Project Functioneel Programmeren

Maximiliaan Leyman
2e zittijd – enkel deel “Programmeertaal”

Inleiding:

De programmeertaal is gebouwd met Java en Python als voorbeeld, maar vooral rond de kern van het besturen van de Mbot. De programmeertaal heb ik Hasky genoemd, naar de naam van mijn robotje en omdat het een mini-taaltje is gemaakt met Haskell. Met veel symbolen kun je min of meer gewoon doen wat je verwacht dat ze zouden doen. + telt op, & is een logische AND, enzovoort.

Syntax:

Nmr ::= Int | Var Name | Nmr+Nmr | Nmr-Nmr | Nmr*Nmr
Con ::= Bool | not Con | Num(<=/</>=/>/==)Num | Con&Con | Con|Con
Cmd ::= Wait Int; | Call Name Int; | if(Con){Pgm} | while(Con){Pgm} | roboCmd; | #Comment#
roboCmd ::= Turn_left Nmr | Turn_right Nmr | Forward Nmr | Stop
| Lamp Nmr Nmr Nmr Nmr | Light Name | Dist Name

Pgm ::= [Cmd]
Bool ::= True | False
Int ::= Integers
Name ::= String, naam voor variabelen in de environment (Int)

Semantiek:

Hasky op zich bestaat uit een series commandos, Cmd, samengevoegd in blokken, Pgms, omgeven door {} accolades. De parser verwijdert op voorhand alle newlines en tabs, zodat deze vrij gebruikt kunnen worden om de code overzichtelijker te maken.

Nmr stellen de numerieke waarden voor in Hasky – dit is ofwel een literal Int, een Name, uitgedrukt als 'Var <String>', die verwijst naar een getalvariabele in de environment, of een uitdrukking met de operatoren (<=, <, >=, >, ==). Uitdrukkingen worden omgeven door ronde haakjes ( ), zonder spaties. Eender waar je in Hasky-code een gewoon nummer zou zetten, kun je in de plaats ook een expressie zetten, die dan ge-evalueerd wordt tot een uiteindelijke literal. Nmr voorbeeld: “1+(2+3)”.

Con stellen logische waarden voor in Hasky. Deze kunnen een booleaanse waarde, True of False aannemen, ofwel de vorm van een logische uitdrukking. Net zoals hierboven moeten deze in ronde haakjes gezet worden. De beschikbare logische uitdrukkingen zijn een logische AND met het teken '&', een logische OR met het teken '|', een logische not door middel van het woord “not”, of een vergelijking van twee Nmr getallen met '<='. D.m.v. '&' en 'not' kunnen hiermee alle andere vergelijkingen gemaakt worden. Con voorbeeld: “True&(2 <= 3)”
Bij de con-uitdrukkingen zet je in tegenstelling tot de Nmr-uitdrukkingen wel spaties. Dit leek mij natuurlijker voor logische expressies, en idealiter zouden beiden mogelijk moeten zijn voor beiden.

Cmd zijn de commandos, statements, waaruit Hasky opgebouwd is. Een Cmd kan “wait <Nmr>;” zijn, hetwelke niks doet en gewoon wacht. “call <Name> <Int>;” voegt een getal met de waarde van de Int toe aan de omgeving onder de naam Name. Een roboCmd stelt een stelt een commando voor de robot voor, verder uitgebreider uitgelegd. Deze drie commando's – namelijk, die die geen

functieblok als argument meevragen, moeten afgesloten worden met een ;.
Cmd's worden gebundeld in blokken die Pgm's heten. Het commando if(<Con>){Pgm} evalueert de Con die tussen de haakjes bij 'if' staat, en voert het Pgm uit als de conditie True evalueert. Het commando while(<Con>){Pgm} is analoog, maar dit blok wordt herhaald totdat de conditie False evalueert. Merk op dat vergelijkende Con's zelf omgeven moeten zijn door haakjes, en er dus twee paar moeten staan – een van de if-clause en een van de Con zelf.
Een Hasky-programma bestaat uit 1 groot Pgm, opgebouwd uit de verschillende Cmds (die mogelijks dan zelf kleinere Pgms als argumenten meekrijgen).

Voor roboCmd zijn er 7 mogelijke opties.
"turn_left", "turn_right", "forward" en "stop" spreken voor zich. De eerste drie nemen mogelijks een 'Nmr' mee om de snelheid van de motoren aan te duiden. Als er geen Nmr meegegeven wordt (dus bv. gewoon "Turn_left;") wordt de waarde automatisch op 150 gezet.
"lamp" neemt vier Nmr argumenten ("lamp <Nmr> <Nmr> <Nmr> <Nmr>") die, zoals in de Mbot bibliotheek, respectievelijk het nummer van de lamp en de drie RGB waarden aangeven waar de lamp op gezet moet worden.
"light" en "dist" nemen elk een name argument ("light <Name>" en "dist <Name>"), waarin het resultaat opgeslagen zal worden. Voor light geeft dit een "0" terug als beiden zwart zijn, een "1" als enkel links zwart is, een "2" als enkel rechts zwart is, en een "3" als geen enkele zwart is. De waarde voor "dist" wordt afgerond naar de dichtstbijzijnde Int.
Met behulp van het commando "wait <Nmr>" kan men het programma laten wachten voor een bepaalde tijdsduratie, in tienden van een seconde.
Verder kunnen tussen de commandos in de code ook commentaarblokken gezet worden. Een commentaarblok wordt begonnen met #, en mag niet halverwege een bestaand command staan. Alles binnen de commentaar, tot je terug een # zet, wordt volledig genegeerd door de interpreter.

Voorbeelden:

Police:

```
{
while(True){
        lamp 1 255 0 0;
        lamp 2 0 0 255;
        wait 5;
        lamp 1 0 0 255;
        lamp 2 255 0 0;
        wait 5;
}
}
```

Avoid:

```
{
call grens 30;
while(True){
        dist see;
        if((Var see <= Var grens)){
                turn_right;
        }
        if((Var grens <= Var see)){
                forward;
        }
        wait 1;
```

```
}
}
```

Line:

```
{
while(True){
	#wait 1;#
	light in;
	if((Var in == 0)){
		forward 80;
	}
	if((Var in == 1)){
		turn_left 80;
	}
	if((Var in == 2)){
		turn_right 80;
	}
	if((Var in == 3)){
		stop;
	}
}
}
```

Opmerking: De test die ik thuis uitgevoerd heb met Line was met nogal een dunne zwarte strook, die hij moeite had om te zien. De snelheid van de motoren vertragen helpt hierbij – maar ahankelijk van hoe snel de robot rijdt, moet de zwarte strook dus breed genoeg zijn. Een motorsnelheid van ongeveer 80, met draaien d.m.v. een motor i.p.v. twee, slaagt er in redelijk dunne lijnen wel redelijk mooi te volgen.

Implementatie:

De code voor Hasky bestaat uit twee grote delen – de parse-module en de interpreter-module. De main-methode die het hele programma doet lopen, staat in Interpreter.hs.
De parse-module is opgedeeld in parsers voor de verschillende datatypes. De parser-file voor elk datatype bevat de definitie van het datatype, een eventuele evaluator voor dit datatype alsook de parser voor dit datatype. De parsers bestaan op zich uit enkele kleine parser-modules, en worden samengehecht d.m.v. `mplus` uit monadplus om uiteindelijk een grote parser voor programmas te bekomen. De finale pgmParser parst geen commandos, maar programmas – het resultaat van de parse is een lijst van commandos die dan sequentieel door de uitvoeringsmodule uitgevoerd kunnen worden. De file met basisfuncties Parser.hs is grotendeels zoals in de les gezien, maar er zijn enkele toevoegingen bij gemaakt.
Bij NmrParser is ook het type Env gedefinieerd – dit stelt een lijst voor van paren van names – strings – en de bijhorende getallen.
Ook heb ik een kleine file, parsetest.hs gemaakt, om de parsing van Hasky programmatjes te testen. Als in de main van de interpreter het parsen faalt, geeft hij een exception, en blijft de robot openstaan. Dan moet telkens GHCi afgesloten en terug opgestart worden opdat de verbinding met de robot terug opnieuw gemaakt kan worden, wat redelijk ergerlijk is. Parsetest laat toe om gewoon de parse van een file te testen zonder die al te proberen uitvoeren.

Het programma wordt gekozen door helemaal bovenaan in de main, de string aan te passen bij readFile.
De mainmethode van Interpreter.hs leest het programma uit de externe file in, en opent daarna een Mbot. Omdat de State die we meenemen zeer ingewikkeld zou worden moest de bot daar ook ingestoken moeten worden, is runPgm zo gedefinieerd dat je het Device van de robot er aan meegeeft.
Met behulp van de rTnN functie worden eerst alle tabs en newlines (inclusief Windows newlines) uit de code gehaald. Hierna parset de parser het text-blok naar een array van Cmd's – een Pgm – die dan samen met de bot meegegeven wordt aan runPgm. runPgm geeft een StateT Env IO () terug – die, voor gebruiksgemak, ook EnvIO () genoemd wordt. Deze wordt dan geevalueerd met evalStateT op een initieel lege array om terug bij een IO () te geraken, die dan aan de mainmethode meegegeven wordt om het programma uit te voeren. Uiteindelijk wordt de bot nog gesloten.
(Opmerking: Indien je een programma schrijft

De basisopbouw van de verschillende commandos is als volgt. Alle commando's krijgen ook de bot meegegeven zodat runRobo relatief simpel blijft, maar niet alle commando's gebruiken hem.

“Wait Nmr” returnt een EnvIO die niets teruggeeft en Haskell een aantal tienden van seconden doet wachten, afhankelijk van het meegegeven getal dat naar een “Nmr” geparset wordt. Voor het besturen van de robot leek tienden van een seconde mij een mooie tijdsduratie, maar dit kan gemakkelijk aangepast worden in de handler van wait.

Comment returnt een EnvIO die gewoon niets doet (tenzij feedback printen).

“Call Name Nmr” haalt de state op m.b.v. 'get' en voegt hieraan een nieuw element toe met de gegeven naam en waarde. Dit gebeurt m.b.v. de addToEnv functie uit Evaluator.hs. Als er al een variabele bestaat met de opgegeven name, wordt deze overschreven. (Om deze variabele later terug te gebruiken in andere commandos, roep je hem op in Hasky met de expressie “Var <naam van de variabele>”
If en While halen beiden de sate op, en gebruiken deze dan om de meegegeven expressie te evalueren naar een uiteindelijke “True” of “False”. Als ze False is, gebeurt er niets. Als ze True is, dan wordt runPgm recursief opgeroepen op de commandos in het Pgm-blok dat erop volgt. While roept zichzelf nadien terug op met dezelfde logische expressie en hetzelfde commandoblok – als de expressie waar was. Zelf beinvloeden ze de state niet. Het resultaat is een EnvIO die al de effecten van de uiteindelijk bekomen commandos (of dus gewoon niets, als de uitdrukking false was) geconcateneerd is.

De Robocmd staan reprogrammeerd als een “Robo Robocmd” commando, dat gewoon de EnvIO teruggeeft van het bijhorende robotcommando, dat in runRobo geimplementeerd staat.

Turn_Left, Turn_Right en Forward geven een EnvIO terug die de robot naar links of naar rechts laten draaien. De snelheid staat vastgesteld in de functie, maar is wel gemakkelijk aan te passen. Het Stop-commando geeft een EnvIO terug die de robot compleet stopt.

“Lamp Nmr Nmr Nmr Nmr” krijgt vier Nmr's mee, en geeft een EnvIO terug die het commando naar de robot stuurt om de lamp op de bijhorende staat te zetten. De Nmrs worden geevalueerd in de context van de state uit EnvIO.

“Light Name” haalt de waarde van de lichtsensor op en houdt die bij in de variabele bij de meegegeven Name. Hij roept de hulpfunctie lineToInt op om het Line-object om te zetten in een int (met waarde zoals hierboven beschreven) om ze te kunnen bijhouden in de int-gebaseerde

environment.

“Dist Name” werkt hetzelfde als Light, en houdt de resulterende waarde bij in de environment. Omdat de environment met Ints werkt, moet deze afgerond worden naar de dichtstbijzijnde Int, maar door in deze functie een vermenigvuldiging door te voeren voor het afronden kan men een zo hoog nodige precisie bekomen voor de sensor.

Conclusie:

Er zijn zeker nog dingen die beter kunnen aan Hasky. Een Env die werkt met Floats/Doubles zou beter zijn – dan zouden delingen ook gemakkelijker te ondersteunen zijn. Ten slotte zou de parser nog iets beter kunnen zijn, onder andere met meer tolerantie voor willekeurig geplaatste spaties en puntkommatekens achter commandos, alsook meer tolerantie rond het plaatsen en weglaten van haakjes.
De taal is echter op zich mooi functioneel, en qua besturing van de robot kunnen het grootste deel van de simpele programma-tjes waar ik mee kon opkomen, geimplementeerd worden – onder andere zeker de drie basisprogrammas.

Appendix broncode:

```haskell
-- Interpreter.hs
-- Maximiliaan Leyman

import CmdParser (Env, parseCmd, parsePgm, Cmd (..), Robocmd (..)   )
import Parser
import NmrParser (Nmr (..), Name)
import ConParser (Con)
import Evaluator (evalCon, evalNmr, addToEnv)
import Control.Monad
import Control.Monad.Trans.State
import Control.Monad.IO.Class
import Control.Concurrent
import MBot
import System.HIDAPI

-- De main-methode bevat de daadwerkelijke uitvoering van een programma. Deze leest
het Hasky-programma uit een textfile, opent de robot,
--   parset en voert het programma uit, en sluit de robot dan weer.
-- Het programma om uit te voeren kan aangepast worden door de string bij readFile
aan te passen.

main = do {
    readpgm <- readFile "avoid.txt";
    bot <- openMBot;
    evalStateT (runPgm (parse parsePgm (rTnN readpgm)) bot) [];
    closeMBot bot
}

-- Remove tabs and Newlines
rTnN :: String -> String
rTnN ('\t':rest) = rTnN rest
rTnN ('\r':rest) = rTnN rest
rTnN ('\n':rest) = rTnN rest
rTnN (x:rest)    = x:rTnN rest
rTnN []          = []

-- EnvIO is een monad - de combinatie van de State en IO monads d.m.v. StateT.
-- De bijgehouden state is een Env, i.e. een array van ints.
type EnvIO a = StateT Env IO a

-- runPgm geeft de EnvIO terug die de uitvoering van het meegegeven programma
voorstelt. Door dit dan met evalStateT te evalueren en mee te geven
--  aan de main-functie wordt het programma dan uiteindelijk uitgevoerd.
-- runPgm wordt ook gebruikt om sub-blokken uit te voeren. De resulterende EnvIO
kan dan samengebind worden met andere commando's/programmas tot het uitendelijke
volledige programma.
runPgm :: [Cmd] -> Device -> EnvIO ()
runPgm [] bot = return ();
runPgm (x:xs) bot = do { a <- runCmd x bot;
                         runPgm xs bot;
                         }
```

```haskell

-- runCmd geeft de EnvIO terug die de uitvoering voorstelt van een enkel commando.
De implementatie is anders voor elk commando.
-- Om het volgen van het programma te vergemakkelijken en inzicht te geven in de
flow van het programma, printen meeste commandos ook een statement naar de console
uit om te volgen.
runCmd :: Cmd -> Device -> EnvIO ()
runCmd (Wait a) bot = do{
    liftIO (print "Waiting");
    lst <- get;
    liftIO (threadDelay (100000 * evalNmr a lst))
}
runCmd Comment bot  = liftIO (print "There is a comment here")
runCmd (Call a b) bot = do {
    liftIO (print "Call statement");
    lst <- get;
    put (addToEnv a (Lit (evalNmr b lst)) lst);
    lst2 <- get;
    liftIO (print lst2);
    }
runCmd (Check a b) bot = do {
    e <- get;
    liftIO (print "Check statement");
    liftIO (print "Current env state:");
    liftIO (print e);
    liftIO (print "Condition to evaluate:");
    liftIO (print a);
    when (evalCon a e) $ runPgm b bot
}
runCmd (While a b) bot = do {
    e <- get;
    when (evalCon a e) $ do {runPgm b bot;runCmd (While a b) bot;}
}
runCmd (Robo a) bot = runRobo a bot

-- runRobo stelt de uitvoering van een Robocmd voor, een commando dat interreageert
met de robot.
-- De robot draait standard met een wiel vooruit en een wiel uit, maar dit kan
redelijk gemakkelijk aangepast worden
--   door de 0 te vervangen door het tegengestelde van de expressie ernaast.
runRobo :: Robocmd -> Device -> EnvIO ()
runRobo (TurnLeft a) bot    = do{
    lst <- get;
    liftIO (print "Robot turning left");
    liftIO (sendCommand bot $ setMotor 0 (evalNmr a lst))
}
runRobo (TurnRight a) bot   = do{
    lst <- get;
    liftIO (print "Robot turning right");
    liftIO (sendCommand bot $ setMotor (evalNmr a lst) 0)
```

```
}
runRobo (Forward a) bot      = do{
    lst <- get;
    liftIO (print "Robot going forward");
    liftIO (sendCommand bot $ setMotor (evalNmr a lst) (evalNmr a lst))
}
runRobo Stop bot         = do{
    liftIO (print "Robot stopping");
    liftIO (sendCommand bot $ setMotor 0 0)
}
runRobo (Lamp a b c d) bot = do{
    lst <- get;
    liftIO (sendCommand bot $ setRGB (evalNmr a lst) (evalNmr b lst) (evalNmr c
lst) (evalNmr d lst))
}
runRobo (Light a) bot      = do {
    liftIO (print "Reading the line sensor");
    lin <- liftIO (readLineFollower bot);
    runCmd (Call a (Lit (lineToInt lin))) bot
}
runRobo (Dist a) bot      = do{
    liftIO (print "Reading the ultrasonic sensor");
    val <- liftIO (readUltraSonic bot);
    runCmd (Call a (Lit (round val))) bot
}

-- Deze hulpfunctie neemt het Line-object dat gegeven wordt door het
readLineFollower-commando van de bot en zet het om naar een integer die wij kunnen
bijhouden in onze Env.
lineToInt :: Line -> Int
lineToInt BOTHB = 0
lineToInt LEFTB = 1
lineToInt RIGHTB = 2
lineToInt BOTHW = 3


-- parsetest.hs
-- Maximiliaan Leyman

import CmdParser (Env, parseCmd, parsePgm, Cmd (..), Robocmd (..)   )
import Parser
import NmrParser (Nmr (..), Name)
import ConParser (Con)
import Evaluator (evalCon, evalNmr, addToEnv)
import Control.Monad
import Control.Monad.Trans.State
import Control.Monad.IO.Class


-- Dit extra bestand dient om de parsing van een Hasky-programma te testen, zonder
dit daadwerkelijk op de robot uit te voeren.
```

```
-- Dit is vooral handig omdat de robot blokkeert als je (Hasky-)programma niet
parset (en dus de main-methode uit Interpreter.hs geen programma heeft om uit te
voeren), want de exception
--  hiervan zorgt ervoor dat de robot nooit gesloten wordt, en het dan nodig is
GHCi te herstarten om verder te kunnen.

main = do {
    readpgm <- readFile "pgm4.txt";
    print (show (apply parsePgm (rTnN readpgm)));

}

-- Remove tabs and Newlines
rTnN :: String -> String
rTnN ('\t':rest) = rTnN rest
rTnN ('\r':rest) = rTnN rest
rTnN ('\n':rest) = rTnN rest
rTnN (x:rest)    = x:rTnN rest
rTnN []          = []


-- Evaluator.hs
-- Maximiliaan Leyman

module Evaluator
( evalNmr
, evalCon
, addToEnv
) where

import Parser
import MonadPlus
import NmrParser (Nmr (..), Env, Name)
import ConParser (Con (..))


-- Deze functie zoekt een nummer op in de Env
findvar :: Name -> Env -> Nmr
findvar a ((n,l):xs)    = if a == n then l else findvar a xs
findvar a []            = Lit 0

addToEnv :: Name -> Nmr -> Env -> Env
addToEnv a v ((n,l):xs)   = if a == n then (n,v):xs else (n,l):addToEnv a v xs
addToEnv a v []           = [(a,v)]

-- Deze functie evalueert een "Nmr" tot een Haskell int
evalNmr :: Nmr -> Env -> Int
evalNmr (Lit n)     e   = n
evalNmr (a :+: b)   e   = evalNmr a e + evalNmr b e
evalNmr (a :-: b)   e   = evalNmr a e - evalNmr b e
evalNmr (a :*: b)   e   = evalNmr a e * evalNmr b e
```

```
evalNmr (a :/: b)   e   = evalNmr a e `quot` evalNmr b e
evalNmr (Var n)     e   = evalNmr (findvar n e) e

-- Deze functie evalueert een "Con" tot een Haskell Bool
evalCon :: Con -> Env -> Bool
evalCon (Boolean x) e   = x
evalCon (Inv x)     e   = not (evalCon x e)
evalCon (a :&: b)   e   = evalCon a e && evalCon b e
evalCon (a :|: b)   e   = evalCon a e || evalCon b e
evalCon (a :<=: b)  e   = evalNmr a e <= evalNmr b e
evalCon (a :<: b)   e   = evalNmr a e <  evalNmr b e
evalCon (a :>=: b)  e   = evalNmr a e >= evalNmr b e
evalCon (a :>: b)   e   = evalNmr a e >  evalNmr b e
evalCon (a :==: b)  e   = evalNmr a e == evalNmr b e


-- CmdParser.hs
-- Maximiliaan Leyman

module CmdParser
( Cmd (..)
, Robocmd (..)
, Env
, parseCmd
, parsePgm
) where

import Parser
import MonadPlus
import ConParser
import NmrParser


-- Dit gegevenstype stelt de verschillende commandos in Hasky voor
data Cmd    = Wait Nmr          -- Wacht Nmr seconden
            | Comment           -- Een comment - doet niks
            | Call Name Nmr     -- Houd het meegegeven Nmr bij als variabele met
als naam Name
            | Check Con [Cmd]   -- Voert het [Cmd] blok uit als Con 'True'
evalueert
            | While Con [Cmd]   -- Voert het [Cmd] blok uit zolang Con 'True'
evalueert
            | Robo Robocmd      -- Voert het robot-commando Robocmd uit
            deriving (Eq, Show)

-- Dit zijn de commandos die daadwerkelijk interreageren met de robot.
data Robocmd    = TurnLeft Nmr
                | TurnRight Nmr
                | Forward Nmr
                | Stop
                | Lamp Nmr Nmr Nmr Nmr
```

```
252                 | Light Name
253                 | Dist Name
254                 deriving (Eq, Show)
255
256 -- Parse een enkel Hasky-commando
257 parseCmd :: Parser Cmd
258 parseCmd = parseWait `mplus` parseCall
259                      `mplus` parseCheck
260                      `mplus` parseWhile
261                      `mplus` parseRobo
262                      `mplus` parseComment
263     where
264     parseWait = do { match "wait ";           -- Parse een 'wait' commando
265                      a <- parseNmr;
266                      token ';';
267                      return (Wait a) }
268     parseCall = do { match "call ";           -- Parse een 'call' commando
269                      a <- parseWord;
270                      token ' ';
271                      b <- parseNmr;
272                      token ';';
273                      return (Call a b) }
274     parseCheck = do { match "if(";            -- Parse een 'if' commando
275                       a <- parseCon;    -- De benaming is hier 'check' om
276 verwarring te vermijden, aangezien zowel Haskell als mijn programmeertaal al 'if'
277 in de code hebben.
278                       token ')';
279                       b <- parsePgm;
280                       return (Check a b) }
281     parseWhile = do { match "while(";         -- Parse een 'while' commando
282                      a <- parseCon;      -- Hlint meldt dat duplication hier
283 vermeden kan worden met Check - dit is waar, maar zou de structuur een beetje
284 verbrodden en het
285                      token ')';          -- ingewikkelder maken dan om dit gewoon te
286 laten staan.
287                      b <- parsePgm;
288                      return (While a b) }
289     parseComment = do { token '#';                -- Parse een comment line
290                         plus (spot (/= '#'));
291                         token '#';
292                         return Comment }
293
294 -- Parse een commando dat met de robot interreageert.
295 parseRobo :: Parser Cmd
296 parseRobo = parseLeft `mplus` parseRight
297                       `mplus` parseFwd
298                       `mplus` parseStop
299                       `mplus` parseLamp
300                       `mplus` parseLight
301                       `mplus` parseDist
302                       `mplus` parseLeftAmount
```

```
                        `mplus` parseRightAmount
                        `mplus` parseFwdAmount
    where
    parseLeft   = do { match "turn_left;";
                       return (Robo (TurnLeft (Lit 150))) }
    parseRight  = do { match "turn_right;";
                       return (Robo (TurnRight (Lit 150))) }
    parseFwd    = do { match "forward;";
                       return (Robo (Forward (Lit 150))) }
    parseStop   = do { match "stop;";
                       return (Robo Stop) }
    parseLamp   = do { match "lamp ";
                       a <- parseNmr;
                       token ' ';
                       b <- parseNmr;
                       token ' ';
                       c <- parseNmr;
                       token ' ';
                       d <- parseNmr;
                       token ';';
                       return (Robo (Lamp a b c d)) }
    parseLight  = do { match "light ";
                       a <- parseWord;
                       token ';';
                       return (Robo (Light a)) }
    parseDist   = do { match "dist ";
                       a <- parseWord;
                       token ';';
                       return (Robo (Dist a)) }
    parseLeftAmount = do { match "turn_left ";
                           a <- parseNmr;
                           token ';';
                           return (Robo (TurnLeft a)) }
    parseRightAmount = do { match "turn_right ";
                           a <- parseNmr;
                           token ';';
                           return (Robo (TurnRight a)) }
    parseFwdAmount  = do { match "forward ";
                           a <- parseNmr;
                           token ';';
                           return (Robo (Forward a)) }

-- Parse een Hasky programma-blok, met accolades als delimiters
parsePgm :: Parser [Cmd]
parsePgm = do{  token '{';
                res <- plus parseCmd;
                token '}';
                return res }


-- NmrParser.hs
```

```haskell
-- Maximiliaan Leyman

module NmrParser
( Nmr (..)
, Name
, Env
, parseNmr
) where

import Parser
import MonadPlus

type Name = String
type Env = [(Name, Nmr)]

-- Dit gegevenstype stelt de verschillende vormen van numerieke expressie in Hasky
voor
data Nmr = Lit Int
         | Var Name
         | Nmr :+: Nmr
         | Nmr :-: Nmr
         | Nmr :*: Nmr
         | Nmr :/: Nmr
         deriving (Eq, Show)

-- Parse een Hasky Nmr expressie
parseNmr :: Parser Nmr
parseNmr = parseLit `mplus` parseAdd `mplus` parseSub `mplus` parseMul `mplus`
parseDiv `mplus` parseVar
    where
        parseLit = do { n <- parseInt;  -- Parse een literal
                        return (Lit n) }
        parseAdd = do { token '(';      -- Parse een optelling
                        a <- parseNmr;
                        token '+';
                        b <- parseNmr;
                        token ')';
                        return (a :+: b) }
        parseSub = do { token '(';      -- Parse een aftrekking
                        a <- parseNmr;
                        token '-';
                        b <- parseNmr;
                        token ')';
                        return (a :-: b) }
        parseMul = do { token '(';      -- Parse een vermenigvuldiging
                        a <- parseNmr;
                        token '*';
                        b <- parseNmr;
                        token ')';
                        return (a :*: b) }
        parseDiv = do { token '(';      -- Parse een deling
```

```
                        a <- parseNmr;
                        token '/';
                        b <- parseNmr;
                        token ')';
                        return (a :/: b) }
        parseVar = do { match "Var ";  -- Parse een variabele en zijn naam
                        a <- parseWord;
                        return (Var a) }



-- ConParser.hs
-- Maximiliaan Leyman

module ConParser
( Con (..)
, parseCon
) where

import Parser
import MonadPlus
import NmrParser
import Debug.Trace

-- Dit gegevenstype stelt de verschillende vormen van logische expressies in Hasky
voor
data Con = Boolean Bool
         | Inv Con
         | Con :&: Con
         | Con :|: Con
         | Nmr :<=: Nmr
         | Nmr :<: Nmr
         | Nmr :>=: Nmr
         | Nmr :>: Nmr
         | Nmr :==: Nmr
        deriving (Eq, Show)

-- Parse een Hasky Con-expressie
parseCon :: Parser Con
parseCon = parseBool `mplus` parseNot `mplus` parseAnd `mplus` parseOr `mplus`
parseCmp
    where
    parseNot = do { token '(';          -- Parse een 'not' expressie
                    match "not ";
                    a <- parseCon;
                    token ')';
                    return (Inv a) }
    parseAnd = do { token '(';          -- Parse een 'and' expressie
                    a <- parseCon;
                    match " & ";
                    b <- parseCon;
```

```
                    token ')';
                    return (a :&: b) }
    parseOr  = do { token '(';          -- Parse een 'or' expressie
                    a <- parseCon;
                    match " | ";
                    b <- parseCon;
                    token ')';
                    return (a :|: b) }

parseCmp :: Parser Con
parseCmp = parseLte `mplus` parseLt `mplus` parseGte `mplus` parseGt `mplus`
parseEq
    where
    parseLte =  do { token '(';          -- Parse een vergelijking
                    a <- parseNmr;
                    match " <= ";
                    b <- parseNmr;
                    token ')';
                    return (a :<=: b) }
    parseLt =   do { token '(';
                    a <- parseNmr;
                    match " < ";
                    b <- parseNmr;
                    token ')';
                    return (a :<: b) }
    parseGte =  do { token '(';
                    a <- parseNmr;
                    match " >= ";
                    b <- parseNmr;
                    token ')';
                    return (a :>=: b) }
    parseGt =   do { token '(';
                    a <- parseNmr;
                    match " > ";
                    b <- parseNmr;
                    token ')';
                    return (a :>: b) }
    parseEq =   do { token '(';
                    a <- parseNmr;
                    match " == ";
                    b <- parseNmr;
                    token ')';
                    return (a :==: b) }


parseBool :: Parser Con
parseBool = parseTrue `mplus` parseFalse
    where
        parseTrue   = do { match "True";
                           return (Boolean True) }
        parseFalse  = do { match "False";
```

```
                            return (Boolean False) }



-- Parser.hs
-- Maximiliaan Leyman
-- Op basis van parser gezien in de les door Christophe Scholliers

module Parser
( Parser
, apply
, parse
, char
, spot
, token
, match
, star
, plus
, parseInt
, parseWord
) where

import MonadPlus
import Data.Char (isDigit)

-- Contains basic parser functionality necessary for the parsing of simple
characters, numbers and strings, upon which the other types of parser will build
(given the input will will consist entirely of text to be parsed).

-- Parser type definition
newtype Parser a = Parser (String -> [(a, String)])

-- Making Parser a functor
instance Functor Parser where
   fmap f (Parser p) = Parser (\s -> [(f a, b) | (a, b) <- p s])

-- Making Parser an applicative functor
instance Applicative Parser where
   pure  = return
   (Parser p1) <*> (Parser p2) = Parser (\s -> [(f a, s2) | (f, s1) <- p1 s, (a,
s2) <- p2 s1])

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

-- Making Parser an instance of monadplus
```

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

-- Apply the parser (returns pair of parsed value and remaining string)
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"

-- Create a parser to parse one character
char :: Parser Char
char = Parser f
  where
  f []     = []
  f (c:s)  = [(c,s)]

-- Parse (match?) a character statisfying a given condition
spot :: (Char -> Bool) -> Parser Char
spot p = do { c <- char; guard (p c); return c }

-- Create a parser to match a given character
token :: Char -> Parser Char
token c = spot (== c)

-- Create a parser to match a given string
--match :: String -> Parser String
--match []     = return []
--match (x:xs) = do {
--                 y <- token x;
--                 ys <- match xs;
--                 return (y:ys)
--               }
match :: String -> Parser String
match = mapM token

-- Parsing sequences
-- Create a parser to match zero or more occurences (of a given parser match)
star :: Parser a -> Parser [a]
star p = plus p `mplus` return []

-- Create a parser to match one or more occurences (of a given parser match)
plus :: Parser a -> Parser [a]
plus p = do x <- p
            xs <- star p
```

```
609             return (x:xs)
610
611 -- Parsing numbers
612 -- Create a parser to match a natural number
613 parseNat :: Parser Int
614 parseNat = do s <- plus (spot isDigit)
615               return (read s)
616
617 -- Create a parser to match a negative number
618 parseNeg :: Parser Int
619 parseNeg = do token '-'
620               n <- parseNat
621               return (-n)
622
623 -- Match an integer
624 parseInt :: Parser Int
625 parseInt = parseNat `mplus` parseNeg
626
627 -- Match a single word
628 parseWord :: Parser String
629 parseWord = plus (spot (\s -> (s /= ' ') && (s /= ')') && (s /= ';')));
630
631
632 -- MonadPlus.hs
633 -- Maximiliaan Leyman
634 -- GHCi vond MonadPlus elders precies niet, dus heb ik het hier zelf ingezet om hem
635 te kunnen gebruiken.
636
637 module MonadPlus
638 ( MonadPlus
639 , mzero
640 , mplus
641 , guard
642 ) where
643
644 class Monad m => MonadPlus m where
645    mzero :: m a
646    mplus :: m a -> m a -> m a
647
648 instance MonadPlus [] where
649     mzero = []
650     mplus = (++)
651
652 guard :: MonadPlus m => Bool -> m ()
653 guard False = mzero
654 guard True  = return ()
655
```