Programowanie funkcyjne

HASKELL

Typ reprezentujący parsery

Parser to funkcja przyjmująca napis i zwracająca:

- wartość

  type Parser a = String -> a

- wartość i nieskonsumowaną część napisu

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

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

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

Podstawowe parsery

parser item kończy się 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 _ = []
```

```
*Main> item ""
[]
*Main> item "anna"
[('a',"nna")]
*Main> failure ""
[]
*Main> failure "anna"
[]
```

Podstawowe parsery

parser return v zwraca wartość v bez konsumowania wejścia

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

```
*Main> myreturn 1 "as"
[(1,"as")]
*Main> myreturn 2 "as"
[(2,"as")]
```

---

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)]
```

```
*Main> (item ++++ myreturn 'd') "as"
[('a',"s")]
*Main> (item ++++ myreturn 'd') "abcd"
[('a',"bcd")]
*Main> (item ++++ myreturn 'd') ""
[('d',"")]
*Main> (item ++++ myreturn 'd') "a"
[('a',"")]
```

Funkcja *parse*

funkcja parse aplikuje parser do napisu

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

```
*Main> myparse (myreturn 1) "aa"
[(1,"aa")]
*Main> myparse item ""
[]
*Main> myparse item "ala"
[('a',"la")]
*Main> myparse (myreturn 1) "ala"
[(1,"ala")]
*Main> myparse (myreturn 2) "ala"
[(2,"ala")]
```

Wyrażenie *do*

```
do  v1 <- p1
    v2 <- p2
    return (g v1 v2)
```

Oznacza: zaaplikuj parser p1 i rezultat nazwij v1,
następnie zaaplikuj parser p2 i jego rezultat nazwij v2,
na koniec zaaplikuj parser return (g v1 v2)

Uwagi

- Wartość zwrócona przez ostatni parser jest wartością całego wyrażenia, chyba że któryś z wcześniejszych parserów zakończył się niepowodzeniem
- Rezultaty pośrednich parserów nie muszą być nazywane, jeśli nie będą potrzebne

Przykład

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

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

Podstawowe parsery

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)
```

Podstawowe parsery

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)
```

Podstawowe parsery

```
Prelude> parse (many digit) "123abc"
[("123","abc")]
Prelude> parse (many digit) "abcdef"
[("","abcdef")]
Prelude> parse (many1 digit) "abcdef"
[]
```

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

Wyrażenia arytmetyczne

Niech wyrażenie będzie zbudowane z cyfr, operacji dodawania (+) i mnożenia (*) oraz nawiasów.
Operacje + i * są prawostronnie łączne, * ma wyższy priorytet niż +.

Gramatyka bezkontekstowa:

```
expr   ::= term ('+' expr | e)
term   ::= factor ('*' term | e)
factor ::= digit | '(' expr ')'
digit  ::= '0' | '1' | ... | '9'
```

```
expr   ::= term '+' expr | term
term   ::= factor '*' term | factor
```

Parser obliczający wartości wyrażeń arytmetycznych:

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

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

```
factor :: Parser Int
factor = do d <- digit
            return (read [d])
          +++
           do char '('
              e <- expr
              char ')'
              return e
eval :: String -> Int
eval input = case parse expr input of
               [(n, [])] -> n
               [(_, out)] -> error ("nieskonsumowane " ++ out)
               [] -> error "bledne wejscie"
```

```
*Main> eval "(3*2+2*6)*2"
36
*Main> eval "2+3"
5
*Main> eval "2+3*5"
17
*Main> eval "2*3+5"
11
*Main> eval "2*3-5"
*** Exception: nieskonsumowane -5
*Main> eval "(3+4)*(5+7)"
84
*Main> eval "(3*2+2*6)*2"
36
```

Literatura

- B.O'Sullivan,J.Goerzen, D.Stewart, Real World Haskell, O'REILLY, 2008.
- K.Doets, J.van Eijck, The Haskell Road to Logic, Math and programming, 2004.
- G.Brzykcy, A.Meissner, Programowanie w Prologu i programowanie funkcyjne, Wyd.PP, 1999.
- Miran Lipovaca, Learn You a Haskell for Great Good!