

# Curs 4

# Analiză sintactică

# Tipul unui analizor sintactic

## Prima încercare

```
type Parser a = String -> a
```

# Tipul unui analizor sintactic

## Prima încercare

**type** Parser a = **String** -> a

- Dar cel puțin pentru rezultate parțiale, va mai rămâne ceva de analizat

# Tipul unui analizor sintactic

## Prima încercare

```
type Parser a = String -> a
```

- Dar cel puțin pentru rezultate parțiale, va mai rămâne ceva de analizat

## A doua încercare

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

# Tipul unui analizor sintactic

## Prima încercare

**type** Parser a = **String** -> a

- Dar cel puțin pentru rezultate parțiale, va mai rămâne ceva de analizat

## A doua încercare

**type** Parser a = **String** -> (a, **String**)

- Dar dacă gramatica e ambiguă?
- Dar dacă intrarea nu corespunde nici unui element din a?

# Tipul unui analizor sintactic

## Dr. Seuss on Parser Monads:



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

A Parser for Things  
is a function from Strings  
to Lists of Pairs  
of Things and Strings!

Art: Seuss; Type: Waser; Rhyme: Ruahr

# Tipul Parser

## Tipul Parser

```
newtype Parser a =  
  Parser { apply :: String -> [(a, String)] }
```

```
-- Folosirea unui parser
```

```
-- apply :: Parser a -> String -> [(a, String)]
```

```
-- apply (Parser f) s = f s
```

```
-- Daca exista parsare, da prima varianta
```

```
parse :: Parser a -> String -> a
```

```
parse m s = head [ x | (x,t) <- apply m s, t == "" ]
```



# Parsare pentru caractere

-- *Recunoasterea unui caracter*

anychar :: Parser **Char**

anychar = Parser f

**where**

f [] = []

f (c:s) = [(c,s)]

## Parsare pentru caractere

-- *Recunoasterea unui caracter*

```
anychar :: Parser Char
```

```
anychar = Parser f
```

```
  where
```

```
    f []      = []
```

```
    f (c:s) = [(c,s)]
```

```
*Main> parse anychar "a"
```

```
'a'
```

```
*Main> parse anychar "ab"
```

```
*** Exception: Prelude.head: empty list
```

```
*Main> apply anychar "abc"
```

```
[( 'a' , "bc" )]
```

# Parsare pentru caractere

```
-- Recunoasterea unui caracter cu o proprietate  
satisfy :: (Char -> Bool) -> Parser Char  
satisfy p = Parser f  
  where  
    f [] = []  
    f (c:s) | p c = [(c, s)]  
             | otherwise = []
```

## Parsare pentru caractere

-- *Recunoasterea unui caracter cu o proprietate*

```
satisfy :: (Char -> Bool) -> Parser Char
```

```
satisfy p = Parser f
```

```
  where
```

```
    f [] = []
```

```
    f (c:s) | p c = [(c, s)]
```

```
              | otherwise = []
```

```
*Main> parse (satisfy isUpper) "A"
```

```
'A'
```

```
(0.01 secs, 52,760 bytes)
```

```
*Main> parse (satisfy isUpper) "a"
```

```
*** Exception: Prelude.head: empty list
```

```
*Main> apply (satisfy isUpper) "Ab"
```

```
[( 'A', "b")]
```

# Parsare pentru caractere

```
-- Recunoasterea unui anumit caracter  
char :: Char -> Parser Char  
char c = satisfy (== c)
```

# Parsare pentru caractere

-- *Recunoasterea unui anumit caracter*

```
char :: Char -> Parser Char
```

```
char c = satisfy (== c)
```

```
*Main> parse (char 'a') "a"  
'a'
```

```
(0.00 secs, 52,824 bytes)
```

```
*Main> parse (char 'a') "ab"
```

```
*** Exception: Prelude.head: empty list
```

```
*Main> apply (char 'a') "ab"  
[( 'a' , "b" )]
```

## Parsarea unui cuvânt cheie

```
-- Recunoasterea unui cuvânt cheie
string :: String -> Parser String
string [] = Parser (\s -> [([],s)])
string (x:xs) = Parser f
  where
    f s = [(y:z,zs) | (y,ys) <- apply (char x) s,
                      (z,zs) <- apply (string xs) ys]
```

## Parsarea unui cuvânt cheie

-- *Recunoasterea unui cuvânt cheie*

```
string :: String -> Parser String
```

```
string [] = Parser (\s -> [([],s)])
```

```
string (x:xs) = Parser f
```

```
  where
```

```
    f s = [(y:z,zs) | (y,ys) <- apply (char x) s,  
                      (z,zs) <- apply (string xs) ys]
```

```
*Main> parse (string "abc") "abc"
```

```
"abc"
```

```
*Main> parse (string "abc") "abcd"
```

```
*** Exception: Prelude.head: empty list
```

```
"*Main> apply (string "abc") "abcd"
```

```
[( "abc", "d")]
```



# Monada Parser

```
-- class Monad m where
--   return :: a -> m a
--   (>=) :: m a -> (a -> m b) -> m b
```

```
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
                              ])
```

# Monada Parser

```
-- Recunoasterea unui cuvant cheie
string :: String -> Parser String
string [] = Parser (\s -> [([],s)])
string (x:xs) = Parser f
  where
    f s = [(y:z,zs) | (y,ys) <- apply (char x) s,
                      (z,zs) <- apply (string xs) ys]
```

e echivalent cu

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

## Combinarea variantelor

```
digit = satisfy isDigit
abcP = satisfy ('elem' ['A','B','C'])

alt :: Parser a -> Parser a -> Parser a
alt p1 p2 = Parser f
    where f s = apply p1 s ++ apply p2 s
```

# Combinarea variantelor

```
digit = satisfy isDigit
abcP = satisfy ('elem' ['A','B','C'])

alt :: Parser a -> Parser a -> Parser a
alt p1 p2 = Parser f
    where f s = apply p1 s ++ apply p2 s
```

```
*Main> apply (alt digit abcP) "1sd"
[( '1' ,"sd")]
*Main> apply (alt digit abcP) "Asd"
[( 'A' ,"sd")]
*Main> apply (alt digit abcP) "dsd"
[]
*Main> parse (alt digit abcP) "A"
'A'
*Main> parse (alt digit abcP) "1"
'1'
```

# Parser e monadă cu plus

```
--      class MonadPlus m where
--          mzero  :: m a
--          mplus   :: m a -> m a -> m a
```

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

- mzero reprezintă analizorul sintactic care eşuează tot timpul
- mplus reprezintă combinarea alternativelor

```
instance Alternative Parser where
    empty  = mzero
    (<|>) = mplus
```

# Parser e monadă cu plus

```
instance MonadPlus Parser where
```

```
    mzero      = Parser (\s -> [])
```

```
    mplus m n   = Parser (\s -> apply m s ++ apply n s)
```

```
instance Alternative Parser where
```

```
    empty = mzero
```

```
    (<|>) = mplus
```

```
*Main> apply (digit <|> abcP) "1www"
```

```
[( '1' , "www" )]
```

```
*Main> apply (digit <|> abcP) "Awww"
```

```
[( 'A' , "www" )]
```

```
*Main> parse (digit <|> abcP) "B"
```

```
'B'
```

```
*Main> parse (digit <|> abcP) "2"
```

```
'2'
```

# Recunoașterea unui caracter cu o proprietate

## Alternative și Gărzi

```
guard :: MonadPlus f => Bool -> f ()  
guard True = return ()  
guard False = mzero
```

```
satisfy :: (Char -> Bool) -> Parser Char  
satisfy p = Parser f  
  where  
    f [] = []  
    f (c:s) | p c = [(c, s)]  
             | otherwise = []
```

e echivalentă cu

```
satisfy :: (Char -> Bool) -> Parser Char  
satisfy p = do   c <- anychar  
                 guard (p c)  
                 return c
```

## Recunoașterea unei secvențe repetitive

```
-- Steluta Kleene (zero, una sau mai multe repetitii)
many :: Parser a -> Parser [a]
many p = some p   'mplus'   return []

-- cel puțin o repetitie
some :: Parser a -> Parser [a]
some p = do    x <- p
              xs <- many p
              return (x:xs)
```



# Recunoașterea unui numar întreg

*-- Recunoasterea unui numar natural*

decimal :: Parser **Int**

```
decimal = do  s <- some digit  
            return (read s)
```

*-- Recunoasterea unui numar negativ*

negative :: Parser **Int**

```
negative = do  char '-'  
              n <- decimal  
              return (-n)
```

*-- Recunoasterea unui numar intreg*

integer :: Parser **Int**

```
integer = decimal 'minus' parseNeg
```

# Recunoașterea unui identificator

## Cum arată un identificator

Un identificator este definit de doi parametri

- felul primului caracter (e.g., începe cu o literă)
- felul restului caracterelor (e.g., literă sau cifră)

Dat fiind un parser pentru felul primului caracter și un parser pentru felul următoarelor caractere putem parsea un identificator:

```
-- Recunoasterea unui identificator
identifier :: Parser Char -> Parser Char -> Parser String
identifier firstCh nextCh = do  c <- firstCh
                                s <- many nextCh
                                return (c : s)
```

Exemplu

```
myId = identifier (satisfy isAlpha) (satisfy isAlphaNum)
```

# Eliminarea spațiilor

## Ignorarea spațiilor

```
skipSpace :: Parser ()  
skipSpace = do _ <- many (satisfy isSpace)  
             return ()
```

## Ignorarea spațiilor de dinainte și după

```
token :: Parser a -> Parser a  
token p = do skipSpace  
             x <- p  
             skipSpace  
             return x
```

# Modulul Exp

```
module Exp where
```

```
import Monad
```

```
import Parser
```

```
data Exp = Lit Int
```

```
      | Exp :+: Exp
```

```
      | Exp :*: Exp
```

```
      deriving (Eq, Show)
```

```
evalExp    :: Exp -> Int
```

```
evalExp    (Lit n)      = n
```

```
evalExp    (e :+: f) = evalExp e + evalExp f
```

```
evalExp    (e :*: f) = evalExp e * evalExp f
```

# Recunoașterea unei expresii

```
parseExp :: Parser Exp
parseExp = parseLit 'mplus' parseAdd 'mplus' parseMul
  where
    parseLit = do n <- integer
                  return (Lit n)
    parseAdd = do char '('
                  d <- token parseExp
                  char '+'
                  e <- token parseExp
                  char ')'
                  return (d :+: e)
    parseMul = do char '('
                  d <- token parseExp
                  char '*'
                  e <- token parseExp
                  char ')'
                  return (d :*: e)
```

# Recunoașterea și evaluarea unei expresii

```
*Exp> parse parseExp "(1 + (2 * 3))"  
Lit 1 :+: (Lit 2 *: Lit 3)  
*Exp> evalExp (parse parseExp "(1 + (2 * 3))")  
7  
*Exp> parse parseExp "( ( 1 + 2 ) * 3 )"  
(Lit 1 :+: Lit 2) *: Lit 3  
*Exp> evalExp (parse parseExp "( ( 1 + 2 ) * 3 )")  
9
```



Pe săptămâna viitoare!