Syntaktická analýza v Haskelli

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

- Jeroen Fokker: Functional Parsers, LNCS 925, 1996, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.138.6885&rep=rep1&type=pdf
- Graham Hutton: **Functional Parsing – Computerphile** - youtube https://www.youtube.com/watch?v=dDtZLm7HIJs
- Johan Jeuring, Doaitse Swierstra: Grammars and Parsing, https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwj5hbiN9ujvAhVi8LsIHffDChYQFjAAegQIBhAD&url=http%3A%2F%2Fwww.staff.science.uu.nl%2F~jeuri101%2Fhomepage%2FPublications%2FMAIN.ps&usg=AOvVaw3h38GOhsCbV0KERUj_-4Od
- 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

Jazykovedné okienko

- **Gramatika** (konečná množina pravidiel): P -> (P) P | ε

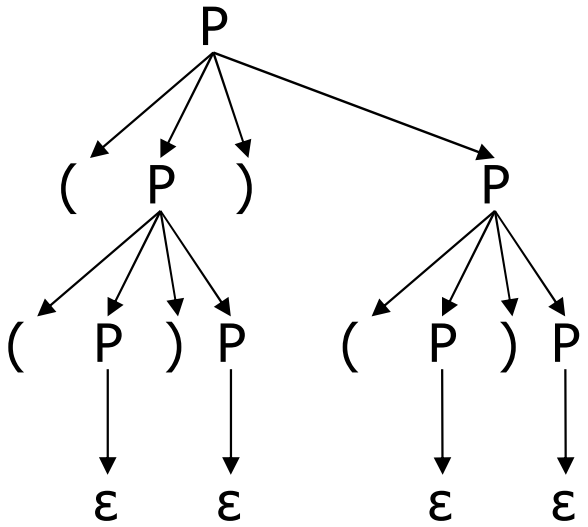
- **Odvodenie** (konečná postupnosť krokov odvodenia):

  P -> (P) P -> ((P)P)P -> ((P)P)(P)P ->^{4} (())()

- **Jazyk** (množina odvoditeľných slov):

  L = { w ∈ {(,)}*| P ->* w}

- **strom** odvodenia:



Aké sú to jazyky ?

R -> a R a | b R b | c R c | ε

T -> (T) | T T | ε

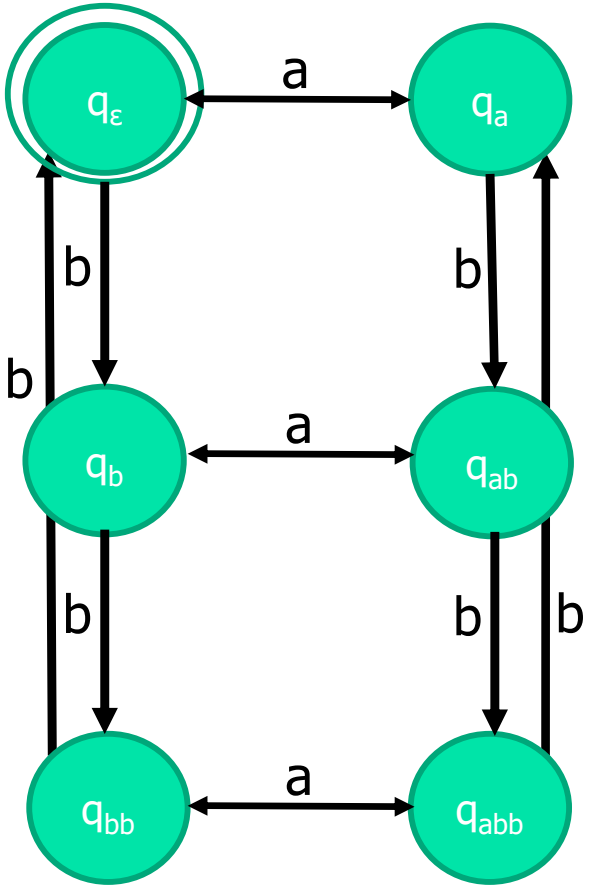
S -> S S | a

V -> a V a | ε
V -> a a V | ε
L -> (L L) | λ id.L | id

$\{w \in \{a,b\}^* | \ 2|\#_a w, \ 3|\#_b w\}$

- $S \rightarrow aS_a$ $\quad S \rightarrow bS_b$
- $S_a \rightarrow aS$ $\quad S_a \rightarrow bS_{ab}$
- $S_b \rightarrow aS_{ab}$ $\quad S_b \rightarrow bS_{bb}$
- $S_{ab} \rightarrow aS_b$ $\quad S_{ab} \rightarrow bS_{abb}$
- $S_{abb} \rightarrow aS_{bb}$ $\quad S_{abb} \rightarrow bS_a$
- $S_{bb} \rightarrow aS_{abb}$ $\quad S_{bb} \rightarrow bS$
- $S \rightarrow \varepsilon$

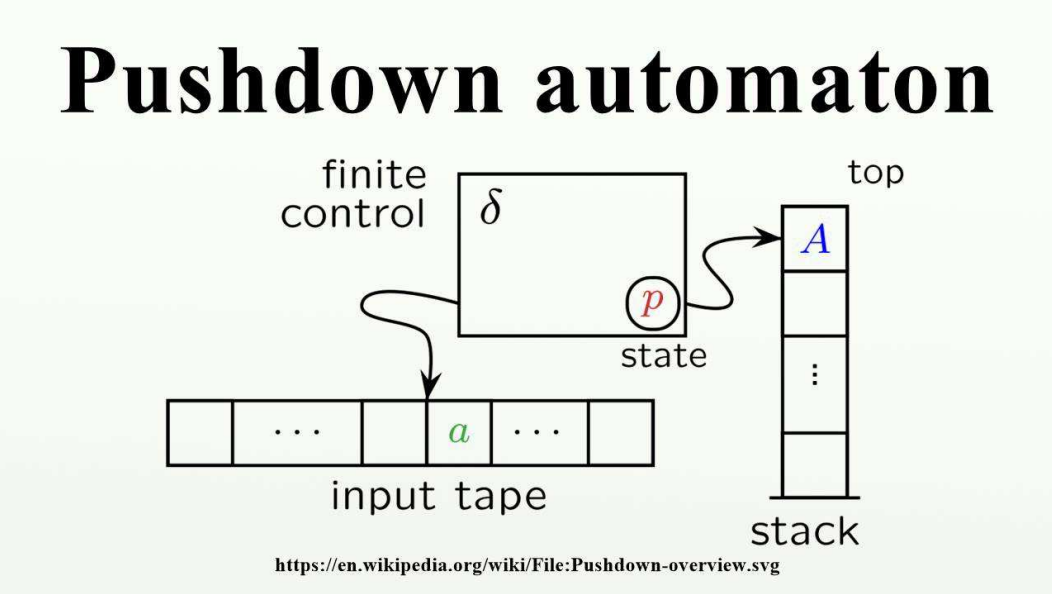
regulárny jazyk
regulárna gramatika
konečný automat



$\{ w \in \{a,b\}^* \mid \quad w = a^n b^n \}$

- Je to regulárny jazyk ?

- S -> aSb
- S -> ε



bezkontektový jazyk – stikne viac ako regulárny
bezkontextová gramatika
zásobníkový automat

$\{w \in \{a,b\}^* | \#_a w = \#_b w\}$

- Je to regulárny jazyk ?

- S -> aSb
- S -> ε
- ab -> ba
- ba -> ab

- S -> ASB
- S -> ε
- AB -> BA -- kontext
- A -> a
- B -> b

kontextová gramatika
??bezkontektový jazyk??
Turingov stroj

$\{w \in \{a,b\}^* | \#_a w = \#_b w\}$

- Je to bezkontextový jazyk ?

- S -> aSbS
- S -> bSaS
- S -> ε

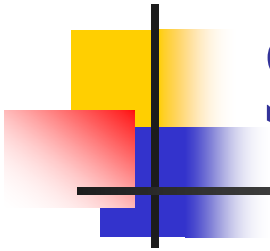
Dôkaz:

1) Ak S ->* w, tak $\#_a w = \#_b w$ ... jasné z pravidiel

2) Ak $\#_a w = \#_b w$, tak S ->* w ?

- w = aw', zrejme $\#_a w' = \#_b w' - 1$
- w' = ubv tak, že $\#_a u = \#_b u$, $\#_a v = \#_b v$.

bezkontextová gramatika
!!bezkontektový jazyk,
zásobníkový automat



Syntaktická analýza

- Je gramatika „Javy" bezkontextový jazyk, alebo nie (ergo kontextový)
- Existuje algoritmus syntaktickej analýzy pre bezkontextové gramatiky:
- Earley https://en.wikipedia.org/wiki/Earley_parser, O(n*n*n)
- A problém je redukovateľný na násobenie matíc

https://en.wikipedia.org/wiki/Earley_parser

Rekurzívny zostup

```
Args -> Term
Args -> Term ‚ Args

Term -> digit
Term -> variable
Term -> symbol
Term -> symbol ( Args )
```

```
-- neprázdny zoznam termov oddelených čiarkou
-- Args := Term { ‚ Term }*
fromStringArgs  :: String -> ([Term], String)
fromStringArgs xs = let (a, xs') = fromString xs in
                    if not (null xs') &&  head xs' == ',' then
                       let (as,xs'') = fromStringArgs (tail xs') in
                          ((a:as), xs'')
                    else ([a], tail xs')
-- Term := digit | Variable | symbol [ ( Args ) ]
fromString  :: String -> (Term, String)
fromString (x:xs)    | isDigit x  = (CN (ord x-48), xs)
fromString [x]       | isAlpha x && isLower x  = (Functor [x] [], [])
fromString (x:y:xs) | isAlpha x && isLower x && y == '(' = let (args, xs') =
                                                    fromStringArgs xs in (Functor [x] args, xs')
fromString (x:xs)   | isAlpha x && isUpper x  = (Var [x], xs)
fromString  xs      = error ("syntax error: " ++ xs)
```

```
>fromString "f(1,X)"
(f(1,X),"")
>fromString "f(1,g(2,Y))"
(f(1,g(2,Y)),"")
```

Aritmetické výrazy

Backus-Naurova forma

parser pre aritmetické výrazy

https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

l'avo-rekurzívna gramatika:

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

E -> E+E | E*E | E-E | E/E
E -> identifier
E -> number
E -> ( E )
```

gramatika LL(1):

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

RD-Parser

https://en.wikipedia.org/wiki/Recursive_descent_parser

ľavo-rekurzívna gramatika:

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

gramatika LL(1):

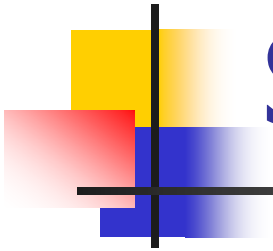
```
<expression> ::=
        <term> |
         <term> + <expression> |
         <term> - <expression>
<term> ::=
        <factor> |
        <factor> * <term> |
        <factor> / <term>
<factor> ::=
        identifier |
        number |
        ( <expression> )
```

```
fromString  :: String -> (AExpr, String)
fromString xs   =
        let (e1, xs') = fromString xs in
        let ('+', xs'') = xs' in
        let (e2, xs''') = fromString xs'' in
           ((Plus e1 e2), xs''')
```

OR ?

```
fromStringF  :: String -> (AFactor, String)
fromStringF (x:xs) | isDigit x   =  ...
fromStringF (x:xs) | isAlpha x   =  ...
fromStringF (x:xs) | x == '('   =
      let (e, xs') = fromString xs in
      (e, tail xs')
```

```
fromStringE  :: String -> (AExpr, String)
fromStringE xs   =
        let (t, xs') = fromStringT xs in
        if elem (head xs') ['+','-'] then
           let (e, xs'') = fromStringE (tail xs') in
              ((Plus/Minus t e), xs'')
        else
            (t, xs')
fromStringT  :: String -> (ATerm, String)
fromStringT xs   =
        let (f, xs') = fromStringF xs in
        if elem (head xs') ['*','/'] then
           let (t, xs'') = fromStringT (tail xs') in
              ((Mult/Div f t), xs'')
        else
            (f, xs')
```

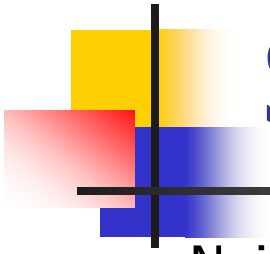


Syntaktický analyzátor

- 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 nezanalyzovanú 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. Čo ak neexistuje ? Čo 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 prestavuje nedeterministický analyzátor vstupného streamu objektov typu symbol (napr. Char), pričom výstupná reprezentácia je typu result:

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

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

p_2: Parser Char Int = [Char] -> [([Char], Int)]
p_2 "123" = [("123",0), ("23",1), ("3",12), ("",123)] -- viacero výsledkov
p_1:parser Char Char = [Char] -> [([Char], Char)]
p_1 "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 []                 = []          -- ak nie je nič na vstupe
symbola (x:xs)    | x=='a' = [ (xs, 'a') ]   -- ak je 'a' na vstupe
                  | otherwise= []           -- ak nie je 'a' na vstupe
```

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

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

P_a -> a

Príklad:

```
> symbol 'a' "abcd"
[("bcd",'a')]
> 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
```

```
> token "abc" "abcdef"
[("def","abc")]
>token "abc" "abdef"
[]
>token "try" "try { } catch"
[(" { } catch","try")]
```

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 Haskelli 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).

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

Triviálne analyzátory

(základné stavebné kamene)

- 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

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

```
> epsilon "abc"
[("abc",())]
> (succeed 77) "abc"
[("abc",77)]
> failp "abc"
[]
```

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

Kombinátory analyzátorov

(pojivo)

- definujme dva symboly v infixovej notácii

```
infixr 6 <*>                -- sekvenčné zreťazenie analyzátorov
infixr 4 <|>                -- zjednotenie analyzátorov
```

- sekvenčné zreťazenie 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
```

```
> ((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.

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

Transformátory analyzátorov

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

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

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

- transformátor **just** analyzátora 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
```

```
> 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.

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

Aplikátor analyzátorov

Aplikátor analyzátora 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'      -- char -> Int, '4' -> 4
```

```
> digit "12abc"
[("2abc",1)]
> satisfy isDigit "12abc"
[("2abc",'1')]
```

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

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).

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

Iterátory analyzátorov

- analyzátor p|ε (vôbec alebo raz, **reg.výraz {p}**):

```
option       ::  Parser s a -> Parser s [a]
option p     =   p  <@  single
                 <|> succeed []
```

```
single x = [x]
single = \x -> [x]
```

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

- analyzátor p* (vôbec, raz alebo viackrát, **reg.výraz {p}***):

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

- analyzátor p+ (aspoň raz, **reg.výraz {p}**$^+$):

```
many1        ::  Parser s a -> Parser s [a]
many1 p      =   p <:*> many p
identifier   = many1 (satisfy isAlpha)
```

```
> (many(symbol 'a') <*> symbol 'b') "aaabb"
[("b",("aaa",'b'))]
> (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  xx    =  foldr (<:*>) (succeed [])  xx
```

$\{a\}^{+}\{b\}^{*}\{c\}$

```
> 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/disjnukciu analyzátorov.

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

Deterministický analyzátor

type DetPars symbol result = [symbol] -> result

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

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

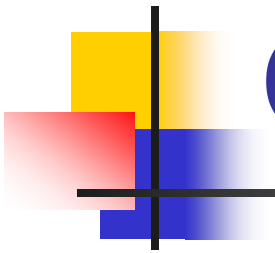
- transformátor nedeterministického p analyzátora na deterministický tým, nemení sa typ:

```
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
```

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



Čísla

- prirodzené čísla

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

Cv11. Definujte natural pomocou foldr/l ;-)

- 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  <@ ???)
                    <@ ???
```

Cv12. Doprogramujte fract, aby fract "12.345" = [("",12.345)

Cv13. Preštudujte si definíciu float

Zátvorkovanie

Analyzátor pre dobre uzátvorkované výrazy podľa gramatiky: P -> (P) P | ε

- 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
```

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

Cv14. Prečo sme v lambda patterne nepoužili štvoricu ?

Cv15. Načo je funkcia const x y = x ? Prečo sme tam rovno nenapísali Nil ?

Zátvorkovanie ešte raz

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

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

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

Cv16. Načo sú potrebné zátvorky okolo open *> parens <* ?

Cv17. Definujte analyzátor parens2, ktorý pozná dva druhy zátvoriek.

- hľbka 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
```

Cv18. 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 []
```

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

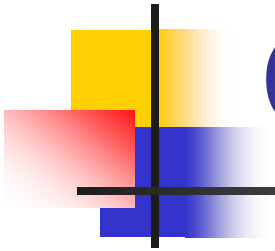
```
commaList:: Parser Char a -> Parser Char [a]
commaList p= listOf p (symbol ',')
```

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

Cv19. Definujte analyzátor pre λ-termy.

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

Cv20. Definujte analyzátor palindromy.



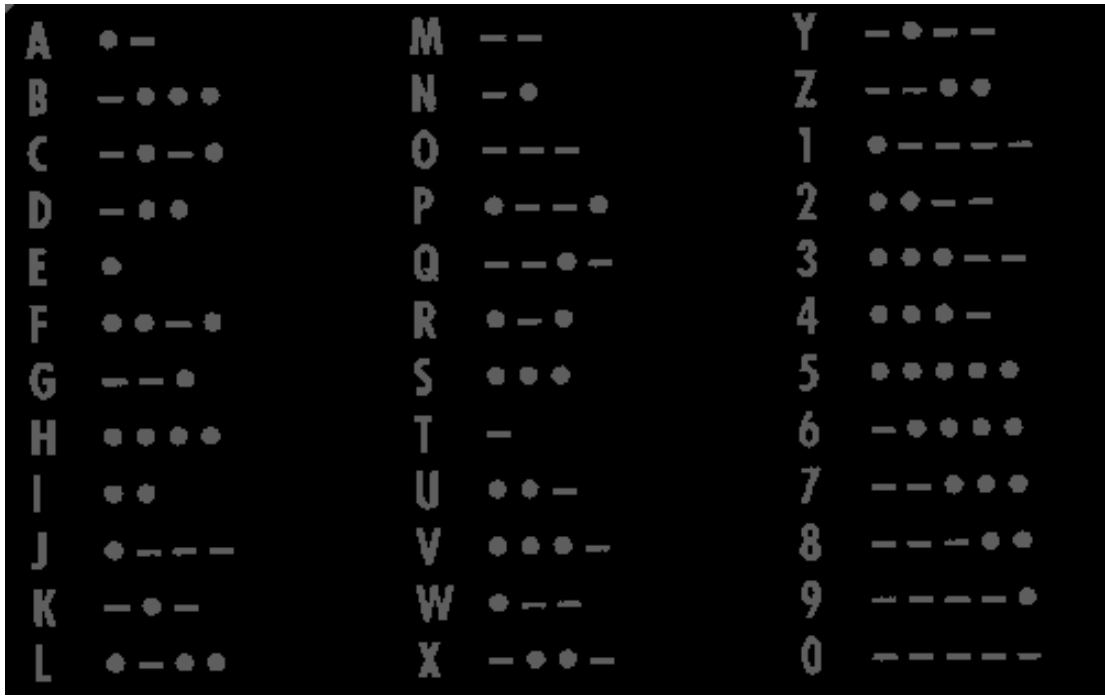
Cvičenie

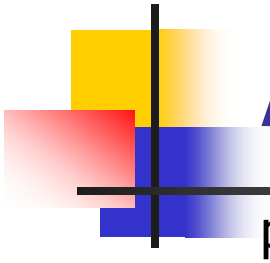
Cv21. Definujte analyzátory pre tieto gramatiky/jazyky:

- R -> a R a | b R b | c R c | ε
- S -> S S | a
- V -> a V a | ε
- T -> (T) | T T | ε
- W -> a W a | b W b | a | b | ε
- U -> aUbU | bUaU | ε
- $\{w \in \{a,b,c\}^* | \#_a w = \#_b w = \#_c w\}$

Cvičenia

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





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  =          integer <@ Con
         <|> identifier
                  <*> ( option (paranthesized (commaList expr))
                               <?@ (Var, flip Fun)
                      )
                   <@ ap'
         <|> parenthesized expr

term  :: Parser Char Expr
term  :: chainr fact
            (       symbol '*' <@ const (:*:)
              <|> symbol '/' <@ const (:/:)
            )
```

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

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

```
"?: " (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)]
```