Parsery funkcyjne

Typ reprezentujący parsery

Parser to funkcja przyjmująca napis i zwracająca

1. wartość

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

2. wartość i nieskonsumowaną część napisu

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

3. j.w. i lista pusta oznacza porażkę, a jednoelementowa sukces

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

Podstawowe parsery

- parser *item* kończy sie niepowodzeniem jeżeli wejściem jest `[]`, a w przeciwnym razie konsumuje pierwszy znak

```
item :: Parser Char
item []     = []
item (x:xs) = [(x, xs)]
```

- parser *failure* zawsze kończy się niepowodzeniem

```
failure :: Parser a
failure _ = []
```

- parser *return v* zwraca wartość *v* bez konsumowania wejścia

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

- parser *p* +++ *q* zachowuje się jak parser *p* jeżeli ten kończy się powodzeniem, a w przeciwnym razie jak parser *q*

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q = \inp -> case p inp of
                    [] -> q inp
                    [(v, out)] -> [(v, out)]
```

- funkcja *parse* aplikuje parser do napisu

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

Przykłady

```
ghci> parse item ""
[]

ghci> parse item "abc"
[('a',"bc")]

ghci> parse failure "abc"
[]

ghci> parse (return 1) "abc"
[(1,"abc")]

ghci> parse (item +++ return 'd') "abc"
[('a',"bc")]

ghci> parse (failure +++ return 'd') "abc"
[('d',"abc")]
```

Operator sekwencji

```
(>>=) :: Parser a -> (a -> Parser b) -> Parser b
p >>= f = \inp -> case parse p inp of
                    []      -> []
                    [(v, out)] -> parse (f v) out
```

Idea działania:

(p >>= f) inp --> p inp

porażka
---------> porażka

sukces (v,out)
----------------> (f v) out

Wyrażenie *do*

Równoważne:

1. ```
   p1 >>= (\v1 -> p2 >>= (\v2 -> p3 v1 v2))
   ```

2. ```
   do v1 <- p1
      v2 <- p2
      p3 v1 v2
   ```

Tzn. zaaplikuj parser p1 i rezultat nazwij v1, następnie zaaplikuj parser p2 i jego rezultat nazwij v2, na koniec zaaplikuj parser (p3 v1 v2)

Uwagi

- Wyrażenia po *do* muszą zaczynać się w tej samej kolumnie
- Rezultaty pośrednich parserów nie muszą być nazywane, jeśli nie będą potrzebne
- Wartość zwrócona przez ostatni parser jest wartością całego wyrażenia, chyba że któryś z wcześniejszych parserów zakończył się niepowodzeniem

Przykład

```
p :: Parser (Char, Char)
p = do x <- item
       item
       y <- item
       return (x, y)

ghci> parse p "abcdef"
[(('a','c'), "def")]

ghci> parse p "ab"
[]
```

Dalsze prymitywy

- parser *sat p* konsumuje i zwraca pierwszy znak jeśli ten spełnia predykat *p*, a w przeciwnym razie kończy się niepowodzeniem

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

- parsery cyfr i wybranych znaków

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

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

- funkcja *many* aplikuje parser wiele razy, kumulując rezultaty na liście, dopóki parser nie zakończy się niepowodzeniem

```
many :: Parser a -> Parser [a]
many p = many1 p +++ return []
```

- funkcja *many1* aplikuje parser wiele razy, kumulując rezultaty na liście, ale wymaga aby przynajmniej raz parser zakończył się sukcesem

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

Przykłady

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

ghci> parse digit "abc"
[]

ghci> parse (char 'a') "abc"
[('a',"bc")]

ghci> parse (many digit) "123abc"
[("123","abc")]

ghci> parse (many digit) "abc"
[("","abc")]

ghci> parse (many1 digit) "abc"
[]
```

Przykład

Parser kumulujący cyfry z napisu w formacie „[*cyfra*,*cyfra*,...]”

```
p :: Parser String
p = do char '['
       d  <- digit
       ds <- many (do char ','
                      digit)
       char ']'
       return (d:ds)
```

```
ghci> parse p "[1,2,3]"
("123","")

ghci> parse p "[1,2,3"
[]
```

Parser wyrażeń arytmetycznych

Niech wyrażenie może być zbudowane z cyfr, operacji '+' i '*' oraz nawiasów. Operacje '+' i '*' są prawostronnie łączne, a '*' ma wyższy priorytet niż '+'.

Gramatyka bezkontekstowa (e oznacza pusty napis):

```
expr   ::= term ('+' expr | e)

term   ::= factor ('*' term | e)

factor ::= digit | '(' expr ')'

digit  ::= '0' | '1' | ... | '9'
```

expr ::= term ('+' expr | e)

```
expr :: Parser Int
expr = do t <- term
          do char '+'
             e <- expr
             return (t + e)
           +++
             return t
```

term ::= factor ('' term | e)*

```
term :: Parser Int
term = do f <- factor
          do char '*'
             t <- term
             return (f * t)
           +++
             return f
```

factor ::= digit | '(' expr ')'

```
factor :: Parser Int
factor = do d <- digit
            return (digitToInt d)
          +++
            do char '('
               e <- expr
               char ')'
               return e
```

```
eval :: String -> Int
eval inp = case parse expr inp of
             [(n, [])] -> n
             [(_, out)] -> error ("nieskonsumowane " ++ out)
             [] -> error "bledne wejscie"

ghci> eval "2*3+4"
10

ghci> eval "2*(3+4)"
14

ghci> eval "2*3-4"
*** Exception: nieskonsumowane -4

ghci> eval "-1"
*** Exception: bledne wejscie
```

Klasy typów

Rodzaje polimorfizmu

- Polimorfizm parametryczny

```
elem x []                    = False
elem x (y:ys) | x == y       = True
              | otherwise    = elem x ys

elem :: ... a -> [a] -> Bool
```

- Przeładowanie (*overloading*)

```
(==) :: ... a -> a -> Bool

Jeśli Integer to  x == y  = integerEq x y
Jeśli krotka to   x == y  = tupleEq x y
Jeśli lista to    x == y  = listEq x y
```

Definiowanie klas typów

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

Typ a jest instancją klasy Eq jeżeli istnieją dla niego operacje (==) i (/=)

```
ghci> :t (==)
(==) :: Eq a => a -> a -> Bool
```

Jeżeli typ a jest instancją Eq, to (==) ma typ a -> a -> Bool

```
ghci> :t elem
elem :: Eq a => a -> [a] -> Bool
```

Deklarowanie instancji klas typów

```
data Mybool = Myfalse | Mytrue

ghci> elem Mytrue [Mytrue, Myfalse]
      No instance for (Eq Mybool) arising
      from a use of ‘elem’

instance Eq Mybool where
    Myfalse == Myfalse = True
    Mytrue == Mytrue   = True
    _ == _             = False
    x /= y             = not (x == y)
```

Mybool jest instancją Eq i definicja operacji (==) oraz (/=) jest następująca

```haskell
data Tree a = Empty | Node a (Tree a) (Tree a)

instance Eq a => Eq (Tree a) where
   Empty            == Empty            = True
   (Node a1 l1 r1) == (Node a2 l2 r2) = (a1 == a2) &&
                                        (l1 == l2) &&
                                        (r1 == r2)
                  _ == _               = False
                  x /= y               = not (x == y)
```

Jeżeli typ a jest instancją Eq, to (Tree a) jest instancją Eq i definicja operacji jest następująca

Metody domyślne

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y     = not (x == y)
```

Dziedziczenie (*inheritance*)

```
class Eq a => Ord a where
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a
    x < y                = x <= y && x /= y
```

Ord jest podklasą Eq (każdy typ klasy Ord musi być też instancją klasy Eq)

Uwaga

Dziedziczenie może być wielokrotne

Pamiętaj o kontekście

```
qsort :: [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort (filter (< x) xs)
               ++ [x]
               ++ qsort (filter (>= x) xs)
```

```
ghci> :load qsort.hs
      No instance for (Ord a) arising from use of '>='
      Possible fix: add (Ord a) to the type signature(s)
                    for 'qsort'
```

```
qsort :: Ord a => [a] -> [a]
```

Podstawowe klasy typów (Prelude.hs)

- Eq, Ord, Show, Read, Num, Enum

Klasa *Show*

```
class Show a where
    show :: a -> String
```

```
ghci> show 123
"123"
```

Klasa *Read*

```
read :: Read a => String -> a
```

```
ghci> read "123"
      Ambiguous type variable ‘a’

ghci> (read "123") :: Float
123.0

ghci> read "123" + 7
130
```

Klasa *Num*

```
class (Eq a, Show a) => Num a where
    (+), (-), (*) :: a -> a -> a
    negate        :: a -> a
    abs           :: a -> a
    x - y         =  x + negate y
    negate x      =  0 - x
```

```
ghci> 1.1 + 2.2
3.3
ghci> negate 3.3
-3.3
ghci> abs (-3)
3
```

Klauzula *deriving*

```
data Tree a = Empty | Node a (Tree a) (Tree a)
              deriving (Eq, Show)
```

Automatycznie utworzy instancje:

```
instance Eq a => Eq (Tree a) where ...
instance Show a => Show (Tree a) where ...
```

```
ghci> (Node 1 Empty Empty) == Empty
False
ghci> show (Node 1 Empty Empty)
"Node 1 Empty Empty"
```

Uwagi

- Klauzula `deriving` może być użyta do tworzenia instancji klas: `Eq`, `Ord`, `Show`, `Read`, `Enum`
- Przy tworzeniu instancji klasy `Ord`, uporządkowanie konstruktorów wynika z ich kolejności w definicji typu:

```
data Mybool = Myfalse | Mytrue deriving (Eq, Ord)

ghci> Myfalse < Mytrue
True
```

- W przypadku typów parametryzowanych ich parametry muszą być również instancjami odpowiednich klas

Klasa *Enum*

```
class Enum a where
    succ, pred :: a -> a
    toEnum     :: Int -> a
    fromEnum   :: a -> Int

data Day = Mon | Tue | Wed | Thu | Fri |
           Sat | Sun deriving (Show, Enum)

ghci> succ Mon
Tue
ghci> (toEnum 5) :: Day
Sat
ghci> fromEnum Mon
0
ghci> [Mon .. Fri]
[Mon,Tue,Wed,Thu,Fri]
```