関数型プログラミング
第13回　モナドパーサ

萩野　達也
hagino@sfc.keio.ac.jp

Slide URL

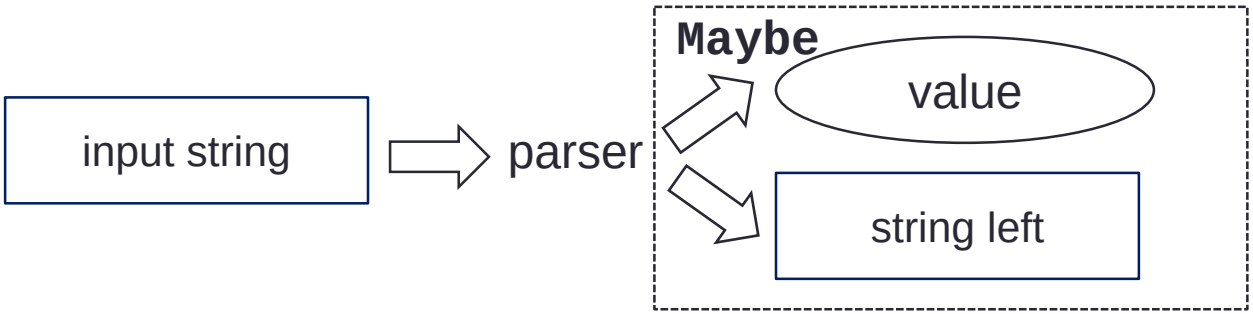
https://vu5.sfc.keio.ac.jp/slide/

モナド パーサ

- モナドを使って構文解析を行ってみましょう.

```
data Parser a = Parser (String -> Maybe (a, String))
```

- 字句解析も構文解析の一部に含めてしまいます.
- **Parser** がパーサのモナドです.
- モナドにするためには型変数を持つ型でなくてはいけません.
- **Parser** がデータコンストラクタです.
- パーサは文字列を受け取り, パースした結果と残りの文字列を返します.
- 構文解析は失敗するかもしれないため **Maybe** を使っています.



パーサを使う

- パーサがデータコンストラクタの中に入れられていて直接使うことができないので，パーサを呼ぶ関数を定義しておきます．

```
data Parser a = Parser (String -> Maybe (a, String))

parse::Parser a -> String -> Maybe (a, String)
parse (Parser p) cs = p cs
```

- **parse** はパーサに与えられた文字列を与えてパース結果を返す関数です．
- 例えば一文字だけを読み込みパーサは次のようになります．

```
parseOne::Parser Char
parseOne = Parser p
  where p [] = Nothing
        p (c:cs) = Just (c, cs)
```

- 実行してみることもできる．

```
> parse ParseOne "123"
Just ('1',"23")
```

Functor Parser

- モナドにする前に **Functor** のインスタンスにする必要があります.
- **Functor f** は **fmap** メソッドを持ちます.
  - **fmap::(a -> b) -> f a -> f b**

```
import Control.Applicative

instance Functor Parser where
  fmap f p = Parser (\cs -> do (v, cs1) <- parse p cs
                               return (f v, cs1))
```

- **Parser** の場合, **fmap** はパースした結果に関数を適用するパーサを作ります.
  - **parseChar::Parser Char**
  - **fmap isDigit parseChar::Parser Bool**

```
> parse (fmap isDigit ParseOne) "123"
Just (True,"23")
```

Applicative Parser

- 次に **Applicative** のインスタンスにします.
- **Applicative f** にするには2つのクラスメソッドを定義します.
  - **pure::a -> f a**
  - **(<*>)::f (a -> b) -> f a -> f b**

```
instance Applicative Parser where
  pure v = Parser (\cs -> return (v, cs))
  p <*> q = Parser (\cs -> do (f, cs1) <- parse p cs
                              (v, cs2) <- parse q cs1
                              return (f v, cs2))
```

- パーサの **pure** は何もパースしません.
- **<*>** は2つのパーサを順番に適用し, 最初のパーサの結果に2つ目のパーサの結果を適用します.
- なお **Applicative** の **pure** と **<*>** は次の規則を満たさなくてはいけません.
  - **pure id <*> v = v**
  - **pure (.) <*> u <*> v <*> w = u <*> (v <*> w)**
  - **pure f <*> pure x = pure (f x)**
  - **u <*> pure y = pure ($ y) <*> u**

Monad Parser

- これで準備が完了したので次に **Monad** のインスタンスにします.
- **Monad m** にするには2つのクラスメソッドを定義します.
  - **return::a -> m a**
  - **(>>=)::m a -> (a -> m b) -> m b**

```
instance Monad Parser where
  p >>= f = Parser (\cs -> do (v, cs1) <- parse p cs
                              parse (f v) cs1)
  return x = Parser (\cs -> return (x, cs))
```

- **return** は **Applicative** の **pure** と同じです.
- **p >>= f** は **p** がうまくパースできたときに, その結果に **f** を適用して次のパースを続けます. 連続してパースするときに使います.
- **p** が失敗したとき (**Nothing**) は **f** は呼ばれません.
- **Monad** の **do** 式を使うと, プログラムが読みやすくなります.
  - **p >>= (\x -> q)**
  - **do { x <- p; q }**

Alternative Parser

- さらに **Alternative** のインスタンスにすることで，パーサが書きやすくなります．
- **Alternative f** にするには2つのクラスメソッドを定義します．
  - **empty::f a**
  - **(<|>)::f a -> f a -> f a**

```
instance Alternative Parser where
  empty = Parser (\cs -> Nothing)
  p <|> q = Parser (\cs -> parse p cs <|> parse q cs)
```

- **empty** は何もしないパーサです．
- **p <|> q** は **p** がうまくパースできたときには **p** の結果で良く，うまくいかなかったときには **q** を試します．
  - Maybe も Alternative なので定義の中で使っています．
- いくつかのパーサを並べて適用できるものを探すのに使うことができます．また，**some** と **many** が定義されています．
  - **some::f a -> f [a]**
  - **many::f a -> f [a]**
- **some** は1つ以上の繰り返し，**many** は0個以上の繰り返しを表します．
- **Alternative** は **empty** と **<|>** で半群になっています．

パーサの構成(1)

- 1文字をパースするパーサはすでに定義しました.

```
parseOne::Parser Char
parseOne = Parser p
  where p [] = Nothing
        p (c:cs) = Just (c, cs)
```

- `parse parseOne "123"`
  `⇒ Just ('1', "23")`
- `parse parseOne ""`
  `⇒ Nothing`

- これを使って, その文字がある条件を満たすか調べるパーサを定義できます.

```
parseSat::(Char -> Bool) -> Parser Char
parseSat f = do x <- parseOne
                if f x then return x else empty
```

- `parse (parseSat isDigit) "123abc"`
  `⇒ Just ('1', "23abc")`
- `parse (parseSat isDigit) "abc"`
  `⇒ Nothing`

パーサの構成(2)

- 1文字目がある文字であるかを調べる.

```
parseChar::Char -> Parser Char
parseChar x = parseSat (== x)
```

- `parse (parseChar 'a') "abc"`
  `⇒ Just ('a', "bc")`
- `parse (parseChar 'a') "123"`
  `⇒ Nothing`

- **parseChar** を連続させて，最初がある文字列と一致するかを調べる.

```
parseString::String -> Parser String
parseString [] = return []
parseString (x:xs) = do parseChar x
                        parseString xs
                        return (x:xs)
```

- `parse (parseString "abc") "abcab"`
  `⇒ Just ("abc", "ab")`
- `parse (parseString "abc") "ababc"`
  `⇒ Nothing`

パーサの構成(3)

- 空白を読み飛ばすパーサ

```
parseSpace::Parser ()
parseSpace = do many (parseSat isSpace)
                return ()
```

- `parse parseSpace "   123"`
  `⇒ Just ((), "123")`
- `parse parseSpace "123"`
  `⇒ Just ((), "123")`

- 数字をパースするパーサ

```
parseNumber::Parser Int
parseNumber = do parseSpace
                 cs <- some (parseSat isDigit)
                 return (read cs)
```

- `parse parseNumber "  123 + 567"`
  `⇒ Just (123, " + 567")`
- 先頭の空白を読み飛ばす.

パーサの構成(4)

- 記号をパースするパーサ

```
parseSymbol::String -> Parser String
parseSymbol xs = do parseSpace
                    parseString xs
```

- `parse (parseSymbol "*") "   * 123"`
  `⇒ Just ("*", " 123")`
- `parse (parseSymbol "*") "   + 123"`
  `⇒ Nothing`

- **parseNumber** と **parseSymbol** 組み合わせることで，いろいろなパースが可能になる.

```
do x <- parseNumber
   parseSymbol "*"
   y <- parseNumber
   return (x * y)

parseSymbol "*" <|> parseSymbol "+"
```

数式の構文解析(1)

- 構文木を作らずに，そのまま評価することにする.
  - **parseExpr::Parser Int**
  - **parseTerm::Parser Int**

- 「項」の構文は
  - **term ::= number (("*" | "/") number)***

なので，「因子」を呼び出し，そのあと * か / を調べる.

```
parseTerm::Parser Int
parseTerm = do x <- parseNumber
               nextNumber x
  where nextNumber x = do parseSymbol "*"
                          y <- parseNumber
                          nextNumber (x * y)
                       <|>
                       do parseSymbol "/"
                          y <- parseFactor
                          nextNumber (x `div` y)
                       <|>
                       return x
```

数式の構文解析(2)

- 「式」の構文は
  - **`expr ::= term (("+" | "-") term)*`**

なので,「項」を呼び出し, そのあと + か - を調べる.

```
parseExpr::Parser Int
parseExpr = do x <- parseTerm
               nextTerm x
  where nextTerm x = do parseSymbol "+"
                        y <- parseTerm
                        nextTerm (x + y)
                     <|>
                     do parseSymbol "-"
                        y <- parseTerm
                        nextTerm (x - y)
                     <|>
                     return x
```

出力をつけて完成

- 入力された文字列を行ごとに分けて，パースした結果を出力する.

```
main::IO ()
main = do cs <- getContents
          putStr $
            unlines $
            map (showResult . parse parseExpr) $
            lines cs
  where showResult (Just (x,[])) =
                              "result = " ++ show x
        showResult _ = "error: syntax"
```

パーサの全体(1)

calcmp.hs

```
-- monad parser --

import Control.Applicative
import Data.Char

data Parser a = Parser (String -> Maybe (a, String))

parse::Parser a -> String -> Maybe (a, String)
parse (Parser p) cs = p cs

instance Functor Parser where
  fmap f p = Parser (\cs -> do (v, cs1) <- parse p cs
                               return (f v, cs1))

instance Applicative Parser where
  pure v = Parser (\cs -> return (v, cs))
  p <*> q = Parser (\cs -> do (f, cs1) <- parse p cs
                              (v, cs2) <- parse q cs1
                              return (f v, cs2))

instance Monad Parser where
  p >>= f = Parser (\cs -> do (v, cs1) <- parse p cs
                              parse (f v) cs1)
  return x = Parser (\cs -> return (x, cs))

instance Alternative Parser where
  empty = Parser (\cs -> Nothing)
  p <|> q = P (\cs -> parse p cs <|> parse q cs)
```

```
-- parser for calculator --

parseOne::Parser Char
parseOne = Parser p
  where p [] = Nothing
        p (c:cs) = Just (c, cs)

parseSat::(Char -> Bool) -> Parser Char
parseSat f = do x <- parseOne
                if f x then return x else empty

parseChar::Char -> Parser Char
parseChar x = parseSat (== x)

parseString::String -> Parser String
parseString [] = return []
parseString (x:xs) = do parseChar x
                        parseString xs
                        return (x:xs)

parseSpace::Parser ()
parseSpace = do many (parseSat isSpace)
                return ()

parseNumber::Parser Int
parseNumber = do parseSpace
                 cs <- some (parseSat isDigit)
                 return (read cs)

parseSymbol::String -> Parser String
parseSymbol xs = do parseSpace
                    parseString xs
```

パーサの全体(2)

```
-- parser for calculator (cont.) --

parseTerm::Parser Int
parseTerm = do x <- parseNumber
               nextNumber x
  where nextNumber x = do parseSymbol "*"
                          y <- parseNumber
                          nextFactor (x * y)
                       <|>
                       do parseSymbol "/"
                          y <- parseNumber
                          nextFactor (x `div` y)
                       <|>
                       return x

parseExpr::Parser Int
parseExpr = do x <- parseTerm
               nextTerm x
  where nextTerm x = do parseSymbol "+"
                        y <- parseTerm
                        nextTerm (x + y)
                     <|>
                     do parseSymbol "-"
                        y <- parseTerm
                        nextTerm (x - y)
                     <|>
                     return x
```

```
-- main --

main::IO ()
main = do cs <- getContents
          putStr $
            unlines $
            map (showResult . parse parseExpr) $
            lines cs
  where showResult (Just (x,[])) =
                              "result = " ++ show x
        showResult _ = "error: syntax"
```

実行例

```
% ./calcmp
1+2
result = 3
1 +2* 3  -4/ 5
result = 7
1 2
error: syntax
1+x-5
error: syntax
```

練習問題13

- ここで紹介したモナドパーサを利用した電卓の計算結果が分数になるように修正しなさい.
  - 0での割り算が発生すると, エラーを出力するようにしなさい.
  - それ以外のエラーと区別しなさい.

実行例

```
% ./calcmp
5 + 4 * 3 / 2
11
5+2/0 - 3
error: division by 0
1/2-1/6
1/3
(1+2)*3
error: syntax
```