Funktory aplikatywne & Parsery kombinatoryczne

Grzegorz Łoś

27 października 2013

Spis treści

Klasa Functor

`Functor` to pudełko, którego zawartość możemy przetransformować.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Klasa Functor

`Functor` to pudełko, którego zawartość możemy przetransformować.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Prawa funktorów:

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

`fmap` jest "homomorfizmem".

Functor to czasem za mało

Chcemy pomnożyć liczby 5 oraz 3 schowane w pudełkach `Maybe`.

Functor to czasem za mało

Chcemy pomnożyć liczby 5 oraz 3 schowane w pudełkach `Maybe`.

```
ghci> let f = fmap (*) (Just 5)
ghci> :t f
f :: Maybe (Integer -> Integer)
```

Functor to czasem za mało

Chcemy pomnożyć liczby 5 oraz 3 schowane w pudełkach `Maybe`.

```
ghci> let f = fmap (*) (Just 5)
ghci> :t f
f :: Maybe (Integer -> Integer)
```

Zagadka:

```
ghci> :t fmap f (Just 3)
```

Functor to czasem za mało

Chcemy pomnożyć liczby 5 oraz 3 schowane w pudełkach `Maybe`.

```
ghci> let f = fmap (*) (Just 5)
ghci> :t f
f :: Maybe (Integer -> Integer)
```

Zagadka:

```
ghci> :t fmap f (Just 3)
```

```
<interactive>:1:6:
    Couldn't match expected type `a0 -> b0'
                with actual type `Maybe (Integer -> Integer)'
    In the first argument of `fmap', namely `f'
    In the expression: fmap f (Just 3)
```

Functor to czasem za mało

Chcemy pomnożyć liczby 5 oraz 3 schowane w pudełkach `Maybe`.

```
ghci> let f = fmap (*) (Just 5)
ghci> :t f
f :: Maybe (Integer -> Integer)
```

Zagadka:

```
ghci> :t fmap f (Just 3)
```

```
<interactive>:1:6:
    Couldn't match expected type ‘a0 -> b0’
                with actual type ‘Maybe (Integer -> Integer)’
    In the first argument of ‘fmap’, namely ‘f’
    In the expression: fmap f (Just 3)
```

Co zrobić, gdy funkcja także jest zamknięta w pudełku?

Functor to czasem za mało

Przy użyciu pattern-matchingu możemy wyłuskać funkcję:

Functor to czasem za mało

Przy użyciu pattern-matchingu możemy wyłuskać funkcję:

```
app :: Maybe (a -> b) -> Maybe a -> Maybe b
app Nothing _ = Nothing
app (Just f) x = fmap f x
```

Functor to czasem za mało

Przy użyciu pattern-matchingu możemy wyłuskać funkcję:

```
app :: Maybe (a -> b) -> Maybe a -> Maybe b
app Nothing _ = Nothing
app (Just f) x = fmap f x
```

Wszystko działa tak jak się spodziewamy:

```
ghci> let f = fmap (*) (Just 5)
ghci> app f (Just 3)
Just 15
```

Functor to czasem za mało

Przy użyciu pattern-matchingu możemy wyłuskać funkcję:

```
app :: Maybe (a -> b) -> Maybe a -> Maybe b
app Nothing _ = Nothing
app (Just f) x = fmap f x
```

Wszystko działa tak jak się spodziewamy:

```
ghci> let f = fmap (*) (Just 5)
ghci> app f (Just 3)
Just 15
```

- Dla list, typu `Either` i wielu innych stosujemy ten sam schemat.

Functor to czasem za mało

Przy użyciu pattern-matchingu możemy wyłuskać funkcję:

```
app :: Maybe (a -> b) -> Maybe a -> Maybe b
app Nothing _ = Nothing
app (Just f) x = fmap f x
```

Wszystko działa tak jak się spodziewamy:

```
ghci> let f = fmap (*) (Just 5)
ghci> app f (Just 3)
Just 15
```

- Dla list, typu `Either` i wielu innych stosujemy ten sam schemat.
- Potrzebna jest odpowiednia abstrakcja.

Klasa Applicative

Funktory aplikatywne to trochę lepsze funktory.

```
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Zdefiniowane w `Control.Applicative`.

Instancje Applicative: Maybe

```
instance Applicative Maybe where
    pure = Just
    (<*>) = app
```

Instancje Applicative: Maybe

```
instance Applicative Maybe where
    pure = Just
    (<*>) = app
```

```
ghci> :m Control.Applicative
ghci>  let f = fmap (*) ( Just 5 )
ghci> f <*> (Just 3)
Just 15
```

Instancje Applicative: Maybe

```
instance Applicative Maybe where
    pure = Just
    (<*>) = app

ghci> :m Control.Applicative
ghci>  let f = fmap (*) ( Just 5 )
ghci> f <*> (Just 3)
Just 15

ghci> Just (5*) <*> (Just 3)
Just 15
```

Instancje Applicative: Maybe

```
instance Applicative Maybe where
    pure = Just
    (<*>) = app

ghci> :m Control.Applicative
ghci>  let f = fmap (*) ( Just 5 )
ghci> f <*> (Just 3)
Just 15

ghci> Just (5*) <*> (Just 3)
Just 15

ghci> Just (*) <*> (Just 5) <*> (Just 3)
Just 15
```

Instancje Applicative: Maybe

Operator (<$>) poprawia czytelność.

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

Instancje Applicative: Maybe

Operator (<$>) poprawia czytelność.

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x

ghci> (*) <$> (Just 5) <*> (Just 3)
Just 15

ghci> (++) <$> Just "Hello " <*> Just "world!"
Just "Hello world!"
```

Instancje Applicative: List

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

Instancje Applicative: List

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

Typ `(<*>)` zawężony dla list:

```
(<*>) :: [a -> b] -> [a] -> [b]
```

Instancje Applicative: List

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

Typ `(<*>)` zawężony dla list:
`(<*>) :: [a -> b] -> [a] -> [b]`
Każda funkcja z listy z lewej jest aplikowana do każdego argumentu ze środkowej listy.

Instancje Applicative: List

```
ghci> [sin, sqrt.abs] <*> [-3.141592, 9]
[-6.5359e-7,0.4121,1.7725,3.0]
```

Instancje Applicative: List

```
ghci> [sin, sqrt.abs] <*> [-3.141592, 9]
[-6.5359e-7,0.4121,1.7725,3.0]

ghci> (\x y -> (x,y)) <$> [2,7] <*> ['A', 'B']
[(2,'A'),(2,'B'),(7,'A'),(7,'B')]
```

Instancje Applicative: List

```
ghci> [sin, sqrt.abs] <*> [-3.141592, 9]
[-6.5359e-7,0.4121,1.7725,3.0]

ghci> (\x y -> (x,y)) <$> [2,7] <*> ['A', 'B']
[(2,'A'),(2,'B'),(7,'A'),(7,'B')]
```

Zagadka:

```
ghci> length $ map (\x y z -> (x, y + z)) ['A'..'E']
        <*> [-4..5] <*> [-5..4]
```

Instancje Applicative: List

```
ghci> [sin, sqrt.abs] <*> [-3.141592, 9]
[-6.5359e-7,0.4121,1.7725,3.0]

ghci> (\x y -> (x,y)) <$> [2,7] <*> ['A', 'B']
[(2,'A'),(2,'B'),(7,'A'),(7,'B')]
```

Zagadka:

```
ghci> length $ map (\x y z -> (x, y + z)) ['A'..'E']
         <*> [-4..5] <*> [-5..4]
```

5 * 10 * 10 = 500

Instancje Applicative: List

```
ghci> [sin, sqrt.abs] <*> [-3.141592, 9]
[-6.5359e-7,0.4121,1.7725,3.0]

ghci> (\x y -> (x,y)) <$> [2,7] <*> ['A', 'B']
[(2,'A'),(2,'B'),(7,'A'),(7,'B')]
```

Zagadka:

```
ghci> length $ map (\x y z -> (x, y + z)) ['A'..'E']
         <*> [-4..5] <*> [-5..4]
```

5 * 10 * 10 = 500

Czy to jedyny sensowny sposób uczynienia listy instancją `Applicative`?

Instancje Applicative: IO

```
(<*>) :: IO (a -> b) -> IO a -> IO b

instance Applicative IO where
    pure = return
    a <*> b = do
        f <- a
        x <- b
        return (f x)
```

Instancje Applicative: IO

```
(<*>) :: IO (a -> b) -> IO a -> IO b

instance Applicative IO where
    pure = return
    a <*> b = do
        f <- a
        x <- b
        return (f x)
```

Każda monada może być funktorem aplikatywnym, (wystarczy spojrzeć na funkcje `liftM`, ap), ale nie na odwrót.

Instancje Applicative: IO

```
sequence :: [IO a] -> IO [a]

sequence [] = return []
sequence (c:cs) = do
    x <- c
    xs <- sequence cs
    return (x:xs)
```

Instancje Applicative: IO

```haskell
sequence :: [IO a] -> IO [a]

sequence [] = return []
sequence (c:cs) = do
    x <- c
    xs <- sequence cs
    return (x:xs)

sequence [] = return []
sequence (c:cs) = (:) <$> c <*> sequence cs
```

Instancje Applicative: ((->) r)

- Co to jest `((->) r)`?

Instancje Applicative: ((->) r)

- Co to jest `((->) r)`?
- Funkcje to pudełka.

Instancje Applicative: ((->) r)

- Co to jest `((->) r)`?
- Funkcje to pudełka.

```
fmap :: (a -> b) -> (r -> a) -> (r -> b)
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

Instancje Applicative: ((->) r)

- Co to jest `((->) r)`?
- Funkcje to pudełka.

```
fmap :: (a -> b) -> (r -> a) -> (r -> b)
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))

(<*>) :: (r -> a -> b) -> (r -> a) -> (r -> b)
instance Applicative ((->) r) where
    pure x = (\_ -> x)
    f <*> g = \x -> f x (g x)
```

Instancje Applicative: ((->) r)

```
ghci> (+) <$> sqrt <*> (^2) $ 4
18.0
```

Instancje Applicative: ((->) r)

```
ghci> (+) <$> sqrt <*> (^2) $ 4
18.0
```

Zagadka:

```
ghci> let f xs = if (even $ length xs)
                 then subtract
                 else (+)
ghci> f <*> sum <*> product $ [1..4]
```

Instancje Applicative: ((->) r)

```
ghci> (+) <$> sqrt <*> (^2) $ 4
18.0
```

Zagadka:

```
ghci> let f xs = if (even $ length xs)
                 then subtract
                 else (+)
ghci> f <*> sum <*> product $ [1..4]
```

14

Instancje Applicative: ((->) r)

Obliczanie wartości drzewa wyrażeń.

```
data Exp v = Var v
           | Val Int
           | Add (Exp v) (Exp v)
eval :: Exp v -> Env v -> Int
eval (Var x) env = fetch x env
eval (Val i) env = i
eval (Add p q) env = eval p env + eval q env
```

Korzystając z tego, że `((->) r)` jest funktorem aplikatywnym implementacja się upraszcza:

Instancje Applicative: ((->) r)

Obliczanie wartości drzewa wyrażeń.

```
data Exp v = Var v
           | Val Int
           | Add (Exp v) (Exp v)
eval :: Exp v -> Env v -> Int
eval (Var x) env = fetch x env
eval (Val i) env = i
eval (Add p q) env = eval p env + eval q env
```

Korzystając z tego, że `((->) r)` jest funktorem aplikatywnym implementacja się upraszcza:

```
eval (Var x) = fetch x
eval (Val i) = pure i
eval (Add p q) = (+) <$> eval p <*> eval q
```

Prawa funktorów aplikatywnych:

identity

```
pure id <*> v = v
```

composition

```
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```

homomorphism

```
pure f <*> pure x = pure (f x)
```

interchange

```
u <*> pure y = pure ($ y) <*> u
```

Pomocnicze operatory

```
(*>) :: Applicative f => f a -> f b -> f b
u *> v = pure (const id) <*> u <*> v

(<*) :: Applicative f => f a -> f b -> f a
u <* v = pure const <*> u <*> v

class Applicative f => Alternative f where
    -- The identity of '<|>'
    empty :: f a
    -- An associative binary operation
    (<|>) :: f a -> f a -> f a
```

Co to jest parser?

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

Co to jest parser?

```
type Parser  = String -> Tree

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

Co to jest parser?

```
type Parser = String -> Tree

type Parser = String -> (Tree, String)

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

Co to jest parser?

```
type Parser  = String -> Tree

type Parser  = String -> (Tree, String)

type Parser  = String -> [(Tree, String)]

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

Co to jest parser?

```
type Parser  = String -> Tree

type Parser  = String -> (Tree, String)

type Parser  = String -> [(Tree, String)]

type Parser a = String -> [(a, String)]

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

Podstawowe cegiełki

```
result :: a -> Parser a
result v = Parser $ \inp -> [(v,inp)]

ghci> parse (result "aaa") "2+2*2"
[("aaa","2+2*2")]
```

Podstawowe cegiełki

```
result :: a -> Parser a
result v = Parser $ \inp -> [(v,inp)]

ghci> parse (result "aaa") "2+2*2"
[("aaa","2+2*2")]

zero :: Parser a
zero = Parser $ \inp -> []

ghci> parse zero "2+2*2"
[]
```

Podstawowe cegiełki

```
result :: a -> Parser a
result v = Parser $ \inp -> [(v,inp)]

ghci> parse (result "aaa") "2+2*2"
[("aaa","2+2*2")]

zero :: Parser a
zero = Parser $ \inp -> []

ghci> parse zero "2+2*2"
[]

item :: Parser Char
item = Parser $ \inp -> case inp of
    [] -> []
    (x:xs) -> [(x,xs)]

ghci> parse item "2+2*2"
[('2',"+2*2")]
```

Składanie parserów

Bardziej skomplikowane parsery można tworzyć poprzez łączenie wyników prostszych parserów. W tym celu można uczynić `Parser` instancją klasy `Monad`.

Składanie parserów

Bardziej skomplikowane parsery można tworzyć poprzez łączenie wyników prostszych parserów. W tym celu można uczynić `Parser` instancją klasy `Monad`.

```
bind :: ParserFun a -> (a -> ParserFun b)
                    -> ParserFun b
bind p f = \inp ->
    concat [f v out | (v,out) <- p inp]
```

Składanie parserów

Bardziej skomplikowane parsery można tworzyć poprzez łączenie wyników prostszych parserów. W tym celu można uczynić `Parser` instancją klasy `Monad`.

```
bind :: ParserFun a -> (a -> ParserFun b)
                    -> ParserFun b
bind p f = \inp ->
    concat [f v out | (v,out) <- p inp]

instance Monad Parser where
    --return :: a -> Parser a
    return = result
    --(>>=) :: Parser a -> (a -> Parser b) -> Parser b
    (Parser p) >>= f = Parser $ bind p (parse . f)
```

Można także “uruchomić” niezależnie dwa parsery i połączyć ich wyniki.

Można także “uruchomić” niezależnie dwa parsery i połączyć ich wyniki.

```
instance MonadPlus Parser where
    -- mzero :: Parser a
    mzero = zero
    -- mplus :: Parser a -> Parser a -> Parser a
    (Parser p) `mplus` (Parser q) =
        Parser $ \inp -> (p inp ++ q inp)
```

Można także “uruchomić” niezależnie dwa parsery i połączyć ich wyniki.

```
instance MonadPlus Parser where
    -- mzero :: Parser a
    mzero = zero
    -- mplus :: Parser a -> Parser a -> Parser a
    (Parser p) `mplus` (Parser q) =
        Parser $ \inp -> (p inp ++ q inp)
```

Dla uproszczenia zapisu wprowadźmy operator (<+>):

```
infixl 3 <+>
(<+>) = mplus
```

Składanie parserów

Alternatywnym podejściem jest uczynienie parsera funktorem aplikatywnym.

Składanie parserów

Alternatywnym podejściem jest uczynienie parsera funktorem aplikatywnym.

```
instance Functor Parser where
    -- fmap :: (a -> b) -> Parser a -> Parser b
    fmap f (Parser p) = Parser $ \inp ->
        map (\(x,str) -> (f x, str)) $ p inp
```

Składanie parserów

Alternatywnym podejściem jest uczynienie parsera funktorem aplikatywnym.

```
instance Functor Parser where
    -- fmap :: (a -> b) -> Parser a -> Parser b
    fmap f (Parser p) = Parser $ \inp ->
        map (\(x,str) -> (f x, str)) $ p inp

instance Applicative Parser where
    -- pure :: a -> Parser a
    pure = result
    --(<*>) :: Parser (a -> b) -> Parser a -> Parser b
    (<*>) = ap
```

Zamiast `MonadPlus` można używać `Alternative`.

Zamiast MonadPlus można używać Alternative.

```
instance Alternative Parser where
    -- empty :: Parser a
    empty = zero
    -- (<|>) :: Parser a -> Parser a -> Parser a
    (Parser p) <|> (Parser q) =
        Parser $ \inp -> (p inp ++ q inp)
```

W dalszej części prezentacji będziemy przedstawiać parsery budowane w sposób monadyczny. Można to zrobić aplikatywnie – jest to zadaniem domowym.

Parsowanie znaków

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

Parsowanie znaków

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

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

Parsowanie znaków

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

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

ghci> parse (char 'd')  "Dom**2"
[]
ghci> parse (char 'D')  "Dom**2"
[('D',"om**2")]
```

Parsowanie znaków

```
digit :: Parser Char
digit = sat (\x -> '0' <= x && x <= '9')
```

Parsowanie znaków

```
digit :: Parser Char
digit = sat (\x -> '0' <= x && x <= '9')

lower :: Parser Char
lower = sat (\x -> 'a' <= x && x <= 'z')
```

Parsowanie znaków

```
digit :: Parser Char
digit = sat (\x -> '0' <= x && x <= '9')

lower :: Parser Char
lower = sat (\x -> 'a' <= x && x <= 'z')

upper :: Parser Char
upper = sat (\x -> 'A' <= x && x <= 'Z')
```

Parsowanie znaków

```
digit :: Parser Char
digit = sat (\x -> '0' <= x && x <= '9')

lower :: Parser Char
lower = sat (\x -> 'a' <= x && x <= 'z')

upper :: Parser Char
upper = sat (\x -> 'A' <= x && x <= 'Z')

letter :: Parser Char
letter = lower <+> upper
```

Parsowanie znaków

```
digit :: Parser Char
digit = sat (\x -> '0' <= x && x <= '9')

lower :: Parser Char
lower = sat (\x -> 'a' <= x && x <= 'z')

upper :: Parser Char
upper = sat (\x -> 'A' <= x && x <= 'Z')

letter :: Parser Char
letter = lower <+> upper

alphanum :: Parser Char
alphanum = letter <+> digit
```

Coraz bardziej skomplikowane parsery

```
string :: String -> Parser String
string "" = result ""
string (x:xs) = do
    char x
    string xs
    result (x:xs)

ghci> parse (string "while") "while(x<7)"
[("while","(x<7)")]
ghci> parse (string "while") "whill!!"
[]
```

Coraz bardziej skomplikowane parsery

```
word :: Parser String
word = do
         x <- letter
         xs <- word
         return (x:xs)
     <+> result ""

ghci> parse word "Hello!!word"
[("Hello","!!word"),("Hell","o!!word"),
("Hel","lo!!word"),("He","llo!!word"),
("H","ello!!word"),("","Hello!!word")]
```

Coraz bardziej skomplikowane parsery

```
word :: Parser String
word = do
         x <- letter
         xs <- word
         return (x:xs)
     <+> result ""

ghci> parse word "Hello!!word"
[("Hello","!!word"),("Hell","o!!word"),
("Hel","lo!!word"),("He","llo!!word"),
("H","ello!!word"),("","Hello!!word")]

many :: Parser a -> Parser [a]
many p = do
         x <- p
         xs <- many p
         return (x:xs)
     <+> result []
```

Coraz bardziej skomplikowane parsery

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

Coraz bardziej skomplikowane parsery

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

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

Coraz bardziej skomplikowane parsery

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

many1 :: Parser a -> Parser [a]
many1 p = do
        x <- p
        xs <- many p
        return (x:xs)

ghci> parse (many letter) "ABC"
[("ABC",""),("AB","C"),("A","BC"),("","ABC")]
ghci> parse (many1 letter) "ABC"
[("ABC",""),("AB","C"),("A","BC")]
ghci> parse ident "ABC"
[]
```

Parsowanie liczb

```
nat :: Parser Int
nat = liftM read (many1 digit)

ghci> parse nat "27-17"
[(27,"-17"),(2,"7-17")]
ghci> parse nat "-17+27"
[]
```

Parsowanie liczb

```
nat :: Parser Int
nat = liftM read (many1 digit)

ghci> parse nat "27-17"
[(27,"-17"),(2,"7-17")]
ghci> parse nat "-17+27"
[]

int :: Parser Int
int = do{_<- char '-'; n <- nat; return (-n)} <+> nat

ghci> parse int "-17+27"
[(-17,"+27"),(-1,"7+27")]
ghci> parse int "x-17+27"
[]
```

Parsowanie wartości odzielonych separatorem

```
sepby1 :: Parser a -> Parser b -> Parser [a]
p `sepby1` sep = do
    x <- p
    xs <- many (do
        sep
        y <- p
        return y )
    return (x:xs)
```

Parsowanie wartości odzielonych separatorem

```
sepby1 :: Parser a -> Parser b -> Parser [a]
p `sepby1` sep = do
    x <- p
    xs <- many (do
        sep
        y <- p
        return y )
    return (x:xs)

sepby :: Parser a -> Parser b -> Parser [a]
p `sepby` sep = (p `sepby1` sep) <+> (result [])

ghci> parse (sepby1 digit (char ';')) "1;2;13;1"
[("121","3;1"),("12",";13;1"),("1",";2;13;1")]
```

Parsowanie tablicy liczb

```
bracket :: Parser a -> Parser b -> Parser c
                    -> Parser b
bracket open p close = do
    open
    res <- p
    close
    return res
```

Parsowanie tablicy liczb

```
bracket :: Parser a -> Parser b -> Parser c
                    -> Parser b
bracket open p close = do
    open
    res <- p
    close
    return res

intArray :: Parser [Int]
intArray = bracket (char '{')
    (sepby int (char ',')) (char '}')
```

Parsowanie tablicy liczb

```
bracket :: Parser a -> Parser b -> Parser c
                    -> Parser b
bracket open p close = do
    open
    res <- p
    close
    return res

intArray :: Parser [Int]
intArray = bracket (char '{')
    (sepby int (char ',')) (char '}')

ghci> parse intArray "{12,67,5}"
[([12,67,5],"")]

ghci> parse intArray "{}"
[([],"")]

ghci> parse intArray "{12,67,5)"
[]
```

Parsowanie wyrażeń

Będziemy parsować wyrażenia arytmetyczne opisane gramatyką

```
expr ::= expr addop term | term
term ::= term multop factor | factor
factor ::= baseOrExp expop factor | baseOrExp
baseOrExp ::= integer | ( expr )
addop ::= + | -
multop ::= * | /
expop ::= **
```

Priorytet i łączność operatorów jest zgodna z typową konwencją.

Parsowanie wyrażeń

```
chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser
    a
p `chainl1` op = do
    x <- p
    fs <- many (do
        f <- op
        y <- p
        return (flip f y))
    return $ foldl (.) id fs x
```

Parsowanie wyrażeń

```
chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser
    a
p `chainl1` op = do
    x <- p
    fs <- many (do
        f <- op
        y <- p
        return (flip f y))
    return $ foldl (.) id fs x

expr :: Parser Int
expr = term `chainl1` addop
```

Parsowanie wyrażeń

```
chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser
    a
p `chainl1` op = do
    x <- p
    fs <- many (do
        f <- op
        y <- p
        return (flip f y))
    return $ foldl (.) id fs x

expr :: Parser Int
expr = term `chainl1` addop

term :: Parser Int
term = factor `chainl1` multop
```

Parsowanie wyrażeń

```
chainr1 :: Parser a -> Parser (a -> a -> a) -> Parser
    a
p `chainr1` op = do
    fs <- many (do
        y <- p
        f <- op
        return (f y))
    x <- p
    return $ foldr (.) id fs x
```

Parsowanie wyrażeń

```
chainr1 :: Parser a -> Parser (a -> a -> a) -> Parser
    a
p `chainr1` op = do
    fs <- many (do
        y <- p
        f <- op
        return (f y))
    x <- p
    return $ foldr (.) id fs x

factor :: Parser Int
factor = baseOrExp `chainr1` expop
```

Parsowanie wyrażeń

```
baseOrExp :: Parser Int
baseOrExp = nat <+> bracket (char '(') expr (char ')')
```

Parsowanie wyrażeń

```
baseOrExp :: Parser Int
baseOrExp = nat <+> bracket (char '(') expr (char ')')

parseOp :: String -> fun -> Parser fun
parseOp str fun = (string str >> return fun)
```

Parsowanie wyrażeń

```
baseOrExp :: Parser Int
baseOrExp = nat <+> bracket (char '(') expr (char ')')

parseOp :: String -> fun -> Parser fun
parseOp str fun = (string str >> return fun)

parseOps :: [(String, fun)] -> Parser fun
parseOps = foldr (\(s,f) acc ->
                (parseOp s f) <+> acc) zero
```

Parsowanie wyrażeń

```
baseOrExp :: Parser Int
baseOrExp = nat <+> bracket (char '(') expr (char ')')

parseOp :: String -> fun -> Parser fun
parseOp str fun = (string str >> return fun)

parseOps :: [(String, fun)] -> Parser fun
parseOps = foldr (\(s,f) acc ->
              (parseOp s f) <+> acc) zero

addop = parseOps [("+", (+)),
                  ("-", (-))]

multop = parseOps [("*", (*)),
                   ("/", (div))]

expop = parseOp "**" (^)
```

Parsowanie wyrażeń

Sprawdźmy naszą implementację!

```
ghci> parse expr "8/2/2"
[(2,""),(4,"/2"),(8,"/2/2")]
```

Parsowanie wyrażeń

Sprawdźmy naszą implementację!

```
ghci> parse expr "8/2/2"
[(2,""),(4,"/2"),(8,"/2/2")]

ghci> parse expr "2**3**2"
[(512,""),(8,"**2"),(2,"**3**2")]
```

Parsowanie wyrażeń

Sprawdźmy naszą implementację!

```
ghci> parse expr "8/2/2"
[(2,""),(4,"/2"),(8,"/2/2")]

ghci> parse expr "2**3**2"
[(512,""),(8,"**2"),(2,"**3**2")]

ghci> parse expr "2**3-(5-3**2)"
[(12,""),(8,"-(5-3**2)"),(2,"**3-(5-3**2)")]
```

Usuwanie białych znaków

```
spaces :: Parser ()
spaces = do
    many1 (sat Char.isSpace)
    return ()

comment :: Parser ()
comment = do
    string "--"
    many (sat (\x -> x /= '\n'))
    return ()

junk :: Parser ()
junk = do
    many (spaces <+> comment)
    return ()
```

Usuwanie białych znaków

```
clear :: Parser a -> Parser a
clear p = junk >> p

natural :: Parser Int
natural = clear nat

integer :: Parser Int
integer = clear int

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

Usuwanie białych znaków

```
clear :: Parser a -> Parser a
clear p = junk >> p

natural :: Parser Int
natural = clear nat

integer :: Parser Int
integer = clear int

symbol:: String -> Parser String
symbol xs = clear (string xs)

identifier :: [String] -> Parser String
identifier ks = clear $ do
    x <- ident
    if (elem x ks)
        then zero
        else return x
```

Literatura

- Monadic parser combinators. Graham Hutton, Erik Meijer. Technical report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.
- Applicative Programming with Effects. Conor McBride and Ross Paterson. 2008.
- http://learnyouahaskell.com/ ♡