Cursul 12 - Happy - generator de parser

- http://www.haskell.org/happy/

Ce este un generator de parser?

- Yacc
  - Intrare: fisier ce contine descrierea sintaxei cu o gramatica independenta de context
  - Iesire: functie C ce contine un parser pentru gramatica specificata
- Happy
  - Intrare: fisier ce contine descrierea sintaxei
  - Iesire: Modul Haskell ce contine un parser pentru gramatica specificata

Cum se foloseste Happy?

- Se defineste gramatica ce se doreste a fi parsata intrun fisier <nume>.l
- Se lanseaza happy <nume>.l
- Se obtine un modul in fisierul <nume>.hs care se poate folosi ca parte a unui program Haskell impreuna cu un analizor lexical

Exemplul 1

- Parser pentru expresii(obisnuite) ce contin variabile, operatori +, -, *, / si expresii de forma **`let var = exp in exp`**
- Fisierul .l incepe totdeauna cu liniile

```
{
module Main where
import Char
}
```

- Codul scris intre {} este inclus nealterat in programul generat

- Urmeaza declaratiile(directive):

```
%name calc
%tokentype { Token }
%error { parseError }
```

  - **Numele functiei ce va fi generata**
  - **Tipul tokenurilor ce vor fi acceptate de parser**
    - **Parserul va fi o functie [Token] -> T**

      unde T este determinat de productiile descrise in continuare
  - **Directiva error stabileste numele funtiei ce va raporta erori**

Declararea tokenurilor posibile

```
%token
let     { TokenLet }
in      { TokenIn }
int     { TokenInt $$ }
var     { TokenVar $$ }
'='     { TokenEq }
'+'     { TokenPlus }
'-'     { TokenMinus }
'*'     { TokenTimes }
'/'     { TokenDiv }
'('     { TokenOB }
')'     { TokenCB }
```

- In stanga apare tokenul ca atare iar in dreapta paternul Haskell(constructor de data de tipul Token)
- Simbolul $$ reprezinta valoarea tokenului repectiv
- Urmeaza in fisier o linie %% care delimiteaza declaratiile de regulile gramaticii

Reguli

```
Exp    : let var '=' Exp in Exp   { Let $2 $4 $6 }
       | Exp1                     { Exp1 $1 }

Exp1   : Exp1 '+' Term            { Plus $1 $3 }
       | Exp1 '-' Term            { Minus $1 $3 }
       | Term                     { Term $1 }

Term   : Term '*' Factor          { Times $1 $3 }
       | Term '/' Factor          { Div $1 $3 }
       | Factor                   { Factor $1 }

Factor: int                       { Int $1 }
       | var                      { Var $1 }
       | '(' Exp ')'              { Brack $2 }
```

Forma regulilor

```
n : t_1 ... t_n  { E }
```

unde:

- n este partea stanga(neterminalul) a productiei
- t_1 ... t_n sunt simbolurile din dreapta
- E este expresia ce reprezinta valoarea lui n in functie de $1..&n – valorile token-ului t_i .. t_n
- Parserul reduce sirul de intrare pana ramane simbolul de start cu valoarea sa – semantica acestuia

Sectiunea de cod {...}

- Definitia functiei parseError
- Tipul de data ce reprezinta expresia parsata (E din regulile anterioare)
- Structura de data pentru token-uri, Token
- Un analizor lexical
- Functia main ce primeste o anume intrare, o parseaza si afiseaza rezultatul
- Pentru exemplul dat:

ParseError si Exp

```
{
parseError :: [Token] -> a
parseError _ = error "Parse error"

data Exp
   = Let String Exp Exp
   | Exp1 Exp1
   deriving Show
data Exp1
   = Plus Exp1 Term
   | Minus Exp1 Term
   | Term Term
   deriving Show
```

ParseError si Exp

```
data Term
   = Times Term Factor
   | Div Term Factor
   | Factor Factor
   deriving Show
data Factor
   = Int Int
| Var String
   | Brack Exp
   deriving Show
```

Token

```
data Token
    = TokenLet
    | TokenIn
    | TokenInt Int
    | TokenVar String
    | TokenEq
    | TokenPlus
    | TokenMinus
    | TokenTimes
    | TokenDiv
    | TokenOB
    | TokenCB
    deriving Show
```

Analizorul lexical

```
lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
    | isSpace c = lexer cs
    | isAlpha c = lexVar (c:cs)
    | isDigit c = lexNum (c:cs)
lexer ('=':cs) = TokenEq : lexer cs
lexer ('+':cs) = TokenPlus : lexer cs
lexer ('-':cs) = TokenMinus : lexer cs
lexer ('*':cs) = TokenTimes : lexer cs
lexer ('/':cs) = TokenDiv : lexer cs
lexer ('(':cs) = TokenOB : lexer cs
lexer (')':cs) = TokenCB : lexer cs
```

Analizorul lexical si main

```
lexNum cs = TokenInt (read num) : lexer rest
      where (num,rest) = span isDigit cs
lexVar cs =
   case span isAlpha cs of
      ("let",rest) -> TokenLet : lexer rest
      ("in",rest) -> TokenIn : lexer rest
      (var,rest) -> TokenVar var : lexer rest


main = getContents >>= print . calc . lexer
}
```

Generarea parserului

- Comanda happy exp1.y
  - Se obtine fisierul exp1.hs (528 linii)

```
module Main where
import Char
-- parser produced by Happy Version 1.17
data HappyAbsSyn t4 t5 t6 t7
   = HappyTerminal Token
…
-- end of Happy Template.
```

- Comanda happy exp1.y –i
  - Se obtine fisierul exp1.hs
  - In plus fisierul exp1.info ce descrie automatul LALR(1) pentru gramatica data

Exemple

```
Main> lexer "let x = 2 in let y = 7 in x * (y - 34/22)"
[TokenLet,TokenVar "x",TokenEq,TokenInt
    2,TokenIn,TokenLet,TokenVar "y",TokenEq,TokenInt
    7,TokenIn,TokenVar "x",TokenTimes,TokenOB,TokenVar
    "y",TokenMinus,TokenInt 34,TokenDiv,TokenInt
    22,TokenCB]

Main> calc(lexer "let x = 2 in 33 + x/22")
Let "x" (Exp1 (Term (Factor (Int 2)))) (Exp1 (Plus (Term
    (Factor (Int 33))) (Div (Factor (Var "x")) (Int
    22))))

Main> calc(lexer "letx = 2 in 33 + x/22")

Program error: Parse error
```

Exemplul 2 - varianta

- In loc de "traducerea " expresiei in tipul de data Exp, putem alege varianta evaluarii expresiei:

```
Term : Term '*' Factor { $1 * $3 }
| Term '/' Factor { $1 / $3 }
| Factor { $1 }
```

- Ce e de facut cu variabilele si forma let?
- Scriem o functie ce determina valoarea variabilei in contextul dat (starea la un moment dat)

```
Exp     : let var '=' Exp in Exp { \p -> $6 (($2,$4 p):p) }
        | Exp1 { $1 }
Exp1    : Exp1 '+' Term { \p -> $1 p + $3 p }
        | Exp1 '-' Term { \p -> $1 p - $3 p }
        | Term { $1 }
Term    : Term '*' Factor { \p -> $1 p * $3 p }
        | Term '/' Factor { \p -> $1 p `div` $3 p }
        | Factor { $1 }
Factor  : int { \p -> $1 }
        | var { \p -> case lookup $1 p of
                                Nothing -> error "no var"
                                Just i -> i }
        | '(' Exp ')' { $2 }

Hugs> :info lookup
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```