Monaadilised parserid

- Parseri ülesandeks on:
  - kontrollida kas sisendstring on süntaktiliselt korrektne;
  - konstrueerida sisendstringile vastav AST.
- Parserite soovitavad omadused:
  - BNF lähedane esitus;
  - parserite konstrueerimine olemasolevatest parseritest;
  - mittedetermineeritud grammatikate kasutamine;
  - kontekstist sõltuvate keelte äratundmine.

Monaadilised parserid

- Parserite tüüp

```
newtype Parser a = P (String -> [(a,String)])

runP :: Parser a -> String -> [(a,String)]
runP (P p) = p
```

- Parserite monaad

```
instance Monad Parser where
   return a  = P $ \cs -> [(a,cs)]
   p >>= f   = P $ \cs -> concat [runP (f a) cs'
                                  | (a,cs') <- runP p cs]
```

Monaadilised parserid

- Primitiivparserid

```
instance MonadPlus Parser where
   mzero     = P $ \cs -> []
   mplus p q = P $ \cs -> runP p cs ++ runP q cs

(<|>)  :: Parser a -> Parser a -> Parser a
p <|> q = p `mplus` q

item   :: Parser Char
item    = P $ \cs -> [(head cs, tail cs) | not (null cs)]
```

- Elementaarparserid

```
sat    :: (Char -> Bool) -> Parser Char
sat p   = do {c <- item; if p c then return c else mzero}

char            :: Char -> Parser Char
char c           = sat (c ==)
```

Monaadilised parserid

- Näide:

```
data Tree = Nil | Bin Tree Tree
parens :: Parser Tree
parens =  do char '('
             t1 <- parens
             char ')'
             t2 <- parens
             return (Bin t1 t2)
          <|> return Nil
```

Monaadilised parserid

- Iteratsioon

```
many   :: Parser a -> Parser [a]
many p  = many1 p <|> return []

many1  :: Parser a -> Parser [a]
many1 p = do {a <- p; as <- many p; return (a:as)}
```

Lihtsaid parsereid

- Võtmesõnad

```
string       :: String -> Parser String
string ""     = return ""
string (c:cs) = do {char c; string cs; return (c:cs)}
```

- Identifikaatorid

```
identifier :: Parser String
identifier  = do {c <- lower; cs <- many alphanum; return (c:cs)}
```

- Naturaalarvud

```
natural :: Parser Int
natural  = do ds <- many1 digit
              return (foldl1 (\a b -> 10*a + b) ds)
digit   :: Parser Int
digit    = do c <- sat isDigit
              return (ord c - ord '0')
```

Lihtsaid parsereid

- Täisarvud

```
integer :: Parser Int
integer  = do {char '-'; n <- natural; return (-n)}
         <|> natural
```

- Reaalarvud

```
floating :: Parser Float
floating = do i <- integer
              f <- do {char '.'; fraction} <|> return 0
              return (fromIntegral i + f)

fraction :: Parser Float
fraction = do {ds <- many1 digit; return (foldr op 0 ds)}
   where d `op` x = (x + fromIntegral d)/10
```

Lihtsaid parsereid

- Tühisümbolid

```
space  :: Parser String
space   = many (sat isSpace)

token  :: Parser a -> Parser a
token p = do {a <- p; space; return a}

keyc c  = token (char c)
keyw cs = token (string cs)
ident   = token identifier
nat     = token natural
int     = token integer
float   = token floating
```

Parserite transformaatorid

- Sulud

```
pack :: Parser a -> Parser b -> Parser c -> Parser b
pack s1 p s2 = do {s1; x <- p; s2; return x}
```

- Näide:

```
paren p = pack (keyc '(') p (keyc ')')
brack p = pack (keyc '[') p (keyc ']')
block p = pack (keyw "begin") p (keyw "end")
```

Parserite transformaatorid

- Eraldajaga jadad

```
sepby          :: Parser a -> Parser b -> Parser [a]
p `sepby` sep  = (p `sepby1` sep) <|> return []

sepby1         :: Parser a -> Parser b -> Parser [a]
p `sepby1` sep = do {a <- p; as <- many (sep >> p); return (a:as)}
```

- Näide:

```
commaList p = sepby p (keyw ",")
semicList p = sepby p (keyw ";")
```

Aritmeetilised avaldised

- Grammatika

$$
\begin{array}{rcl}
\text{expr} & = & \text{int} \\
 & | & \text{expr} + \text{expr} \\
 & | & \text{expr} - \text{expr} \\
 & | & \text{expr} * \text{expr} \\
 & | & \text{expr} \ / \ \text{expr} \\
 & | & \text{expr} \ \hat{} \ \text{expr} \\
 & | & (\text{expr})
\end{array}
$$

- Abstraktne süntaksipuu

```
data Expr = Con Int
          | Expr :+: Expr
          | Expr :-: Expr
          | Expr :*: Expr
          | Expr :/: Expr
          | Expr :^: Expr
```

Aritmeetilised avaldised

- Parser ver. 0

```
expr0 =    do {e0 <- expr0; keyc '+'; e1 <- expr0; return(e0 :+: e1)}
       <|> do {e0 <- expr0; keyc '-'; e1 <- expr0; return(e0 :-: e1)}
       <|> do {e0 <- expr0; keyc '*'; e1 <- expr0; return(e0 :*: e1)}
       <|> do {e0 <- expr0; keyc '/'; e1 <- expr0; return(e0 :/: e1)}
       <|> do {e0 <- expr0; keyc '^'; e1 <- expr0; return(e0 :^: e1)}
       <|> do {i <- int; return (Con i)}
       <|> paren expr0
```

- Ei tööta kuna grammatika on vasakrekursiivne!!

Aritmeetilised avaldised

- Parser ver. 1

```
expr1 = do a  <- atom1
           op <- oper1
           e  <- expr1
           return (a `op` e)
       <|> atom1

oper1 =    (keyc '+' >> return (:+:))
       <|> (keyc '-' >> return (:-:))
       <|> (keyc '*' >> return (:*:))
       <|> (keyc '/' >> return (:/:))
       <|> (keyc '^' >> return (:^:))

atom1 = do {i <- int; return (Con i)}
       <|> paren expr1
```

Parserite transformaatorid

- Eraldajaga jadad

```
chainl    :: Parser a -> Parser (a->a->a) -> Parser a
chainl p s = do x  <- p
                ys <- many (do {op <- s; y <- p; return (op,y)})
                return (foldl (\a (op,y) -> a `op` y) x ys)

chainr    :: Parser a -> Parser (a->a->a) -> Parser a
chainr p s = do ys <- many (do {y <- p; op <- s; return (y,op)})
                x  <- p
                return (foldr (\(y,op) b -> y `op` b) x ys)
```

Aritmeetilised avaldised

- Parser ver. 2

```
expr2 :: Parser Expr
expr2 = chainl term2 (    (keyc '+' >> return (:+:))
                      <|> (keyc '-' >> return (:-:)))

term2 :: Parser Expr
term2 = chainl fact2 (    (keyc '*' >> return (:*:))
                      <|> (keyc '/' >> return (:/:)))

fact2 :: Parser Expr
fact2  = chainr atom2 (keyc '^' >> return (:^:))

atom2 :: Parser Expr
atom2 = do {i <- int; return (Con i)}
       <|> paren expr2
```

Aritmeetilised avaldised

- Aritmeetiliste avaldiste väärtustaja

```
expr3 :: Parser Int
expr3 = chainl term3 (    (keyc '+' >> return (+))
                      <|> (keyc '-' >> return (-)))

term3 :: Parser Int
term3 = chainl fact3 (    (keyc '*' >> return (*))
                      <|> (keyc '/' >> return div))

fact3 :: Parser Int
fact3  = chainr atom3 (keyc '^' >> return (^))

atom3 :: Parser Int
atom3 = int <|> paren expr3
```

Monaadilised parserid

- Efektiivsust parandavaid kombinaatoreid

```
first  :: Parser a -> Parser a
first p = P $ \cs -> case runP p cs of
                       []     -> []
                       (x:xs) -> [x]


(<|>)  :: Parser a -> Parser a -> Parser a
p <|> q = first (p `mplus` q)
```

- Kogu sisendi parsimine

```
parse :: Parser a -> String -> a
parse p cs = case runP (first (space >> p)) cs of
                 [(x,"")] -> x
                 _        -> error "Parse error"
```