Süntaksanalüüs

- 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.

Süntaksanalüüs

- Parserite tüüp

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

- Triviaalsed parserid

```
failp :: Parser a
failp = const []

succp :: a -> Parser a
succp x = \cs -> [(x,cs)]

epsilon :: Parser ()
epsilon = succp ()
```

Süntaksanalüüs

- Elementaarparserid

```
satp :: (Char -> Bool) -> Parser Char
satp p []     =  []
satp p (c:cs) =  [(c,cs) | p c]

symp :: Char -> Parser Char
symp c = satp (==c)

tokp :: String -> Parser String
tokp s = \cs -> [(s, drop n cs) | s == take n cs]
       where n = length s
```

Parserite komponeerimine

- Järjestikkompositsioon

```
infixr 6 <*>
(<*>)  :: Parser a -> Parser b -> Parser (a,b)
(p1 <*> p2) cs = [((v1,v2),cs2) | (v1,cs1) <- p1 cs,
                                  (v2,cs2) <- p2 cs1]
```

- Paralleelne kompositsioon

```
infixr 4 <|>
(<|>)  :: Parser a -> Parser a -> Parser a
(p1 <|> p2) cs = p1 cs ++ p2 cs
```

- Näide:

```
ac_bc :: Parser (Char,Char)
ac_bc = (symp 'a' <|> symp 'b') <*> symp 'c'
```

Parserite transformaatorid

- Väärtustega manipuleerimine

```
infixr 5 <@
(<@)   :: Parser a -> (a -> b) -> Parser b
(p <@ f) cs =  [(f v, cs') | (v,cs') <- p cs]
```

- Näide:

```
data Tree = Nil | Bin Tree Tree
parens :: Parser Tree
parens = (    symp '('
          <*> parens
          <*> symp ')'
          <*> parens
         )             <@ (\ (_,(x,(_,y))) -> Bin x y)
         <|> epsilon   <@ const Nil
```

Parserite komponeerimine

- Järjestikkompositsiooni lühendid

```
infixr 6 <*
(<*)    :: Parser a -> Parser b -> Parser a
p <* q   = p <*> q  <@  fst

infixr 6 *>
(*>)    :: Parser a -> Parser b -> Parser b
p *> q   = p <*> q  <@  snd

infixr 6 <:*>
(<:*>)  :: Parser a -> Parser [a] -> Parser [a]
p <:*> q = p <*> q  <@  uncurry (:)
```

Parserite transformaatorid

- Iteratsioon

```
many   :: Parser a -> Parser [a]
many p  =     p <:*> many p
          <|> succp []

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

- Üldistatud järjestikkompositsioon

```
seqp :: [Parser a] -> Parser [a]
seqp = foldr (<:*>) (succp [])
```

- Üldistatud paralleelkompositsioon

```
altp :: [Parser a] -> Parser a
altp = foldr (<|>) failp
```

Lihtsaid parsereid

- Võtmesõnad

```
tokp :: String -> Parser String
tokp = seqp . map symp
```

- Identifikaatorid

```
ident :: Parser String
ident = satp isAlpha <:*> many (satp isAlphanum)
```

- Naturaalarvud

```
digit :: Parser Int
digit = satp isDigit <@ \x -> ord x - ord '0'

natural :: Parser Int
natural = many1 digit
          <@ foldl1 (\a b -> 10*a + b)
```

Lihtsaid parsereid

- Täisarvud

```
sign :: Parser (Int -> Int)
sign = symp '-' <@ const negate  <|>  succp id

integer :: Parser Int
integer = sign <*> natural <@ uncurry ($)
```

- Reaalarvud

```
fract :: Parser Float
fract = many1 digit <@ foldr op 0
      where d `op` x = (x + fromInt d)/10

float :: Parser Float
float = (integer <@ fromInt)
        <*> (symp '.' *> fract  <|>  succp 0)
        <@ uncurry (+)
```

Lihtsaid parsereid

- Tühisümbolid

```
sp :: Parser a -> Parser a
sp = (many (satp isSpace) *>)

spsym ::  Char -> Parser Char
spsym = sp . symp

sptok ::  String -> Parser String
sptok = sp . tokp

spident :: Parser String
spident = sp ident

spint :: Parser Int
spint = sp integer
```

Parserite transformaatorid

- Optsioon

```
optp :: Parser a -> Parser [a]
optp p =     p <@ (:[])
         <|> succp []
```

- Sulud

```
pack :: Parser a -> Parser b -> Parser c -> Parser b
pack s1 p s2 = s1 *> p <* s2
```

- Näide:

```
paren p = pack (spsym '(') p (spsym ')')
brack p = pack (spsym '[') p (spsym ']')
block p = pack (sptok "begin") p (sptok "end")
```

Parserite transformaatorid

- Eraldajaga jadad

```
listp :: Parser a -> Parser b -> Parser [a]
listp p s =     p <:*> many (s *> p)
            <|> succeed []
```

- Näide:

```
commaList p = listp p (symp ',')
semicList p = listp p (symp ';')
listExpr    = brack . commaList
pasBlock    = block . semicList
```

Aritmeetilised avaldised

- Grammatika

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

- Abstraktne süntaksipuu

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

Aritmeetilised avaldised

- Parser ver. 1

```
expr =     atom <*> oper <*> expr
             <@ \ (a,(op,e)) -> op a e
       <|> atom

oper =     spsym '+' <@ const (:+:)
       <|> spsym '-' <@ const (:-:)
       <|> spsym '*' <@ const (:*:)
       <|> spsym '/' <@ const (:/:)

atom =     spint <@ Con
       <|> paren expr
```

Parserite transformaatorid

- Eraldajaga jadad

```
chainl :: Parser a -> Parser (a->a->a) -> Parser a
chainl p s = p <*> many (s <*> p)
               <@ uncurry (foldl (flip ap2))
           where ap2 (op,y) = (`op` y)

chainr :: Parser a -> Parser (a->a->a) -> Parser a
chainr p s = many (p <*> s) <*> p
               <@ uncurry (flip (foldr ap1))
           where ap1 (x,op) = (x `op`)
```

Aritmeetilised avaldised

- Parser ver. 2

```
expr :: Parser Expr
expr = chainl term (    spsym '+' <@ const (:+:)
                    <|> spsym '-' <@ const (:-:))

term :: Parser Expr
term = chainl fact (    spsym '*' <@ const (:*:)
                    <|> spsym '/' <@ const (:/:))

fact :: Parser Expr
fact = spint <@ Con  <|>  paren expr
```

Aritmeetilised avaldised

- Aritmeetiliste avaldiste väärtustaja

```
expr :: Parser Int
expr = chainl term (    spsym '+' <@ const (+)
                    <|> spsym '-' <@ const (-))

term :: Parser Int
term = chainl fact (    spsym '*' <@ const (*)
                    <|> spsym '/' <@ const div)

fact :: Parser Int
fact = spint <@ id  <|>  paren expr
```

Parserite efektiivsus

- Efektiivsust parandavaid kombinaatoreid

```
just :: Parser a -> Parser a
just p = filter (null . snd) . p

first :: Parser a -> Parser a
first p cs = [head r | not (null r)]
           where r = p cs

greedy     = first . many
greedy1    = first . many1
compulsion = first . option

sp = (greedy (satp isSpace) *>)
```

Monaadilised parserid

- Parserite monaad

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

instance Functor Parser where
    fmap f (P p) = P (\inp -> [(f v, out) | (v,out) <- p inp])

instance Monad Parser where
    return v     = P (\inp -> [(v,inp)])
    (P p) >>= f  = P (\inp -> concat [f' out | (v,out) <- p inp,
                                               let P f' = f v])

instance MonadPlus Parser where
    mzero  = P (\inp -> [])
    (P p) `mplus` (P q) = P (\inp -> (p inp ++ q inp))
```

Monaadilised parserid

- Teised primitiivsed parserid

```
item :: Parser Char
item  = P (\inp -> case inp of
                     []     -> []
                     (x:xs) -> [(x,xs)])
force         :: Parser a -> Parser a
force (P p)   = P (\inp -> let x = p inp in
                           (fst (head x), snd (head x)) : tail x)
first         :: Parser a -> Parser a
first (P p)   = P (\inp -> case p inp of
                             []     -> []
                             (x:xs) -> [x])
papply             :: Parser a -> String -> [(a,String)]
papply (P p) inp = p inp
```

Monaadilised parserid

- Tuletud kombinaatorid

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

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

many      :: Parser a -> Parser [a]
many p     = force (many1 p +++ return [])

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

Monaadilised parserid

- Tuletud kombinaatorid

```
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 {x <- p; xs <- many (do {sep; p});
                     return (x:xs)}

ops     :: [(Parser a, b)] -> Parser b
ops xs  = foldr1 (+++) [do {p; return op} | (p,op) <- xs]

bracket :: Parser a -> Parser b -> Parser c -> Parser b
bracket open p close = do {open; x <- p; close; return x}
```

Monaadilised parserid

- Tuletud kombinaatorid

```
chainl  :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainl p op v  = (p `chainl1` op) +++ return v

chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
p `chainl1` op = do {x <- p; rest x}
  where rest x = do {f <- op; y <- p; rest (f x y)}
                 +++ return x
chainr  :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainr p op v  = (p `chainr1` op) +++ return v

chainr1 :: Parser a -> Parser (a -> a -> a) -> Parser a
p `chainr1` op = do {x <- p; rest x}
  where rest x = do {f <- op; y <- p `chainr1` op; return (f x y)}
                 +++ return x
```