



Syntaktická analýza v Haskell

úlohou syntaktickej analýzy je zistiť, či vstupný stream lexém patrí do nejakého popísaného jazyka. Side-effect toho je/môže byť vnútorná reprezentácia.

- Jeroen Fokker: Functional Parsers, LNCS 925, 1996,
<http://people.cs.uu.nl/jeroen/article/parsers/>
- Johan Jeuring, Doaitse Swierstra: Grammars and Parsing,
<http://www.cs.uu.nl/docs/vakken/gont/diktaat.pdf>
- Daan Leijen: **Parsec**, a fast combinator parser,
<http://research.microsoft.com/en-us/um/people/daan/download/parsec/parsec-letter.pdf>
- Doaitse Swierstra: Combinator Parsers: From Toys to Tools,
<http://people.cs.uu.nl/doaitse/Papers/2000/HaskellWorkshop.pdf>

Pozn.:

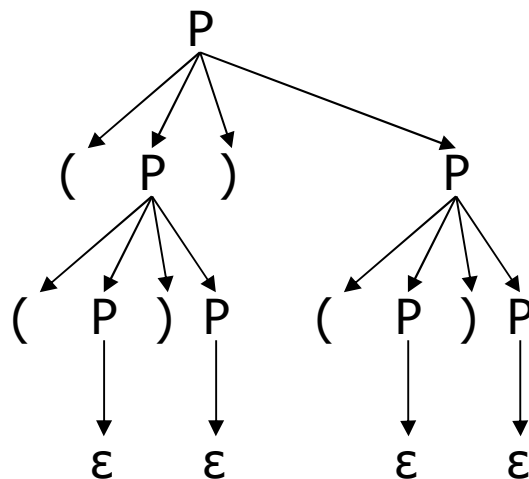
v celom texte sa mimovoľne objavuje rôzne vy(s)kloňované slovo *parser*, čoho slovenským ekvivalentom je syntaktický analyzátor

- gramatika: $P \rightarrow (P)P \mid \varepsilon$

- odvodenie: $P \rightarrow (P) \quad P \rightarrow ((P)P) \quad P \rightarrow ((P)P)(P)P \rightarrow^4 (())()$

- jazyk: $L = \{ w \in \{ (,) \}^* \mid P \rightarrow^* w \}$

- strom odvoduena:


$$R \rightarrow a R a \mid b R b \mid c R c \mid \epsilon$$

S -> S S | a

$$L \rightarrow (LL) \mid \lambda \text{ id.L} \mid \text{id}$$



Syntaktický analyzátor

- program, ktorý o vstupnej vete rozhoduje, či patrí alebo nie do jazyka,
- pre rôzne gramatiky to môže byť rôzne ťažký problém,
- *my nechceme riešiť ťažké problémy...*
- ak gramatika nie je ľavo-rekurzívna, vieme napísať pomerne priamočiaro analyzátor metódou rekurzívneho zostupu,
- ak je ľavo-rekurzívna, tak ju najprv prerobíme, aby sme si uvarili čaj...
- cieľom tejto prednášky je vybudovať:
 - sériu jednoduchých atomických syntaktických analyzátorov,
 - sériu kombinátorov, ktorými z nich skladáme zložitejšie analyzátory,
 - pochopiť filozófiu kombinácie analyzátorov napr. tým, že napíšeme analyzátor pre lambda-výrazy.

Alternatívne prístupy:

- tzv. monadické parsery, špeciálny štýl písania kódu.
- Haskell obsahuje modul Parsec - knižnica pre písanie analyzátorov,



Syntaktická analýza

Najprv si ujasnime typ funkcie, ktorá realizuje syntaktickú analýzu.

- **p**::Parser je funkcia, ktorá zoberie vstup a vráti vnútornú formu, ak patrí do jazyka.

```
type Parser = String -> Tree
```

- šikovnejšie je definovať **p** ako funkciu, ktorá zoberie vstup a vráti nezanalýzovanú časť vstupu (plus vnútornú reprezentáciu):

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

- ak parametrizujeme typ vnútornej reprezentácie ako *result*, dostaneme:

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

To funguje v prípade, ak pre nejaký prefix vstupného streamu existuje odvodenie v jazyku. Co ak neexistuje ? Co ak ich je viac ?



Nedeterministická analýza

- čo ak analyzátor vráti viacero riešení, resp. žiadne (existuje viacero odvodení) ?... Pozbierame ich do zoznamu – tzv. nedeterministický parser

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

- keďže typ `String = [Char]`, po úplnej parametrizácii dostaneme typ `Parser`, ktorý predstavuje nedeterministický analyzátor vstupného streamu objektov typu `symbol` (napr. `Char`), pričom výstupná reprezentácia je typu `result`:

`type Parser symbol result = [symbol] -> [[symbol],result]`

- príklad analyzátoru na type `String = [Char]`, ktorý vráti `Char`, resp. `Int`

`p2: Parser Char Int = [Char] -> [[Char], Int]`

`p2 "123" = [("123",0), ("23",1), ("3",12), ("",123)]` -- viacero výsledkov

`p1: parser Char Char = [Char] -> [[Char], Char]`

`p1 "abc" = []` -- žiaden výsledok

type Parser symbol result = [symbol] -> [[symbol],result]

Elementárne analyzátory

Začneme od veľmi jednoduchých analyzátorov, ktoré budeme neskôr kombinovať do zložitejších.

- analyzátor, ktorý rozpoznáva jedno (konštantné) písmenko 'a':

```
symbola      :: Parser Char Char
symbola []    = []
symbola (x:xs) | x=='a' = [ (xs, 'a') ]
                | otherwise= []
```

-- ak nie je nič na vstupe
-- ak je 'a' na vstupe
-- ak nie je 'a' na vstupe

- parametrizujeme konštantu a zovšeobecníme pre stream symbolov ľub.typu:

```
symbol      :: Eq s => s -> Parser s s
symbol a (x:xs) | a==x = [ (xs, x) ]
                | otherwise= []
```

$P_a \rightarrow a$

Príklad: `symbol a [] = []`
Main> symbol 'a' "abcd"
[("bcd",'a')]
Main> symbol 'a' "bcd"
[]

type Parser symbol result = [symbol] -> [[symbol],result]



Elementárne analyzátory

- parser, ktorý analyzuje jeden symbol:

```
symbol          :: Eq s => s -> Parser s s
symbol a []      = []
symbol a (x:xs)  = [ (xs, a) | a == x ]
```

- parser, ktorý analyzuje postupnosť lexém:

```
token           :: Eq[s] => [s] -> Parser s [s]
token k xs      | k==take n xs  = [ (drop n xs, k)]
                | otherwise     = []
                where n = length k
```

```
Main> token "abc" "abcdef"
[("def","abc")]
Main> token "abc" "abdef"
[]
```

type Parser symbol result = [symbol] -> [[symbol],result]

Elementárne analyzátory

po vzore definície symbol z minulého slajdu, definujme analyzátor, ktorý akceptuje jeden symbol spĺňajúci logickú podmienku:

```
satisfy      :: (s -> Bool) -> Parser s s
satisfy p [] = []
satisfy p (x:xs) = [ (xs, x) | p x ]
```

```
Main> satisfy isDigit "123abc"
[("23abc",'1')]
Main> satisfy isDigit "abc123"
[]
```

Cv1: Keďže satisfy je zovšeobecnením symbol, definujte symbol ako inštanciu satisfy.

Cv2: V Haskellu je funkcia isDigit. Definujte analyzátor digit10::Parser Char Char, ktorý analyzuje jednu desiatkovú cifru. Hint: pomocou satisfy.

Cv3: Definujte analyzátor hexa::Parser Char Char, ktorý analyzuje jednu šesnástkovú cifru, t.j. (0, 1, ..., F).



Triviálne analyzátory

- ϵ -pravidlo

```
epsilon      :: Parser s ()           -- () je ako typ void
epsilon xs   = [ ( xs, () ) ]        -- () hodnota typu (), ako null
```

- ϵ -pravidlo s tým, že vráti vnútornú reprezentáciu

```
succeed      :: r -> Parser s r
succeed v xs  = [ (xs, v) ]          -- v je výst.hodnota typu r
```

```
epsilon      :: Parser s ()           -- epsilon ako inštancia succeed
epsilon      = succeed ()
```

- parser, ktorý nič neakceptuje

```
fail         :: Parser s r
fail xs      = []
```

type Parser symbol result = [symbol] -> [[symbol],result]

Kombinátory analyzátorov

- definujeme dva symboly v infixovej notácii

```
infixr 6 <*>      -- sekvenčné zret'azenie analyzátorov
infixr 4 <|>      -- zjednotenie analyzátorov
```

- sekvenčné zret'azenie analyzátorov:

```
(<*>)                :: Parser s a -> Parser s b -> Parser s (a,b)
(p1 <*> p2) xs      = [ (xs2, (v1,v2))
                        | (xs1, v1) <- p1 xs
                        , (xs2, v2) <- p2 xs1
                        ]
```

- dijsunkcia (zjednotenie) analyzátorov:

```
(<|>)                :: Parser s a -> Parser s a -> Parser s a
(p1 <|> p2) xs      = p1 xs ++ p2 xs
```

```
Main> ((symbol 'a' <|> symbol 'b') <*> (symbol '1' <|> symbol '2')) "a2cd"
[("cd",('a','2'))]
```

Cv4. Prečo je lepšie definovať operátor <|> ako infixr miesto infixl.

Cv5. Definujte analyzátor, ktorý akceptuje reťazce začínajúce YES alebo NO.



Transformátory analyzátorov

Transformátor analyzátoru je funkcia (funkcionál), ktorá zoberie analyzátor a vráti nový analyzátor s odvodenými/zmenenými vlastnosťami.

- transformátor **sp** analyzátoru *p*, ktorý funguje ako *p* ale ignoruje úvodné medzery:

```
sp                :: Parser Char a -> Parser Char a
sp p              = p . dropWhile (==' ')
```

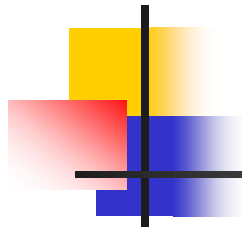
- transformátor **just** analyzátoru *p* akceptuje vstup podľa *p* len, ak bol dočítaný celý vstupný stream:

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

```
Main> just (symbol 'a' <*> (symbol 'b' <|> symbol 'c' <|> succeed ' ')) "ac"
[("","('a','c'))]
```

Cv5'. Definujte analyzátor, ktorý akceptuje reťazce YES alebo NO.

Cv6. Definujte just pomocou list-comprehension, bez použitia filter.



Aplikátor analyzátorov

Aplikátor analyzátoru je kombinátor, ktorý modifikuje typ reprezentácie tak, že aplikuje nejakú funkciu na výslednú internú reprezentáciu.

`infix 5 <@` `-- infixová notácia pre aplikátor`

```
(<@)      :: Parser s a -> (a->b) -> Parser s b
(p <@ f) xs = [ (ys, f v)
                | (ys, v) <- p xs
                ]
```

- analyzátor `digit` akceptuje cifru a vráti jej desiatkovú hodnotu:

```
digit      :: Parser Char Int
digit      = satisfy isDigit <@ f
            where f c = ord c - ord '0'
```

```
Main> digit "12abc"
[("2abc",1)]
```



Skratky – rôzne klony <*>

- kombinátor analyzátorov p a q, ktorý ignoruje výsledok q:

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

```
(p1 <* p2) xs = [ (xs2, v1)  
                  | (xs1, v1) <- p1 xs  
                  , (xs2, _) <- p2 xs1  
                  ]
```

- kombinátor analyzátorov p a q, ktorý ignoruje výsledok p:

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

- kombinátor analyzátorov p:a a q:[a], ktorý vytvára zoznam výsledkov typu [a]:

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

Cv7. Definujte <:*> bez list (x,xs) = (x:xs).

- ```
Main> (option (symbol 'a')) "abc"
[("bc", "a"), ("abc", "")]
```

```
Main> (many(symbol 'a') <*> symbol 'b') "aaabb"
[("b",("aaa",'b'))]
Main> (many(symbol 'a') <*> many1(symbol 'b')) "aaabb"
[("",("aaa","bb")),("b",("aaa","b"))]
```



# Transformátor sequence

- sequence zoberie sekvenciu (zoznam) analyzátorov rovnakého typu a vráti analyzátor, ktorý vracia zoznam výsledkov

sequence :: [Parser s a] -> Parser s [a]

sequence = foldr (<:\*>) (succeed []) {a}+{b}\*{c}

```
Main> sequence [many1 (symbol 'a'), many (symbol 'b'), option (symbol 'c')] "abbb"
[("",["a","bbb",""]),("b",["a","bb",""]),("bb",["a","b",""]),("bbb",["a","",""])]
```

Cv8. Definujte sequence bez foldr.

Cv8'. Definujte token pomocou sequence.

Cv9. Definujte analyzátor mobilného čísla (pre niektorého operátora), resp. PSČ

Cv10. Definujte analyzátor korektného dátumu gregoriánskeho kalendára

- choice zoberie sekvenciu (zoznam) analyzátorov rovnakého typu a vráti analyzátor, ktorý reprezentuje zjednotenie/disjunkciu analyzátorov.

choice :: [Parser s a] -> Parser s a

choice = foldr (<|>) failp



# Deterministický analyzátor

type DetPars symbol result = [symbol] -> result

- transformátor nedeterministického p analyzátoru na deterministický tým, že vyberie prvú možnosť:

```
some :: Parser s a -> DetPars s a
some p = snd . head. just p
```

- transformátor nedeterministického p analyzátoru na deterministický tým, nemení sa typ:

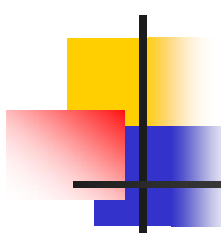
```
determ :: Parser a b -> Parser a b
determ p xs | null r = []
 | otherwise = [head r]
 where r = p xs
```

- pažravý – z iterácie  $\{\}^*$ , resp.  $\{\}^+$  vyberie prvú možnosť

```
greedy = determ . many
greedy1 = determ . many1
```

```
Main> many digit "1234"
[("",[1,2,3,4]),("4",[1,2,3]),("34",[1,2]),("234",[1]),("1234",[1])]
Main> greedy digit "1234"
[("",[1,2,3,4])]
```





# Čísła

---

- prirodzené čísla

```
natural :: Parser Char Int
natural = greedy1 digit <@ foldl f 0
 where f a b = a*10 + b
```

Cv10. Definujte natural pomocou foldr ;-)

- Celé čísla

```
integer :: Parser Char Int
integer = option (symbol '-') <*> natural <@ f
 where f ([],n) = n
 f (_,n) = -n
```

- racionálne čísla

```
fract :: Parser Char Float
fract = integer <*>
 (option (symbol '.') <*> natural <@ ???)
 <@ ???
```

Cv11. Doprogramujte fract, aby fract "12.345" = [("",12.345)

Cv12. Preštudujte si definíciu float



# Zátvorkovanie

Analyzátor pre dobre uzátvorkované výrazy podľa gramatiky:  $P \rightarrow ( P ) P \mid \varepsilon$

- prvá idea: priamočiaro prepíšeme pravidlá gramatiky, problém výst.typ:
- výstupnú hodnotu kódujeme do stromu:

```
data Tree = Nil | Bin (Tree, Tree)
```

```
parens :: Parser Char Tree
```

```
parens = (symbol '('
 <*> parens
 <*> symbol ')'
 <*> parens
) <@ (\(_, (x, (_, y))) -> Bin(x, y))
 <|> epsilon <@ const Nil
```

```
Main> parens "()"
[("", Bin (Nil, Nil)), ("()", Nil)]
Main> just parens "()"
[("", Bin (Nil, Nil))]
Main> just parens "()()
[("", Bin (Nil, Bin (Nil, Nil)))]
Main> just parens "(())"
[("", Bin (Bin (Nil, Nil), Nil))]
Main> just parens "()"
```

Cv13. Prečo sme v lambda patterne nepoužili štvoricu ?

Cv14. Načo je funkcia `const x y = x` ? Prečo sme tam rovno nenapísali `Nil` ?



# Zátvorkovanie ešte raz

```
open = symbol '('
close = symbol ')'
```

```
Main> nesting "(((())()))"
[("",3),("(((())())",0)]
Main> just nesting "(((())()))"
[("",3)]
```

```
parens :: Parser Char Tree
parens = (open *> parens <* close) <*> parens <@ Bin <|> succeed Nil
```

Cv15. Načo sú potrebné zátvorky okolo `open *> parens <*` ?

Cv16. Definujte analyzátor `parens2`, ktorý pozná dva druhy zátvoriek.

- hĺbka dobre uzátvorkovaného výrazu

```
nesting :: Parser Char Int
nesting = (open *> nesting <* close) <*> nesting <@ f <|> succeed 0
 where f (x,y) = (1+x) 'max' y
```

```
foldparens :: ((a,a)->a) -> a -> Parser Char a
```

```
foldparens f e = p
```

```
 where p = (open *> p <* close) <*> p <@ f <|> succeed e
```

Cv17. `foldparens` je zovšeobecnenie `parens` a `nesting`. Definujte ich s `foldparens`.



# Zátvorkovanie poslednýkrát

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

```
Main> just (listOf natural (symbol ','))
"12,34,567,8"
[("",[12,34,567,8])]
```

```
commaList :: Parser Char a -> Parser Char [a]
commaList p = listOf p (symbol ',')
```

```
Main> just (commaList natural) "12,34,567,8"
[("",[12,34,567,8])]
```

Cv18. Definujte analyzátor pre  $\lambda$ -termy.

```
Main> lambda "(\\x.(x x) \\x.(x x))"
[("",[APL (LAMBDA "x" (APL (ID "x") (ID "x")))) (LAMBDA "x" (APL (ID "x") (ID "x")))))]
```

Cv19. Definujte analyzátor palindromy.



# Cvičenia

---

- parser binárnej/hexa konštanty
- parser palindromov
- parser morseovej abecedy

|   |         |   |         |   |           |
|---|---------|---|---------|---|-----------|
| A | • -     | M | - -     | Y | - • - -   |
| B | - • • • | N | - •     | Z | - - • •   |
| C | - • - • | O | - - -   | 1 | • - - - - |
| D | - • •   | P | • - - • | 2 | • • - -   |
| E | •       | Q | - - • - | 3 | • • • - - |
| F | • • - • | R | • - •   | 4 | • • • -   |
| G | - - •   | S | • • •   | 5 | • • • • • |
| H | • • • • | T | -       | 6 | - • • • • |
| I | • •     | U | • • -   | 7 | - - • • • |
| J | • - - - | V | • • • - | 8 | - - - • • |
| K | - • -   | W | • - -   | 9 | - - - - • |
| L | • - • • | X | - • • - | 0 | - - - - - |



# Aritmetické výrazy

parser pre aritmetické výrazy

ľavo-rekurzívna gramatika:

```
<expression> ::=
 <expression> + <expression> |
 <expression> * <expression> |
 <expression> - <expression> |
 <expression> / <expression> |
 identifier |
 number |
 (<expression>)
```

gramatika LL(1):

```
<expression> ::=
 <term> |
 <term> + <expression> |
 <term> - <expression>
<term> ::=
 <factor> |
 <factor> * <term> |
 <factor> / <term>
<factor> ::=
 identifier |
 number |
 (<expression>)
```



# Aritmetické výrazy

```
data Expr = Con Int | Var String | Fun String [Expr] |
 Expr :+: Expr | Expr **: Expr | ...
```

```
fact :: Parser Char Expr
fact =
```

```
 <|> identifier
```

```
 <*> (option (parenthesized (commaList expr))
```

```
 <?@ (Var, flip Fun)
```

```
)
```

```
 <@ ap'
```

```
 <|> parenthesized expr
```

```
"?: " (just expr) "123"
[("","Con 123)]
```

```
"?: " (just expr) "foo(1,2)"
[("","Fun "foo" [Con 1,Con 2])]
```

```
term :: Parser Char Expr
```

```
term :: chainr fact
```

```
 (symbol '*' <@ const (:*:))
```

```
 <|> symbol '/' <@ const (:/:))
```

```
)
```

```
"?: " (just expr) "1*3"
[("","Con 1 **: Con 3)]
```

Cv20. Pozrite si definíciu `chainr :: Parser s a -> Parser s (a->a->a) -> Parser s a`



# Aritmetické výrazy

```
expr :: Parser Char Expr
```

```
expr = chainr term
```

```
 (symbol '+' <@ const (:+:)
 <|> symbol '-' <@ const (: -:)
)
```

```
type Op a = (Char, a->a->a)
```

```
gen :: [Op a] -> Parser Char a -> Parser Char a
```

```
gen ops p = chainr p (choice (map f ops))
```

```
 where f (s,c) = symbol s <@ const c
```

```
multis = [('*', (:*:)), ('/', (:/:))]
```

```
addis = [('+', (:+:)), ('-', (: -:))]
```

```
expr = gen addis term
```

```
term = gen multis fact
```

```
expr = addis 'gen' (multis 'gen' fact)
```

```
expr = foldr gen fact [addis, multis]
```

```
"?: " (just expr) "1+2*3"
```

```
[("","Con 1 :+: (Con 2 :*: Con 3))]
```

```
"?: " (just expr) "1*2+3"
```

```
[("","(Con 1 :*: Con 2) :+: Con 3)]
```