



Monadické parsers

Graham Hutton, Erik Mejer: Functional Pearls: Monadic Parsing in Haskell, 1997

<http://www.cs.nott.ac.uk/~gmh/bib.html#pearl>

Graham Hutton, Erik Mejer: Monadic Parser Combinators

<http://www.cs.nott.ac.uk/~pszgmh/monparsing.pdf>

Daan Leijen: *Parsec, a fast combinator parser*

<https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/parsec-paper-letter.pdf>

Cieľom tejto prednášky je ukázať na problémy syntaktickej analýzy, ktorý sme začali minule, ideu fungovania monád a monadického štýlu programovania.

Mnohé analyzátory, ktoré skonštruujeme sa historicky nazývajú inými menami, ale ich analógie nájdete aj v predošlej prednáške.

Rôzne typy monád budeme študovať na budúcej prednáške.



Varovanie

- Namiesto parser.hs z minulej prednášky používame [mparser.hs](#)



Syntaktický analyzátor

Minule sme definovali analyzátor ako nasledujúci typ:

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

dnes to bude (zámena poradia – rešpektujúc použité zdroje):

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

resp.:

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

Primitívne parsersy:

```
return  :: a->Parser a
```

-- tento sa volal succeed

```
return v = \xs -> [(v,xs)]
```

-- nečíta zo vstupu, dá výsledok v

```
return v xs = [(v,xs)]
```

```
zero     :: Parser a
```

-- tento sa volal fail

```
zero     = \xs -> []
```

-- neakceptuje nič

```
zero xs  = []
```

```
item     :: Parser Char
```

-- akceptuje ľubovoľný znak

```
item     = \xs -> case xs of
```

-- tento znak dá do výsledku

```
    []      -> []
```

-- prázdny vstup, neakceptuje

```
    (v:vs)  -> [(v,vs)]
```

-- akceptuje znak v

```
"?: " parse (item) "abc"  
[('a',"bc")]
```



Kombinátory parserov

```
seq      :: Parser a -> Parser b -> Parser (a,b)      -- zret'azenie <*>
p `seq` q = \xs -> [ ((v,w),xs'') |
                      (v,xs') <- p xs,
                      (w,xs'') <- q xs']
"?: " parse (item `Main.seq` item) "abc"
[(['a','b'),('c')]
```

- tento kombinátor zbytočne vyrába vnorené výrazy typu
 $\backslash(a,(b,(c,d)))\rightarrow\dots$
- v tejto časti seq (<*>) upadne do zabudnutia, nahradí ho bind (>>=),
- nový kombinátor bind (>>=) kombinuje analyzátor $p::\text{Parser } a$ s funkciou $qf:: a \rightarrow \text{Parser } b$, ktorá ako argument dostane výsledok analýzy analyzátoru p a vráti analyzátor q:

```
bind      :: Parser a -> (a -> Parser b) -> Parser b
p `bind` qf = \xs -> concat [ (qf v) xs' | (v,xs')<-p xs]
```

- v je výsledok analýzy p,
- $(qf\ v)$ je nový analyzátor parametrizovaný výsledkom v.
- $\text{concat} :: [[a]] \rightarrow [a]$



Ako sa bind používa

`bind` $:: \text{Parser } a \rightarrow (a \rightarrow \text{Parser } b) \rightarrow \text{Parser } b$
`p `bind` qf` $= \backslash xs \rightarrow \text{concat } [(qf \ v) \ xs' \mid (v, xs') \leftarrow p \ xs]$

spôsob použitia:

```
p1 `bind` (\x1 ->  
p2 `bind` (\x2 ->  
...  
pn `bind` (\xn ->  
return (f x1 x2 . ... xn) ) )... )
```

preprogramujeme seq:

```
p `seq` q =  
  p `bind` (\x ->  
    q `bind` (\y ->  
      return (x,y) ) )
```

spôsob čítania:

najprv pustíme analyzátor p_1 , ktorého výsledok je v premennej x_1 ,
potom pustíme analyzátor p_2 , ktorého výsledok je v premennej x_2 ,
...
nakoniec pustíme analyzátor p_n , ktorého výsledok je v premennej x_n ,
výsledok celej analýzy dostaneme ako funkciu $f \ x_1 \ x_2 \ . \ \dots \ x_n$.



Príklady pre bind operátor

```
"?: " parse (item `bind` \x-> return x) "abc"  
[('a',"bc")]
```

```
"?: " parse (item `bind` \x-> item) "abc"  
[('b',"c")]
```

```
"?: " parse (item `bind` \x-> item `bind` \y -> return ((x:[])++(y:[]))) "abc"  
[("ab","c")]
```

```
"?: " parse (item `bind` \x-> item `bind` \y -> return ([x,y])) "abc"  
[("ab","c")]
```

```
"?: " parse ( return 4 `bind` \x->return 9 `bind` \y->return (x+y)) "abc"  
[(13,"abc")]
```



Príklady jednoduchých parserov

- ilustrujme na niekoľkých príkladoch použitie bind:

```
sat      :: (Char->Bool) -> Parser Char
sat pred = item `bind` \x->
    if pred x then return x else zero
```

```
Main> parse (item) "abcd"
[('a',"bcd")]
```

```
-- ten sa volal satisfy
-- akceptuje symbol, pre ktorý
-- platí predikát pred
```

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

```
Main> parse (sat isLower) "a123ad"
[('a',"123ad")]
```

```
-- ten sa volal symbol
-- akceptuje znak x
```

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

```
Main> parse (char 'a') "a123ad"
[('a',"123ad")]
```

```
-- akceptuje cifru
```

```
Main> parse (digit) "123ad"
[('1',"23ad")]
```



Zjednotenie

- zjednotenie/disjunkciu analyzátorov poznáme ako `<|>`:

```
plus      :: Parser a -> Parser a -> Parser a
p `plus` q = \xs -> (p xs ++ q xs)
p `plus` q xs = (p xs ++ q xs)
```

```
Main> parse (letter `plus` digit) "123abc"
[('1',"23abc")]
```

- definujme analyzátor word, ktorý akceptuje postupnosť písmen, `{letter}*:`
gramatická idea: `word -> letter word | ε`

```
word      :: Parser String
word      = nonEmptyWord `plus` return "" where
    nonEmptyWord = letter `bind` \x ->
                        word `bind` \xs ->
                        return (x:xs)
```

```
Main> parse word "abcd"
[("abcd", ""), ("abc", "d"), ("ab", "cd"), ("a", "bcd"), ("", "abcd")]
```

Problém je, že nechceme dostať toľko riešení, chceme *greedy* verziu...



Deterministické plus

- predefinujeme zjednotenie analyzátorov:

```
(+++)  
p +++ q      :: Parser a -> Parser a -> Parser a  
              = \xs -> case p `plus` q xs of  
                    []      -> []  
                    (x:xs) -> [x]  -- z výsledku zober prvé riešenie
```

```
word'      :: Parser String  
word'      = nonEmptyWord +++ return "" where  
    nonEmptyWord = letter `bind` \x ->  
        word' `bind` \xs ->  
        return (x:xs)
```

```
Main> parse word' "abcd12"  
[("abcd","12")]
```



Monády

Monáda je analógia algebraickej štruktúry nazývanej monoid s doménou M a binárnou asociatívnou operáciou $M \times M \rightarrow M$ s (ľavo a pravo)-neutrálnym prvkom.

- definícia v Haskell:

```
class Monad m where
```

```
return    :: a -> m a
```

-- neutrálny prvok vzhľadom na $>>=$

```
>>=      :: m a -> (a -> m b) -> m b
```

-- asociatívna operácia, nič iné ako bind

- chceli by sme vytvoriť Monad Parser t, ale musíme predefinovať typ Parser:

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

- to nás stojí trochu syntaktických zmien:

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

```
parse (Parser p) = p
```

-- inak: $\text{parse (Parser p) xs} = p \text{ xs}$

```
instance Monad Parser where
```

```
return v    = Parser(\xs -> [(v,xs)])
```

```
p >>= f      = Parser(
```

-- bind

```
\xs -> concat [ parse (f v) xs' | (v,xs')<-parse p xs])
```

bind je >>=

Monad comprehension

| | | |
|--|--|--|
| $p_1 \gg= \backslash x_1 \rightarrow$ | $[f \ x_1 \ x_2 \ . \ \dots \ x_n \mid$ | $\text{do } \{x_1 \leftarrow p_1 ;$ |
| $p_2 \gg= \backslash x_2 \rightarrow$ | $x_1 \leftarrow p_1$ | $x_2 \leftarrow p_2 ;$ |
| \dots | $x_2 \leftarrow p_2$ | \dots |
| $p_n \gg= \backslash x_n \rightarrow$ | \dots | $x_n \leftarrow p_n ;$ |
| $\text{return } (f \ x_1 \ x_2 \ . \ \dots \ x_n)$ | $x_n \leftarrow p_n]$ | $\text{return } (f \ x_1 \ x_2 \ . \ \dots \ x_n)$ |
| | | $\}$ |

```
string    :: String -> Parser String           -- volal sa token
string ""  = return ""
string (x:xs) = char x >>= \_ ->
                string xs >>= \_ ->
                return (x:xs)
```

-- výsledok char x zahod'
-- výsledok string xs zahod'
-- výsledok je celé slovo x:xs

```
string    :: String -> Parser String           -- prepíšeme do novej
string ""  = return ""                         -- syntaxi
string (c:cs) = do { _ <- char c; _ <- string cs; return (c:cs)} -- explicitné...
string (c:cs) = do {char c; string cs; return (c:cs)}           -- čitateľnejšie...
```

```
Main> parse (string "begin") "beginend"
[("begin","end")]
```



Iterátory $\{p\}^*$, $\{p\}^+$

- $\text{many}_p \rightarrow \text{many1}_p \mid \varepsilon$
 $\text{many0} \quad \quad \quad :: \text{Parser } a \rightarrow \text{Parser } [a]$
 $\text{many0 } p \quad \quad \quad = \text{many1 } p \text{ +++ return } []$
- $\text{many1}_p \rightarrow p \text{ many}_p$
 $\text{many1} \quad \quad \quad :: \text{Parser } a \rightarrow \text{Parser } [a]$
 $\text{many1 } p \quad \quad \quad = \text{do } \{a \leftarrow p; as \leftarrow \text{many } p; \text{return } (a:as)\}$

`Main> parse (many0 digit) "123abc"`
`[("123","abc")]`
- opakovanie p s oddel'ovačom sep : $\{p \text{ sep}\}^* p \mid \varepsilon$
 $\text{sepby} \quad \quad \quad :: \text{Parser } a \rightarrow \text{Parser } b \rightarrow \text{Parser } [a]$
 $p \text{ `sepby` sep} \quad = (p \text{ `sepby1` sep}) \text{ +++ return } []$

`"?: " parse (digit `sepby` char ',') "1,2,3abc"`
`[("123","abc")]`
- $p \{p \text{ sep}\}^*$
 $\text{sepby1} \quad \quad \quad :: \text{Parser } a \rightarrow \text{Parser } b \rightarrow \text{Parser } [a]$
 $p \text{ `sepby1` sep} \quad = \text{do } \{ a \leftarrow p; as \leftarrow \text{many}(\text{do } \{ _ \leftarrow \text{sep}; v \leftarrow p; \text{return } v \});$
 $\quad \quad \quad \text{return } (a:as) \}$

`Main> parse ((many0 digit) `sepby` (char '+' `plus` char '*')) "1+2*3abc"`
`[("1","2","3"),"abc"]`



Zátvorky

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

```
open      :: Parser Char
open      = char '('
close     = char ')'
```

■ verzia 1

```
paren     :: Parser ()
paren     = do { open; paren; close; paren; return () }
           +++
           return ()
```

```
" Main> parse (paren) "()(())"
[((),"")]
```

-- nezaujíma nás výstupná hodnota

■ verzia 2

```
data Bin = Nil | Node Bin Bin
           deriving(Show, Read, Eq)
```

-- vnútorná reprezentácia

```
parenBin  :: Parser Bin
parenBin  = do {
    open; x<-parenBin; close; y<-parenBin; return (Node x y) }
           +++
           return Nil
```

-- analyzátor zo String do Bin

```
Main> parse parenBin "()()()"
[(Node (Node Nil Nil) (Node Nil Nil),"")]
Main> parse parenBin "()()()"
[(Node Nil Nil,"()()")]
```

$P \rightarrow (P) P \mid [P] P \mid \epsilon$

Zátvorky 2

- naivne

```
parenBr      :: Parser ()  
parenBr      =  
do {  
    (open `plus` openBr) ; parenBr;  
    (close `plus` closeBr) ; parenBr; return () }  
+++ return ()
```

- reálne

```
parenBr      :: Parser ()  
parenBr      =  
do { open; parenBr; close; parenBr; return () }  
+++  
do { openBr ; parenBr; closeBr ; parenBr; return () }  
+++  
return ()
```

```
Main> parse parenBr "([[]])([[])"  
[((),"" )]  
Main> parse parenBr "([[([])])([[])"  
[((),"([[([])])([[])"])]
```



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> )
```



expr naivne

```
<expr> ::=
    <expr> + <expr> |
    <expr> - <expr> |
    <factor>
<factor> ::=
    nat |
    ( <expr> )
```

Gramatika je ľavo-rekurzívna, preto sa to zacyklí:

```
expr    = do {x <- expr; f <- addop; y<-expr; return (f x y)}
        `plus`
        factor
```

```
factor  = nat
        `plus`
        do { open; x<-expr; close; return x }
```

```
addop   = do { char '+'; return (+) } `plus` do { char '-'; return (-) }
```

- riešenie je zlé, čo ale stojí za zmienku je typ
addop :: Parser (Int -> Int -> Int)
- parser, ktorého vnútorná reprezentácia toho, čo zanalyzuje je funkcia,
- táto funkcia sa potom použije na kombináciu dvoch posebe-idúcich výsledkov iného parsera (f x z)



chainl

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \mid \langle \text{term} \rangle + \langle \text{expr} \rangle \mid \langle \text{term} \rangle - \langle \text{expr} \rangle$

$\langle \text{term} \rangle ::= \langle \text{factor} \rangle \mid \langle \text{factor} \rangle * \langle \text{term} \rangle \mid \langle \text{factor} \rangle / \langle \text{term} \rangle$

$\langle \text{factor} \rangle ::= \text{number} \mid (\langle \text{expr} \rangle)$

- Ak tú myšlienku zovšeobecníme dostaneme nasledujúci kód:

```
chainl1      :: Parser a -> Parser (a -> a -> a) -> Parser a
```

```
p `chainl1` op = do {a <- p; rest a}
```

```
    where
```

```
        rest a = do {f <- op; b <- p; rest (f a b)}
```

```
        `plus`
```

```
        return a
```

- Aritmetické výrazy:

```
expr      = term `chainl1` addop
```

```
term      = factor `chainl1` mulop
```

```
factor    = nat `plus` (do {open; n <- expr; close; return n})
```

```
mulop     = do { char '*'; return (*) } `plus` do { char '/'; return (div) }
```

```
Main> parse expr "1+2*3+10"
```

```
[(17,""),(8,"0"),(7,"+10"),(3,"*3+10"),(1,"+2*3+10")]
```

Cvičenia

- parser pre λ -calcul, t.j. Parser LExp:

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

- parser binárnej/hexa konštanty

```
Main> parse binConst "1101"  
[13,""]  
Main> parse hexaConst "FF"  
[255,""]
```

- parser palindromov,
t.j. Parser ()
- parser morseovej abecedy,
t.j. Parser String

| | | | | | |
|---|---------|---|---------|---|-----------|
| 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 | — — — — — |



Ako na Parsec

Daan Leijen autor knižnice

<https://hackage.haskell.org/package/parsec>

wiki:

<http://www.haskell.org/haskellwiki/Parsec>

nie na 1.čítanie

<http://book.realworldhaskell.org/read/using-parsec.html>

module Main where

import Text.ParserCombinators.Parsec

run :: Show a => Parser a -> String -> IO()

run p input = case (parse p "" input) of

Left err -> do { putStr "parse error at" ; print err }

Right x -> print x

```
run (char '!') "!123!"
run (oneOf "!.") "?123"
run letter "a"
run letter "123"
run digit "123"
run word "abc def"
run word "abc123"
```



Zátvorky 3

```
paren :: Parser ()           -- parser, ktorý nevracia žiadnu hodnotu
paren = do{ char '(' ; paren ; char ')' ; paren }
        <|> return ()
```

```
data Bin = Nil | Node Bin Bin   -- vnútorná reprezentácia
        deriving(Show, Read, Eq)
```

```
parenBin  :: Parser Bin       -- analyzátor zo String do Bin
parenBin  = do { char '('; x<-parenBin; char ')'; y<-parenBin; return (Node x y) }
        <|>
        return Nil
```

```
nesting :: Parser Int        -- parser, ktorý vracia hĺbku výrazu
nesting = do{ char '('; n<-nesting; char ')'; m<-nesting; return (max (n+1) m) }
        <|> return 0
```



Minského zápalkový stroj

stmt ::=

```
while <expr> do <stmt> end while
if <expr> then <stmt> [else <stmt>] end if
<id> ++
<id> --
```

expr ::=

```
<id> hrka
<id> nehrka
true
false
<expr> && <expr>
<expr> || <expr>
!<expr>
(<expr>)
```



Lexikálna analýza

```
import Text.ParserCombinators.Parsec.Token
import Text.ParserCombinators.Parsec.Language
```

```
lexer :: TokenParser ()
lexer = makeTokenParser( emptyDef
    { commentStart = "{-"
    , commentEnd   = "-}"
    , identStart   = letter
    , identLetter  = alphaNum
    , opStart      = oneOf "!&|"
    , opLetter     = oneOf "!&|"
    , reservedOpNames = ["!", "&&", "||"]
    , reservedNames = ["true", "false", "if", "then", "else", "end", "while", "do"]
    } )
```



Haskell & Java Style

```
import Text.ParserCombinators.Parsec.Language(haskellStyle)
import Text.ParserCombinators.Parsec.Language(javaStyle)
```

```
haskellStyle :: LanguageDef st
haskellStyle = emptyDef
  { commentStart  = "{-"
  , commentEnd    = "-}"
  , commentLine   = "--"
  , nestedComments = True
  , identStart    = letter
  , identLetter   = alphaNum <|> oneOf "_'"
  , opStart       = opLetter haskellStyle
  , opLetter      =
      oneOf " ! # $ % & * + . / < = > ? @ \\ ^ | - ~ "
  , reservedOpNames = []
  , reservedNames  = []
  , caseSensitive  = True
  }
```

```
javaStyle :: LanguageDef st
javaStyle  = emptyDef
  { commentStart = "/*"
  , commentEnd   = "*/"
  , commentLine  = "//"
  , nestedComments = True
  , identStart   = letter
  , identLetter  = alphaNum <|> oneOf "_'"
  , reservedNames = []
  , reservedOpNames = []
  , caseSensitive = False
  }
```



Vnútrotná reprezentácia výrazu

```
data Expr =  
  Var String HrkaNehrka |  
  Con Bool |  
  UnOp UnOp Expr |  
  BinOp BinOp Expr Expr  
  deriving Show
```

```
data HrkaNehrka =  
  Hrka |  
  Nehrka  
  deriving Show
```

```
data UnOp = Not deriving Show  
data BinOp = And | Or deriving Show
```

```
run hrkExpr "(A nehrka) && (B hrka)"  
BinOp And (Var "A" Nehrka)  
           (Var "B" Hrka)
```

```
run hrkExpr "((A nehrka) && (B hrka))"  
BinOp And (Var "A" Nehrka)  
           (Var "B" Hrka)
```

```
run hrkExpr "!((A nehrka) && (B hrka))"  
UnOp Not  
      (BinOp And  
        (Var "A" Nehrka)  
        (Var "B" Hrka))
```



```
import Text.ParserCombinators.Parsec.Expr
```



Parser výrazov

```
hrkExpr :: Parser Expr
hrkExpr = buildExpressionParser table factor <?> "hrk-expression"
table = [
    [Prefix ( do { _ reservedOp "!"; return (UnOp Not)} )],
    [Infix (do { _reservedOp "&&" ; return (BinOp And) }) AssocLeft],
    [Infix (do { _reservedOp "||" ; return (BinOp Or) }) AssocLeft]
]

factor = _parens hrkExpr
<|> (do { id <- _identifier ;
        kind <- do { _reserved "hrka"; return Hrka }
        <|>
        do { _reserved "nehrka"; return Nehrka };
        return (Var id kind)} )
<|> (do { _reserved "true" ; return (Con True)} )
<|> (do { _reserved "false" ; return (Con False)} )
```

Vnútroená reprezentácia príkazu

```
data Stmt =  
    Inc String |  
    Dec String |  
    If Expr Stmt Stmt |  
    While Expr Stmt |  
    Seq [Stmt]  
    deriving Show
```

```
run hrkStat "while A hrka && B nehrka do A++;B++;C-- end while"  
Seq [While (BinOp And (Var "A" Hrka) (Var "B" Nehrka))  
      (Seq [Inc "A",Inc "B",Dec "C"])]
```

```
run hrkStat "if ! A hrka then A++ else A-- end if"  
Seq [If (UnOp Not (Var "A" Hrka))  
      (Seq [Inc "A"])  
      (Seq [Dec "A"])]
```

hrkStat :: Parser Stmt
stmtparser :: Parser Stmt



Parser pre hrkStat

```
hrkStat = do { _whiteSpace; s<-stmtparser; eof; return s}
stmtparser = fmap Seq (_semiSep1 stmt1)
stmt1 = do {   v <- _identifier
              ; do { _reservedOp "++"; return (Inc v) }
                <|>
              do { _reservedOp "--"; return (Dec v) }
            }
    <|>
do {   _reserved "if"
      ; b <- hrkExpr
      ; _reserved "then"
      ; p <- stmtparser
      ; _reserved "else"
      ; q <- stmtparser
      ; _reserved "end" ; _reserved "if"
      ; return (If b p q)
    } <|> ...
```



Domáca úloha

- ľahšie

použité Parsec a napíšete analyzátor pre KSP-Turingov stroj

<http://dai.fmph.uniba.sk/courses/FPRO/source/tstroj.pdf>

výsledný analyzátor musí zvládnuť príklady zo zadania KSP

- ťažšie

použité Parsec a napíšete analyzátor pre KSP-Frontový Pascal

<http://dai.fmph.uniba.sk/courses/FPRO/source/froscal.pdf>

výsledný analyzátor musí implementovať gramatiku zo zadania KSP

- klasika

použité Parsec a napíšete analyzátor pre jazyk Brainf*ck

<https://en.wikipedia.org/wiki/Brainfuck>

v prípade, že parser nebude vaše dielo, budete ho musieť vedieť modifikovať