute stmt; loop()} }
158                              in loop()
159  execute (Print expr) = liftIO . void . print =<< eval expr
160  execute (Seq stmts) =  foldr ((>>) . execute) (return ()) stmts
161  execute (Drive sp) = do {  v <- eval sp;
162                             drive v}
```

```
163   execute (Turn dir sp) = do {  v <- eval sp;
164                                 turn dir v}
165   execute (Lights d1 d2 d3 d4) = do { e1 <- eval d1;
166                                       e2 <- eval d2;
167                                       e3 <- eval d3;
168                                       e4 <- eval d4;
169                                       lights e1 e2 e3 e4 }
```

Parser

```
1  module Parser where
2
3  import           Control.Applicative
4  import           Control.Monad (MonadPlus, mplus, mzero)
5
6  -- The type of parsers
7  newtype Parser a = Parser (String -> [(a, String)])
8
9  -- Apply a parser
10 apply :: Parser a -> String -> [(a, String)]
11 apply (Parser f) = f
12
13 -- Return parsed value, assuming at least one successful parse
14 parse :: Parser a -> String -> a
15 parse m s = one [ x | (x,t) <- apply m s, t == "" ]
16   where
17     one [] = error "no parse"
18     one [x] = x
19     one xs | length xs > 1 = error "ambiguous parse"
20
21 instance Monad Parser where
22   return x = Parser (\s -> [(x,s)])
23   m >>= k = Parser (\s ->
24             [ (y, u) |
25               (x, t) <- apply m s,
26               (y, u) <- apply (k x) t ])
27
28 instance Applicative Parser where
29   pure        = return
30   (<*>) m1 m2 = do { f <- m1; x2 <- m2; return (f x2) }
31
32 instance Functor Parser where
33   fmap f m = pure f <*> m
34
35 instance MonadPlus Parser where
36   mzero     = Parser (const [])
37   mplus m n = Parser (\s -> apply m s ++ apply n s)
38
39 instance Alternative Parser where
40     (<|>) = mplus
41     empty = mzero
```

Extended Parser

```
1  module Extended_Parser (multipleStatementsParser) where
2
3    import Parser
4    import Datatypes
5    import Control.Applicative (many, some)
```

```
6   import Control.Monad (MonadPlus, mplus, mzero, guard, when, unless)
7   import Data.Char (isDigit, isAlpha)
8
9   -- Parse one character
10  char :: Parser Char
11  char = Parser f
12    where
13      f [] = []
14      f (c:s) = [(c,s)]
15
16  -- match one or more occurrences
17  plus :: Parser a -> Parser [a]
18  plus p = do
19           x <- p
20           xs <- some p
21           return (x:xs)
22
23  -- Parse a character satisfying a predicate (e.g., isDigit)
24  spot :: (Char -> Bool) -> Parser Char
25  spot p = do { c <- char; guard (p c); return c }
26
27  -- Match a given character
28  token :: Char -> Parser Char
29  token c = spot (== c)
30
31  -- match a natural number
32  parseNat :: Parser Int
33  parseNat = do s <- some (spot isDigit);
34                return (read s)
35
36  parseDouble :: Parser Float
37  parseDouble = do  s <- parseInt;
38                    t <- token '.';
39                    d <- parseNat;
40                    return  (read (show s ++ [t] ++ show d))
41
42
43  -- match an identifier
44  parseWords :: Parser Words
45  parseWords = plus (spot isAlpha)
46
47  -- match a Direction
48  parseDirection :: Parser Direction
49  parseDirection = parseLeft `mplus` parseRight
50              where
51                parseLeft   = do {match "leftski";
52                                  return Datatypes.Left}
53                parseRight  = do {match "rightski";
54                                  return Datatypes.Right}
55
56  -- match a negative number
57  parseNeg :: Parser Int
58  parseNeg = do
59              token '-'
60              n <- parseNat
61              return (-n)
62
```

```
63  -- match an integer
64  parseInt :: Parser Int
65  parseInt = parseNat `mplus` parseNeg
66
67  match :: String -> Parser String
68  match = mapM token
69
70  multipleStatementsParser :: Parser Statement
71  multipleStatementsParser = do { stmts <- many parseStatement;
72                                  return $ Seq stmts}
73
74  parseStatement :: Parser Statement
75  parseStatement  = parsePrint `mplus` parseIf `mplus` parseWhile `mplus`
        parseAssignement `mplus` parseDrive `mplus` parseLight `mplus` parseTurn `
        mplus` parseComment
76          where
77            parsePrint        = do {  match "printski(";
78                                      body <- parseExpression;
79                                      match ");";
80                                      return (Print body) }
81            parseIf           = do {  match "ifski";
82                                      prop <- parseExpression;
83                                      match "thenski";
84                                      token '{';
85                                      thenSt <- multipleStatementsParser;
86                                      token '}';
87                                      match "elski";
88                                      token '{';
89                                      elseSt <- multipleStatementsParser;
90                                      token '}';
91                                      return (If prop thenSt elseSt) }
92            parseWhile        = do {  match "whileski";
93                                      prop <- parseExpression;
94                                      match "doski{";
95                                      body <- multipleStatementsParser;
96                                      token '}';
97                                      return (While prop body) }
98            parseAssignement  = do {  name <- parseWords;
99                                      match ":=";
100                                     exp <- parseExpression;
101                                     token ';';
102                                     return (Assignement name exp) }
103           parseDrive        = do {  match "driveski";
104                                     sp <- parseExpression;
105                                     token ';';
106                                     return (Drive sp)
107                                     }
108           parseTurn         = do {  match "turnski";
109                                     dir <- parseDirection;
110                                     expr <- parseExpression;
111                                     token ';';
112                                     return (Turn dir expr)}
113           parseLight        = do {  match "ledski";
114                                     token '(';
115                                     d1 <- parseExpression;
116                                     token ',';
117                                     d2 <- parseExpression;
```

```
118                                     token ',';
119                                     d3 <- parseExpression;
120                                     token ',';
121                                     d4 <- parseExpression;
122                                     match ");";
123                                     return (Lights d1 d2 d3 d4)}
124             parseComment      = do {  match "--";
125                                     com <- parseExpression;
126                                     match "--";
127                                     return Comment
128                                     }
129 parseExpression :: Parser Expression
130 parseExpression = parseDec `mplus` parseIdent `mplus` parseFalse `mplus`
      parseTrue `mplus` parseLit `mplus` parseAdd `mplus` parseMul `mplus`
      parseDiv `mplus` parseSub `mplus` parseEq `mplus` parseGreater `mplus`
      parseLesser `mplus` parseAnd `mplus` parseNot `mplus` parseOr `mplus`
      parseReadSensor `mplus` parseLightSensor
131           where
132             parseDec    = do { d <- parseDouble;
133                                return (Decimal d)}
134             parseIdent  = do { n <- parseWords;
135                                return (Var n)}
136             parseLit    = do { n <- parseInt;
137                                return (Con n) }
138             parseTrue   = do{  match "#true";
139                                return (BoolVal True)}
140             parseFalse  = do { match "#false";
141                                return (BoolVal False)}
142             parseAdd    = do { token '(';
143                                d <- parseExpression;
144                                token '+';
145                                e <- parseExpression;
146                                token ')';
147                                return (Add d e) }
148             parseMul    = do { token '(';
149                                d <- parseExpression;
150                                token '*';
151                                e <- parseExpression;
152                                token ')';
153                                return (Mul d e) }
154             parseDiv    = do { token '(';
155                                d <- parseExpression;
156                                token '/';
157                                e <- parseExpression;
158                                token ')';
159                                return (Div d e) }
160             parseSub    = do { token '(';
161                                d <- parseExpression;
162                                token '-';
163                                e <- parseExpression;
164                                token ')';
165                                return (Sub d e) }
166             parseEq     = do { token '(';
167                                left  <- parseExpression;
168                                match "==";
169                                right <- parseExpression;
170                                token ')';
```

```
171                                 return (Eq left right) }
172             parseGreater = do {   token '(';
173                                   left  <- parseExpression;
174                                   token '>';
175                                   right <- parseExpression;
176                                   token ')';
177                                   return (Greater left right) }
178             parseLesser  = do {   token '(';
179                                   left  <- parseExpression;
180                                   token '<';
181                                   right <- parseExpression;
182                                   token ')';
183                                   return (Lesser left right) }
184             parseAnd = do {       token '(';
185                                   left  <- parseExpression;
186                                   match "&&";
187                                   right <- parseExpression;
188                                   token ')';
189                                   return (And left right) }
190             parseOr = do {        token '(';
191                                   left  <- parseExpression;
192                                   match "||";
193                                   right <- parseExpression;
194                                   token ')';
195                                   return (Or left right) }
196             parseNot = do {       match "~";
197                                   prop <- parseExpression;
198                                   return (Not prop) }
199             parseReadSensor = do {match "#sensorski";
200                                   return ReadSensor}
201             parseLightSensor = do { match "#lightski";
202                                     dir <- parseDirection;
203                                     return (LightSensor dir)}
```

State Monad

```
1  module StateMonad (fetch, write, Env, Environment) where
2
3  import          Parser
4  import          Datatypes
5  import          Extended_Parser
6
7
8  import Control.Monad (void)
9  import Control.Monad.IO.Class
10 import Control.Monad.Trans.State.Lazy (State, StateT(..), get, put)
11 import Data.Map as Map
12 import Data.Maybe
13
14 type Environment = Map.Map Words Value
15
16 type Env a = StateT Environment IO a
17
18 fetch :: String -> Env Value
19 fetch k = do {  envir <- get;
20                return $ fromJust $ Map.lookup k envir}
21
22 write :: Words -> Value -> Env()
```

```
23 write n v   = do {  env <- get;
24                     put $ Map.insert n v env;
25                     return ()}
```

Interpreter

```
1  import StateMonad
2  import Extended_Parser
3  import Parser
4  import Evaluator
5
6  import Data.Map (empty)
7  import Simulator
8
9  import System.Environment (getArgs)
10 import Control.Monad.State.Lazy (runState)
11 import Control.Monad.Trans.State.Lazy (State, StateT(..), get, put, modify)
12
13 main :: IO ()
14 main = do   filePath <- getArgs;
15            content <- readFile $ head filePath;
16            let stmts = parse multipleStatementsParser (prepare content);
17            let e = Data.Map.empty;
18            runStateT (execute stmts) e;
19            return ()
20
21 prepare:: String -> String
22 prepare = filter (`notElem` "\t\n\r ")
```