

Parsec による簡易インタプリタの実装と構文の 設計

機械科学・航空宇宙学科 3 年 1w192224 田久健人

January 23, 2023

はじめに

進捗 →

https://github.com/tkyawa/project_research_a

abstract

Haskell のパーサコンビネータの Parsec を用いた簡易なインタプリタの実装を通して、パーサコンビネータの雰囲気を感じ、実際に構文の設計を行う。

目次

- ① モナディックパーサ概要
 - Monad の概要
 - Parsec の各パーサについて
 - 四則演算のインタプリタ
- ② Parsec による IMP の実装
 - IMP の概要
 - 実装
 - IMP の構文上の問題点
- ③ IMP の改良
- ④ 今後の展開

Monad とは

(Haskell の型クラスとしての)Monad . . .

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Monad とは

Monad の利点 . . .

- 状態の扱いを，副作用を気にせずに行える場合がある
- 関数合成等ができる

状態について

"状態をもつ" 例 . . .

```
add: x -> e:e + 1
```

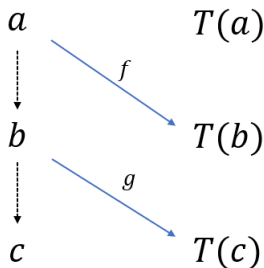
関数呼び出しの度に異なる結果が返る

状態について

定義 1.1

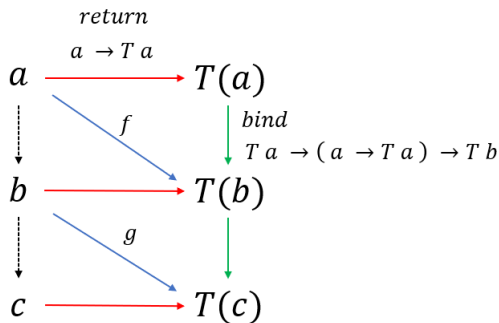
ある値 x が状態を持つことを $T(x)$ と表記する

状態を用いた Monad 概要図①



$a \rightarrow T(b), b \rightarrow T(c)$ の関数合成はできない

状態を用いた Monad 概要図②



bind, return によって関数合成っぽいことができる

Monad の例 1: Maybe ①

```
data Maybe a = Nothing
              | Just a
              deriving (Eq, Ord)
```

```
instance Monad Maybe where
  (Just x) >>= f = f x
  Nothing >>= f = Nothing
  return x = Just x
```

bind の実装が, Just と Nothing のパターンマッチ

Monad の例 1: Maybe ②

モナド値が取りうる全ての状態に対しパターンマッチを行うことにより, bind が実装されている

→ 抽象的な概念と実装の橋渡し

Monad の例 2:State

```
instance Monad (State s) where
  f >>= m = State $ \s ->
    let (s', a) = runState f s
    in runState (m a) s'
```

→ 取り出した値の状態に対するパターンマッチで bind を実装

Parser Monad ①

パーサを Monad として扱うことを考える.

`Parser :: String -> AST`

↓

`Parser :: String -> (AST, String)`

↓

`Parser :: String -> (a, String)`

Parser Monad ②

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

```
instance Monad Parser where
```

```
    return a = Parser $ \cs -> [(a,cs)]
```

```
    p >>= f    =
```

```
    Parser $ \cs -> let l = parse p cs
```

```
                    ll = map func l
```

```
                    in concat ll
```

```
    where
```

```
        func (a,cs') = parse (f a) cs'
```

最小構成のパーサ

一文字を受理するパーサ

```
item :: Parser Char
item = Parser $ \cs -> case cs of
    ""      -> []
    (c:cs') -> [(c,cs')]
```

必ず失敗するパーサ

```
failure :: Parser a
failure = \inp -> []
```


最小構成のパーサ

各パーサは, bind により合成可能, do 記法が使用できる

四則演算の実装

全体の構成:

AST: data 型で表現

Lexer, Parser: Parsec の各関数で実装

Eval: Haskell に渡す

```
data AST = ...
```

```
(lexer :: String -> Parser String)
```

```
parser :: Parser AST
```

```
eval :: AST -> Integer
```

演算子順位の実装

演算子順位は，パーサ実装で(おそらく)最初に頭を使う(面白い，難しい)部分

Parsec では，`buildExpressionParser` を用いて直感的(?)に実装可能

buildExpressionParser ①

実装例:

```

expr :: Parser Expr
expr = buildExpressionParser
      [[binary "*" Mul AssocLeft,
        binary "/" Div AssocLeft]
       , [binary "+" Add AssocLeft,
          binary "-" Sub AssocLeft, prefix "-" Negate]
      ]
      atom
where
  binary name fun assoc =
    Infix (reservedOp name >> return fun) assoc
  prefix name fun = Prefix (fun <$ reservedOp name)

```

buildExpressionParser ②

つづき:

```
atom :: Parser Expr
atom = do symbol "("
          x <- expr
          symbol ")"
          return x
        <|> (Const <$> natural)
```

buildExpressionParser を用いない実装

一般的には...

- 構文段階 or パース段階で再帰的な定義を与える
- chainl 等を用いる

デモ①

(いろんな四則演算)

まとめ

Parser を Monad で実装することで、幅広いパーサ関数を実装できた

IMP 概要

IMP is a simple imperative language

IMP の構文 (BNF)

```
Command ::= Command; Command
          | if BExpr then Command else Command
          | while BExpr do Command
          | String := AExpr
          | Skip
```

```
BExpr ::= true
        | false
        | AExpr < AExpr
```

```
AExpr ::= integer
        | var
        | AExpr + AExpr
        | AExpr × AExpr
```

全体の構成

ファイル構成:

```
IMP - - Syntax.hs
    | - Parser.hs
    | - Env.hs
    | - Eval.hs
    | - Main.hs
```

Syntax.hs

構文を AST で, AST をデータ型で表現

```
data Command = Seq [Command]
              | If BExpr Command Command
              | While BExpr Command
              | Assign String AExpr
              | Skip
              deriving (Eq,Show)
```

```
data BExpr = Bool Bool
           | Greater AExpr AExpr
           | Less AExpr AExpr
           deriving (Eq,Show)
```

Syntax.hs

つづき

```
data AExpr = Id String
           | Integer Integer
           | Add AExpr AExpr
           | Sub AExpr AExpr
           | Mul AExpr AExpr
           | Div AExpr AExpr
           | Pow AExpr AExpr
           | Fact AExpr
           | Negate AExpr
           deriving (Eq, Show)
```

Parser.hs

Parsec の関数を用いて, Lexer, Parser 部を実装

parseAExpr

四則演算のパーサと同じ雰囲気の実装
(実装自体は buildExpressionParser に丸投げ)

parseCommand

予約語を受理するパーサと式のパーサを組み合わせる
例: `parseIfCommand`

```
parseIfCommand :: Parser Command
parseIfCommand =
  do reserved "if"
     cond <- parseBExpr
     reserved "then"
     stmt1 <- parseCommand
     reserved "else"
     If cond stmt1 <$> parseCommand
```


Seq について

```
sequenceOfCommand =  
  do list <- sepBy1 parseCommand semi  
  return $ if length list == 1  
    then head list else Seq list
```

セミコロンをセパレータとして使用

(補足) セミコロンについて①

本実装におけるセミコロン:

Statement ; Statement ; ... ; Statement

- 一番最後のは付けない
- Pascal 等が同じ方式

(補足.) セミコロンについて②

C 言語のセミコロン:

式，文よりも細かいレベルでセミコロンの規則が定められている

(補足.) セミコロンについて③

例 (C の BNF):

```
<statement> ::= <labeled-statement>
                | <expression-statement>
                | <compound-statement>
                | <selection-statement>
                | <iteration-statement>
                | <jump-statement>
```

```
<expression-statement> ::= {<expression>}? ;
```

→ 本実装では、大まかな機能の実装に集中したいため、(とりあえず) セミコロンは単に文のセパレータとして用いる

Env.hs

変数 (環境) を Data.Map (連想リスト) で実装

```
type Env = Map String Integer
```

Eval.hs

Command(文) は Env を返す

```
evalCommand :: Env -> Command -> Env
```

AExpr, BExpr(式) は値を返す

```
evalBExpr :: Env -> BExpr -> Bool
```

```
evalAExpr :: Env -> AExpr -> Integer
```

デモ②

```
x := 1;
```

```
x := 1; y := 2; if(x > y) then{x := 10} else{y := 10}
```

```
x := 1; while(x < 10) do {x := x + 1}
```

IMP の構文上の問題点

IMP の型 (Integer と Boolean) は、構文上完全に切り離されている。 (AExpr と BExpr)

→ 関数の実装を考えると...

各型に対し別の構文で関数を実装することになる

まとめ

- IMP を実装した
- IMP の型表現の窮屈さを感じた

式を統合する

2つの Expr を統合する

```
data Expr =  Var String
           |  Integer Integer
           |  Bool Bool
           |  Negative Expr
           |  Add Expr Expr
           ...
           |  Sub Expr Expr
           |  Greater Expr Expr
           |  Less Expr Expr
           |  Equal Expr Expr
           deriving (Eq,Show)
```

型の表現

Expr の統合により, これまで AExpr.Integer に限定していた変数の型を Expr の各リテラル (Integer, Bool) に対応できるようにする必要がある.

Env.hs の変更点

変数部分を汎用に (型環境みたいな)

```
type Env = Map String TypeEnv
```

```
data TypeEnv = Integer Integer
              | TypeBool Bool
              deriving(Show)
```

Env の変更に伴う評価の変更

```
evalStatement :: Env -> Statement -> Env
```

```
evalExpr :: Env -> Expr -> TypeEnv
```

→(if 文の expr 等) 構文レベルでの型制限ができないため、パターンマッチで絞る必要がある

型エラーについて

例:if 文の評価

```
evalStatement env (If b x y) =  
  case evalExpr env b of  
    TypeBool True  -> evalStatement env x  
    TypeBool False -> evalStatement env y  
    _               -> error  
                      "If statement expected type 'Bool' as Expr"
```

デモ③

```
x := 1; y := true
```

```
x := 1; y := 2; if(x > y) then{x := 10} else{y := 10}
```

```
x := 1; while(x < 10) do {x := x + 1}
```

まとめ

式を統合し，拡張性を高めた

今後の展開

関数の実装

Func の構文的な位置づけ①

Expr に組み込むことを検討

```
data Expr = ...  
          | Func Expr Statement  
          | Apply Expr Expr
```

func (引数) 処理

Func の構文的な位置づけ②

処理が Statement だが、値を返したい

→Statement 内に、Expr との橋渡しとなるような構文が必要

```
data Statement = ...  
               | Return Expr
```

(Statement の構文も Expr に入れてもいい感じになる?)

関数型の実装

TypeEnv 内に関数型を実装したい

```
data Env = Map String TypeEnv
```

```
data TypeEnv = TypeInteger Integer  
              | TypeBool Bool  
              | Closure Env Expr Statement
```

Env の入れ子みたいに実装？

想定動作例

```
hoge := func(arg1){x := arg1 * 2; };  
> fromList [("hoge",Closure[("arg1",Assign(Var(x),  
    Mul(Var(arg1),Integer(2))))])] ]  
x := hoge(2);  
> fromList [("hoge",Closure[("arg1",Assign(Var(x),  
    Mul(Var(arg1),Integer(2))))], "x",TypeInteger(4))]
```

参考文献