Cursul 11 – Parser functional

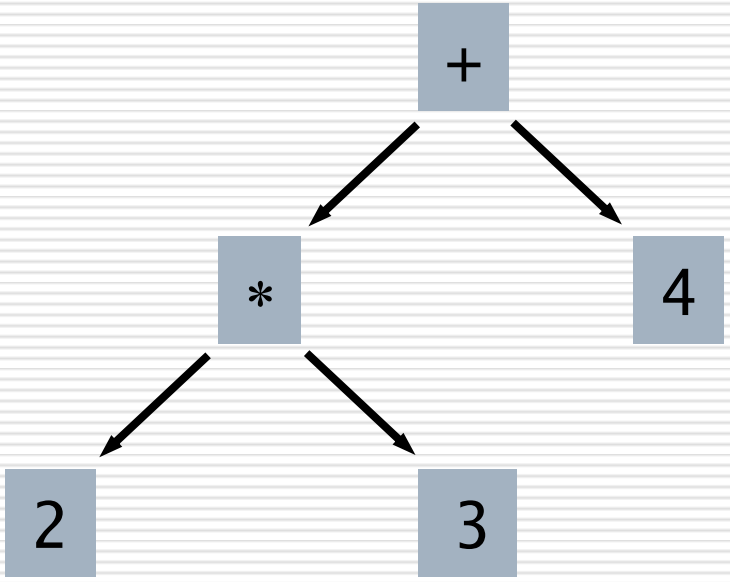
- Parserul ca functie
- Parsere de baza: return, esec, item
- Combinarea paserelor
  - Secventa
  - Alegere
- Primitive derivate
- Parserul ca monada
- Studiu de caz: expresii aritmetice

Ce este un parser?

- Un parser este un program ce analizeaza un text si determina structura sa sintactica relativ cu o descriere sintactica (gramatica independenta de context)

Ce este un parser?

2*3+4



- Parserul poate fi privit ca o functie

  `type Parser = String → Tree`

- Mai general parserul poate produce un arbore fara sa consume intregul sir de intrare si atunci:

  `type Parser = String → (Tree, String)`

- Inca mai general, daca presupunem ca parserul returneaza o lista de rezultate(lista [] va insemna esec, lista singleton va insemna succes):

  `type Parser = String → [(Tree, String)]`

Parserul ca tip parametrizat

- Este util sa parametrizam tipul pentru a putea scrie parser cat mai general: un parser de tip a este o functie care are la intrare un string si produce o lista de rezultate, fiecare fiind o pereche ce cuprinde un rezultat de tip a si un string:

```
type Parser a = String → [(a, String)]
```

```
type Parser a = String → [(a, String)]
```

A parser for things

Is a function from strings

To list of pairs

Of things and strings

return, esec si item

- Parserul `return v` se termina cu succes si produce v

```
return :: a -> Parser a
return v = \ i -> [(v, i)]
```

(sau `return v i = [(v, i)]`)

- Parserul *esec*:

```
esec :: Parser a
esec i = []
```

- Parserul item consuma un caracter din sirul de intrare:

```
item :: Parser a
item = \ i -> case i of
                  [] -> []
                  (x:xs) -> [(x, xs)]
```

Aplicatia parse

```
parse :: Parser a -> String -> [(a, String)]
parse p input = p input

*Main> parse (preturn 1) "hallo“
[(1,"hallo")]

*Main> parse esec "hallo"
[]

*Main> parse item "hallo"
[('h',"allo")]
```

Alegere p +++ q

- Se aplica p; daca acesta esueaza se aplica q:

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q = \inp -> case parse  p inp of
            [] -> parse q inp
            [(v, out)] -> [(v, out)]
```

```
*Main> parse (preturn 'd' +++ item ) "abc"
[('d',"abc")]
*Main> parse (esec +++ item ) "abc"
[('a',"bc")]
```

Primitive de parsare derivate

- Parserul **`sat p`** este un parser pentru un caracter ce satisface predicatul p:

```
sat :: (Char -> Bool) -> Parser Char
sat = do  x <- item
          if p x then return x else esec
```

Din `sat` construim:

```
digit :: Parser Char
digit = sat isDigit

lower :: Parser Char
lower = sat isLower

upper :: Parser Char
upper = sat isUpper
```

```
letter :: Parser Char
letter = sat isAlpha

alphanum :: Parser Char
alphanum= sat isAlphaNum

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

Parserul string

```
string :: String -> Parser String
string [] = return []
string (x:xs) do char x
                 string xs
                 return (x:xs)
```

Repetitia

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

many1 :: Parser a -> Parser [a]
many1 p = do      v <- p
                  vs <- many p
                  return (v:vs)
```

Identificatori, numere, spatii

```
ident :: Parser String
ident = do x <- lower
           xs <- many alphanum
           return (x:xs)

nat :: Parser Int
nat = do xs <- many1 digit
         return(read xs)


space :: Parser ()
space = do xs <- many(sat isSpace)
           return ()
```

Ignorarea spatiilor

```
token :: Parser a -> Parser a
token p = do space
             v <- p
             space
             return v
identifier:: Parser String
identifier = token ident

natural :: Parser Int
natural = token nat

symbol :: String -> Parser String
symbol xs = token(string xs)
```

Un parser pentru liste de numere

```
p :: Parser [Int]
p = do  symbol "["
        n <- natural
        ns <- many(do symbol ","
                      natural)
        symbol "]"
        return (n:ns)
```

Parserul ca monada

- Unele din definitiile de mai sus nu sunt corecte:
  - Notatia do este "sintactic sugar" pentru operatorul monadic >>= (ce trebue definit)
- Trebue adaugat:

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

instance Monad Parser where
    return v  =  P (\inp -> [(v,inp)])
    p >>= f   =  P (\inp -> case parse p inp of
                              [] -> []
                              [(v,out)] -> parse (f v) out)
```

Parserul ca monada

- (http://thor.info.uaic.ro/~grigoras/pf/exemple/curs11/Parsing.lhs)

```
*Parsing> parse (string "aa") "aaan"
[("aa","an")]

*Parsing> parse identifier "  alpha  beta   gama  "
[("alpha","beta   gama  ")]

*Parsing> parse identifier "  12alpha"
[]

*Parsing> parse (symbol ":=" ) "   :=  gama"
[(":=","gama")]
```

Un parser pentru expresii aritmetice

- Daca definim expresiile aritmetice prin gramatica:

```
expr    ::= term + expr | term
term    ::= factor * term | factor
factor  ::=  (expr) | nat
nat     ::= 0|1|2|...
```

atunci aceasta gramatica se traduce intr-un paser prin rescrierea regulilor folosind primitivele de parsare

http://thor.info.uaic.ro/~grigoras/pf/exemple/curs11/parser Exp.lhs

Un parser pentru expresii aritmetice

```
*Main> eval "  25+99*(8 + 2)"
1015

*Main> eval "25+99*(8o + 2)"
*** Exception: unused input *(8o + 2)

*Main> eval " a 25+99*(8 + 2)"
*** Exception: invalid input
```