関数型プログラミング
第11回　モナドパーサ

萩野　達也
hagino@sfc.keio.ac.jp

Slide URL

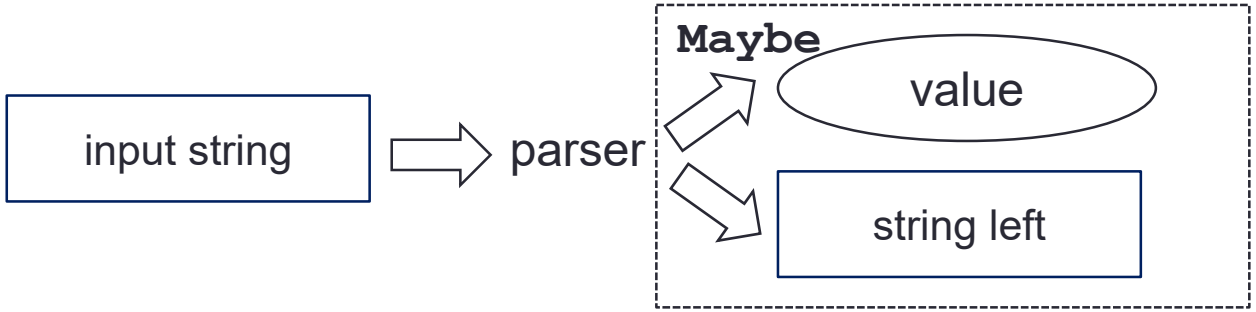
https://vu5.sfc.keio.ac.jp/slide/

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ます．

```
> 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 Integer
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, parseTerm, parseFactor::Parser Integer`

- 「因子」の構文は
  - `factor ::= number | "(" expr ")"`

  なので，「数字」または「(」の場合には「式」を呼び出す.

```
parseFactor::Parser Integer
parseFactor = parseNumber
              <|>
              do { parseSymbol "(" ;
                   x <- parseExpr ;
                   parseSymbol ")" ;
                   return x }
```

数式の構文解析(2)

- 「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 Integer
parseTerm = parseNumber >>= nextFactor where {
  nextFactor x = do { parseSymbol "*" ;
                      y <- parseNumber ;
                      nextFactor (x * y) }
                 <|>
                 do { parseSymbol "/" ;
                      y <- parseNumber ;
                      nextFactor (x `div` y) }
                 <|>
                 return x
}
```

数式の構文解析(3)

- 「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 Integer
parseExpr = parseTerm >>= nextTerm 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 = getContents >>=
          putStr . unlines . map process . lines  where {
  process cs = showResult $ parse parseExpr cs ;
  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 = Parser (\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 Integer
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.) -

parseFactor::Parser Integer
parseFactor = parseNumber
              <|>
              do { parseSymbol "(" ;
                   x <- parseExpr ;
                   parseSymbol ")" ;
                   return x }

parseTerm::Parser Integer
parseTerm = parseFactor >>= nextFactor where {
  nextFactor x = do { parseSymbol "*" ;
                      y <- parseFactor ;
                      nextFactor (x * y) }
                 <|>
                 do { parseSymbol "/" ;
                      y <- parseFactor ;
                      nextFactor (x `div` y) }
                 <|>
                 return x
}
```

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

-- main --

main::IO ()
main = getContents >>= putStr .
            unlines .
            map (showResult . parse parseExpr) .
            lines    where {
  showResult (Just (x,[])) = "result = " ++ show x ;
  showResult _ = "error: syntax"
}
```

実行例

```
% stack runghc 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11－1

- 電卓の因子の構文を次のように拡張し
  - +4
  - -(2+3)

  などを取り扱えるようにしなさい.

追加

```
factor ::= number | "(" expr ")" | "-" factor | "+" factor
```

calcmp.hs

```
...
parseFactor::Parser Integer
parseFactor = parseNumber
              <|>
              do { parseSymbol "(" ;
                   x <- parseExpr ;
                   parseSymbol ")" ;
                   return x }
              <|>
              ...
```

練習問題11－2

- 分数を定義したと同じように複素整数を定義しなさい.
  - 複素整数$(a + b\,i)$とは，実部および虚部ともに整数の複素数のことです.

```
data Complex = Complex Integer Integer
```

- **Complex**を**Show**および**Num**クラスのインスタンスにしなさい．複素数の四則演算は次のようになっています.
  - $(a + b\,i) + (c + d\,i) = (a + c) + (b + d)\,i$
  - $(a + b\,i) - (c + d\,i) = (a - c) + (b - d)\,i$
  - $(a + b\,i) \times (c + d\,i) = (ac - bd) + (ad + bc)\,i$
  - $(a + b\,i) \div (c + d\,i) = \frac{ac+bd}{c^2+d^2} + \frac{bc-ad}{c^2+d^2}\,i$

Numの**abs**と**signum**は次のように適当に定義しなさい.

complex.hs

```
data Complex = Complex Integer Integer
instance Show Complex where {
...
}
instance Num Complex where {
  (Complex a b) + (Complex c d) = ... ;
  (Complex a b) * (Complex c d) = ... ;
  negate (Complex a b) = Complex (-a) (-b) ;
  signum (Complex a b) = Complex (signum a) 0;
  abs (Complex a b) | a < 0 = negate (Complex a b)
                    | otherwise = Complex a b ;
  fromInteger a = Complex a 0
}
main = print $ Complex 1 2 + Complex 3 4
```

練習問題11－3

- ここで紹介したモナドパーサを拡張し複素整数を扱うことのできる電卓にしなさい.

```
factor ::= complex | "(" expr ")" | "+" factor | "-" factor

complex ::= "i" | number "i" | number
```

- なお複素整数の割り算は**Num**では定義していないので，別の関数として定義する必要があります.
  - `complexdiv (Complex a b) (Complex a b) = ...`
- **`parseComplex::Parser Complex`** を定義して複素整数をパースできるようにしなさい.
- **`parseExpr`** などの型は **`Parser Integer`** から **`Parser Complex`** に変更しなさい.

実行例

```
% stack runghc calcmpi.hs
i*i
result = -1
1+i
result = 1+1i
5/(1+2i)
result = 1-2i
(2+3i)*(2-3i)
result = 13
```

その他のHaskellの話題

- レイアウト構文
  - インデントにより構文を指定する
- ファイル入出力
  - 標準入出力以外のファイルの読み書き
- ネットワーク処理
  - ソケットを使ったネットワーク処理

授業内期末試験について

- 期末試験は1月17日の授業内で行います.
  - 教室に来てください.
  - オンラインでの参加は認めません.

- 持ち込み条件
  - Haskellの動作するPCを1台のみ持ち込み可能とします.
  - 自筆の手書きノートは持ち込んでもかまいません.
  - PCによる通信はSOLの参照および下記の課題提出サイトのみとします.
  - その他の通信は不正とみなします.

- 試験方法
  - 問題用紙を紙で配ります.
  - 作成したHaskellのソースプログラムを下記の課題提出サイトから授業時間内に登録してください.

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