拡張性のある PEG パーサの実装

木山真人[†1]

概要：PEG という形式文法がある．PEG には，曖昧さがない，字句解析器と統合している，という利点がある．また，パーサコンビネータという，小さなパーサを組み合わせて目的のパーサを作成する手法がある．この手法はホスト言語の機能をそのまま用いるという利点がある．著者は Haskell で動作する PEG のパーサコンビネータライブラリを作成し，それを用いてパーサを作成している．パーサの状態を State モナドで保持していると，パーサに新たな機能を組み合わせようとするとき，内部構造が複雑になるという問題が起きる．そこで，最小限のパーサを，State モナド，Error モナド，拡張用モナドの 3 つから成ると定義し，拡張する際は，拡張用モナドのみを入れ子構造で拡張する手法を提案・実装した．本提案手法によって内部構造の複雑化は解決できたが，入れ子構造が深くなることにより，機能の再利用性が低下するという別の問題が起きた．

キーワード：Parser，PEG，Haskell

1. はじめに

Parsing Expression Grammar(PEG)[1]という形式文法がある．PEG には，曖昧さがない，字句解析器と統合している，という利点がある．これらの利点のため，様々な文法の定義に使用されている．

また，パーサを作る方法として，パーサコンビネータがある[2]．これは，パーサコンビネータが提供する機能を使って，小さなパーサを組み合わせて目的のパーサを作成する方法である．この方法は，パーサをホスト言語で手書きするため，ホスト言語の機能をそのまま用いるという利点がある．

著者は Haskell で動作する Monadic な PEG パーサコンビネータを作成している．そのライブラリを作成して利用しているとき，パーサに新たな機能を組み合わせようとすると，内部構造が複雑になるという問題が起きた．パーサに拡張を容易に行える機能がライブラリにあれば，このような問題が避けられると考える．

そこで本稿では，Haskell で作成する Monadic な PEG パーサコンビネータに拡張性を追加する手法について提案し，実装を行う．2 章では PEG について簡潔に述べる．3 章では提案手法とその具体的な使用例について述べ，拡張性を確認する．4 章で，提案手法について考察し，5 章で本稿のまとめを述べる．

†1 熊本大学大学院自然科学研究科
Kumamoto Graduate School of Science and Technology, Kumamoto University

2. PEG

PEG は形式文法の 1 つであり，文脈自由文法の BNF 記法と似ている．しかし，選択の挙動が異なる．PEG の選択は，最初の解析が成功すればそれを使い，そうでない場合は，次を試すという意味になる．また選択を表すために，記号「|」ではなく，記号「/」を使う．

PEG では，各解析規則は

A ← e

という形式で表される．A は非終端記号，e は解析表現となる．終端記号，非終端記号，空文字列は解析表現であり，以下の表現も解析表現になる．

連結：e1 e2
選択：e1 / e2
AND-Predicate：&e
NOT-Predicate：!e
ゼロ回以上の繰り返し： e*
1 回以上の繰り返し：e+
省略可能：e?

ここで，e1，e2，e は任意の既存の解析表現である．また，各非終端記号には 1 つしか定義が存在しない．

ここでは特に連接，選択，AND-Predicate，NOT-Predicate について以下に説明する．

- 連接:e1 e2

入力が e1 にマッチした後, e2 にマッチするかどうかを調べる．

- 選択：e1 / e2
  入力が e1 にマッチするかどうかを調べ，**失敗したら**，e2 を調べる．
- AND-Predicate：&e
  入力が e にマッチするかどうかを調べる．ただし，入力は進めない．
- NOT-Predicate：!e
  入力が e にマッチするかどうかを調べ，**失敗したら**成功を返す．ただし，入力は進めない．

以上の説明から，ある文法定義を PEG で定義する場合の例を以下に示す．

S ← &(A!b)a+B!c
A ← a A? b
B ← b B? c

これは aabbcc のような a，b，c が同じ数だけ出現する文字列を認識できる文法である．

PEG パーサはそのまま再帰下降構文解析器で実装可能である．ただし，先読みが無制限に可能であるため，最悪の場合は指数関数時間になってしまう．その問題点を解決するため，解析の途中結果をメモ化し，線形時間で動作する Packrat Parser という手法がある[3]．

3. 提案手法

本章では, Haskell で PEG パーサを作成する際に問題となる点について説明する．次にその問題点を解決するための提案手法について述べ，その実装について述べる．その後，具体的な使用例を見ながら，拡張性について説明する．

3.1 問題点

著者は Haskell で PEG パーサを実装している．その際，新たな機能を追加しようとすると，パーサの内部構造がより複雑になってしまうという問題が起きた．

これは，内部の状態保存に State モナドを使っており, 新たな機能を追加するたびに, State モナドが複雑になっていくことが原因であると考えられる．

State モナドを使わずに，グローバル変数を使って状態を管理すればよいのだが，その方法では，安全性を破壊しかねない．また，コードの見通しも悪くなる．

そこで，PEG パーサ自体を最小限の状態から，徐々に拡張するようにし，拡張方法を工夫することで，新たな機能を追加しても，内部構造が複雑にならないようにできないかと考えた．

3.2 提案手法

提案手法では，最小限のパーサを定義し，その最小限のパーサを徐々に拡張することで，新しい機能の追加を可能にしている．新しい機能を追加する部分専用のモナドを用意し，ユーザはそのモナドに拡張部分を書くようにする．この手法を用いれば，内部構造が複雑にならず，拡張が容易になると考える．

まずは，提案手法の核となる，最小限のパーサについて述べる．

パーサの役割とは，文字列である入力を渡すと，成功した文字列と失敗した残りの文字列部分をペアにして返してくれると考える．例えば，

aabbccdd

というような入力を与えるとする．ここでパーサが aabbcc まで成功したと仮定すると，

(aabbcc，dd)

という結果が返ってくればよい．これは，Haskell であれば，State モナドとみなせる．

さらに，与えられた入力に対してパーサがまったく成功しなかった場合は，失敗を表す必要がある．そのため，成功したか失敗したかを表すためには，Error モナドが必要になると考える．

以上の 2 点を考えると，最小限のパーサとは, State モナドと Error モナドが合わさったモナドであると考えられる．

ただし，この 2 つのモナドでパーサを作成すると，拡張方法が複雑になると考える．なぜなら，新しい機能を追加するたびに，新しい機能で使用するデータを State モナドに追加するため，State モナドで管理するデータが複雑になるためである．

そこで, State モナドと Error モナドだけでなく，拡張用のモナドを準備し，新しい機能で使用するデータはその拡張用モナドを使うようにすればよい．こうすることで，元の State モナドに新たなデータが追加されずに，複雑さを回避できると考える．

State モナド，Error モナド，拡張用モナドを

まとめたモナドを最小限の PEG パーサとし，そのパーサを徐々に拡張していくことで，拡張性の問題を解決できると考える．

3.3 実装

```
01.type ParserT s e m a =
02.        StateT s (ErrorT e m) a
03.runParserT p s = runErrorT $ runStateT p s
```

図 1 最小限のパーサのコード

提案手法での考察から，Haskell で最小限のパーサを表現するコードを図 1 に示す．ここではモナドを結合するためにモナドトランスフォーマーを使っている．また，コード中の m が拡張用モナドとなり，ユーザはここで新たな機能の追加を行う. 2 行目までが最小限のパーサであり, 3 行目は作られたパーサを実行する関数である．

```
01.p1 <|> p2 = mplus p1 p2
02.seqP p1 p2 = do p <- p1
03.                q <- p2
04.                return (p ++ q)
05.notP p =
06.    StateT $ ¥s ->
07.      ErrorT $ do a <- runParserT p s
08.                  case a of
09.                    Left _ -> return $ Right ((), s)
10.                    Right _ -> return mzero
11.andP = notP . notP
```

図 2 PEG の基本関数

第 2 章で説明した連接, 選択, AND-Predicate, NOT-Predicate の実装について, そのコードを図 2 に示す．1 行目は選択の実装である．<|> は引数としてパーサを 2 つ取り，新たなパーサを返す．その実装は，MonadPlus で定義されている mplus を使うだけとなっている．これは, 基本のパーサが Error モナドで包まれているため，PEG の選択通りに動作する．2 から 4 行目が連接の実装になる. 2 つのパーサを順に実行し，その結果を結合して返すパーサを返すようになっている．5 から 10 行目が NOT-Predicate の実装の実装になっている．7 行目にあるように引数として与えられたパーサを一度実行し，その結果で返す値を変えるようなパーサを作っている. 9 行目は失敗した場合を書いてあり，入力は進めず，成功したとする. 10 行目は成功した場合を書いてあり，失敗したとして，MonadPlus で定義された mzero を返す. このような実装にすることで失敗が伝播するようにしている．11 行目は AND-Predicate の実装である．NOT-Predicate を 2 つつなげると，AND-Predicate と同じ関数になる．

ここまで説明してきた機能は，パーサの根幹となる部分であり，新たにパーサを作成しようとするユーザはこの部分に変更を加える必要はない．新たにパーサを作成するユーザは図 1 で示した最小限のパーサを使って，自分が作成したいパーサを作り，そのパーサに対応した機能を追加するだけで良い．

3.4 使用例

提案手法の実装を使って，最も簡単なパーサを作ってみる．文法は，第 2 章で説明した

S ← &(A!b)a+B!c
A ← a A? b
B ← b B? c

という文法を使う．また，今回作成するパーサは，最も簡単なパーサのため，拡張する部分はなく，入力が文字列でエラーも文字列を取るようにする．より効率よく文字列を扱いたい場合であれば, Byte String を使うように宣言すればよい．

```
01. type Parser a = ParserT String String Identity a
02. parse p s = runIdentity $ runParserT p s
```

図 3 最小限のパーサの例

図 3 にパーサの定義を示す. 1 行目がパーサの定義になる．入力もエラーも単に文字列を使うため，String となっている．また，今回は拡張を行わないため，拡張用モナドは Identity となる. 2 行目はパーサを実行するための関数であり，拡張は何もしていないため，最後に runIdentity を実行するだけとなる．

図 4 にパーサの基本部分のコードを示す．char は 1 文字がマッチするかどうか，string は文字列がマッチするかどうか, oneOf はその文字列のどれかに 1 文字がマッチするかどうかをチェックするパーサを返す関数である．satisfy は文字列の先頭の文字に対して与えた関数 f が True を返すかどうかをチェックし，True ならば，1 文字進めて成功，False ならば失敗を返すようになっている．ここで重要な関数は，layer であり，この関数がパーサを作成している．StateT，ErrorT と順番に包むこと

でパーサを作成している．ちなみに，option は引数として与えたパーサがマッチすれば，そのデータを，そうでなければ，引数として与えたデータを返すようなパーサを作成する関数であり，これはパーサの実装方法によらないため，AND-Predicate などと同様にユーザが変える必要はない．後で使うため，ここで定義してある．

```
01.layer f = StateT $ ¥s ->
02.                    ErrorT $ f s
03.satisfy f s =
04.    case s of
05.            c:cs | f c -> return $ Right (c, cs)
06.            otherwise -> return $ Left "not match"
07.char :: Char -> Parser Char
08.char c = layer $ satisfy (== c)
09.string :: String -> Parser String
10.string []        = return []
11.string (c:cs) = (:) <$> (char c) <*> (string cs)
12.oneOf :: String -> Parser Char
13.oneOf cs = layer $ satisfy (`elem` cs)
14. option a p = p <|> return a
```

図 4　パーサの基本部分

```
01.a = (string "a") `seqP` (option "" a) `seqP` (string "b")
02.b = (string "b") `seqP` (option "" b) `seqP` (string "c")
03.s1 = (andP (a <* (notP (string "b"))))
04.s2 = ((some (char 'a')) `seqP` b)
05.s3 = notP ((string "a") <|> (string "b") <|>
06.(string "c"))
07.s :: Parser String
08.s = s1 *> s2 <* s3
```

図 5　簡単なパーサ

目的の文法を認識するパーサを図 5 に示す．s が a，b，c が同じ数だけ出現する文字列を認識できるパーサになる．s3 は文法には現れていないが，入力文字列の最後を認識するパーサとなる．文法とコードがきれいに対応している．実際にパーサを実行するときは，

```
parse s "aabbcc"
```

のようにする．

パーサを作りたいユーザは，パーサのデータにあった関数だけを作成すればよい．連接や選択といった基本的な関数は新たに作成する必要はない．

```
digit = oneOf ['1'..'9']
digit0 = char '0' <|> digit
zero = string "0" <* notP digit0
nNum = (:) <$> digit <*> (many digit0)
dNum = (nNum <|> zero) `seqP` string "." `seqP` dPart
  where dPart = (:) <$> digit0 <*> dPart <|> (some digit) <* notP (char '0')
pNum = dNum <|> nNum <* notP (char '.')
num = (option "" (string "-") `seqP` pNum) <|> zero
number :: Parser Double
number = do n <- num
            return (read n)
```

図 6　数字を認識するパーサ

例えば，同じパーサを使って，数字を認識するパーサを作成しようとすると，図 6 のようになる．基本的な関数を組み合わせるだけで，より複雑なパーサを作成できる．

次に，拡張部分を加えたパーサ作成について説明する．拡張する機能は，今調べている文字列のポジションを記録する機能とどのようにパースしたかをすべて保存する機能である. 前回のパーサと違う部分は, Parser の型と，layer，satisfy である．その他の部分は変更がない．

```
type PosT m = StateT Int m
runPosT = runStateT

type AccT m = StateT String m
runAccT = runStateT

type Parser a = ParserT String String (PosT (AccT Identity)) a
parse p s pos acc = runIdentity $ runAccT (runPosT (runParserT p s) pos) acc

layer f = StateT $ ¥s ->
            ErrorT $
              StateT $ ¥pos ->
                StateT $ ¥acc -> f s pos acc

satisfy f s pos acc = case s of
                        c:cs | f c -> return ((Right (c, cs), pos + 1), acc ++ (show pos))
                        otherwise -> return ((Left "not match", pos), acc)
```

図 7　拡張用モナドを使ったパーサ

図 7 に拡張用モナドを使ったパーサの変更部分を示す．PosT，AccT という拡張のためのモナドトランスフォーマーを準備する．これを入れ子にして拡張用モナドとして使用することで，パーサが拡張できる．Parser 型をみると，拡張用モナドの部分が入れ子になっているのが分かる．また，それに対応して，parse 関数も変化してる．入れ子の順番に runPosT，runAccT が実行されており，最後に runIdentity が実行されている．これはさらなる拡張を考慮して最後をIdentityモナドにしている．もし，拡張をしないのであれば，Identity モナドは必要ない．layer は入れ子になったモナドの順番で作成されている．また，satisfy では拡張される順番で値を返すようになっている．

このように拡張する場合は，拡張のモナドトランスフォーマーを用意し，拡張用モナドとして利用すれば，容易に拡張できる．ユーザが準備すべきは，拡張のモナドトランスフォーマーを定義し，layer と satisfy をそれに応じて変更するだけである．

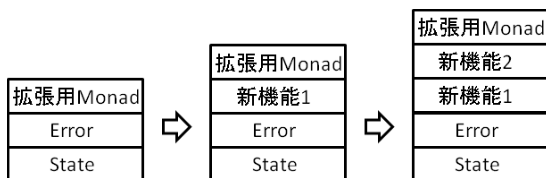

図 8　拡張の概念

図 8 にこの拡張方法における概念図を示す．下の 2 つの層は PEG パーサとして基本的な部分であり，ユーザはここを変更しない．基本的な部分が共通しているため，拡張しても連接や選択といった基本的な関数を変更する必要がない．新機能を追加する場合は，拡張用のモナドトランスフォーマーを準備すればよい．元の拡張用モナドは次の拡張するために，残しておく．もちろん，まったく拡張しないのであれば，拡張用モナドはなくしても問題はない．

実際に，図 8 で作られたパーサを，

parse s "aabbcc" 0 ""

と実行すると，

((Right ("aabbcc",""),10),"0123456789")

という値が返ってきて，パーサが成功したことを確認できる．また，

parse s "aaabbbcc" 0 ""

と実行すると，

((Left "not match",14),"012345678910111213")

という値が返ってきて，パーサの失敗を確認できる．

4. 考察

本章では，提案手法について考察を行う．

本提案手法は，モナドトランスフォーマーを入れ子に使用することで，他の拡張との依存性を排除し，パーサの拡張を容易にすることである．

通常の実装では，State モナドにパーサの状態をすべて実装していたため，新たな機能を追加しようとすると，内部構造が複雑になるという問題があった．

提案手法を用いれば，新たな機能を追加するときは，別のモナドを準備すればよく，内部構造が複雑になるということはない．しかし，拡張をすればするほど入れ子が深くなり，そのため，新しい機能の再利用性が低くなる．なぜなら，拡張する順番に対応して，他の関数が作られなければならないためである．そのため，機能の取り外しが容易に行えなくなってしまう．

そのため，提案手法は拡張性の問題をある程度解決したといえるが，機能の再利用に問題があるのではないかと考える．

再利用性を高めるためには，本提案手法のように入れ子構造で順序に依存するのではなく，順序に依存しない拡張方法を考えなければならない．Swierstra による Data types á la carte[4]では，データ型とそのデータを扱う関数を分離して組み合わせる手法について述べており，また，同じ手法で Free Monad を組み合わせられることを示している．この手法を応用できれば，入れ子構造をなくせるのではないかと考える．また，Oleg らによる Extensible Effects[5]では，継続モナドを基にして新たなモナドトランスフォーマーを作っており，この手法が本提案手法に応用できるかもしれない．

5. まとめ

本稿では，Monadic な PEG パーサコンビネータの拡張が容易になるような提案手法について述べた．

著者が Haskell で動作する PEG のパーサコンビネータライブラリを作成・利用しているとき，パーサに新たな機能を組み合わせようとすると，内部構造が複雑になるという問題が起きた．これは，内部の状態保存に State モナドを使っており，新たな機能を追加するたびに，State モナドが複雑になっていくことが原因であった．

そこで，最小限のパーサを，State モナド，Error モナド，拡張用モナドの 3 つから成ると定義し，拡張する際は，拡張用モナドのみを入れ子構造で拡張するようにした．これにより，内部構造が複雑になるという問題点は解決した．

内部構造の複雑化は解決できたが，入れ子構造が深くなることにより，機能の再利用性が低下するという別の問題が起きた．これは今後の課題となる．

質疑・応答

Q ルールごとにアクション指定しなくて大丈夫なのか？

A 拡張機能で後から対応すればよい．メモ化したデータを使えばできると考える．

Q 遅延評価をうまく使ってメモ化できないの？

A グローバル変数を無理やり使うとできるかも．ただ綺麗にやるのは難しそう．

Q メモ化しないと遅い？

A 圧倒的に遅い

Q 拡張って具体的にどういう意味なのかもっと詳しく教えて

A パーサの基本動作（文字列を与えると，成功か失敗を返す．成功した場合は，マッチした部分までと，それ以外を返す.）以外はすべて拡張としている．メモ化や読み込んだ文字列の位置もすべて拡張となる．

参考文献

1) Bryan Ford: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation, Symposium on Principles of Programming Languages, POPL'04, pp.111-122, ACM(2004)

2) Graham Hutton, Erik Meijer: Monadic Parser Combinators, NOTTCS-TR-96-4(1996)

3) Bryan Ford: Packrat Parsing: Simple, Powerful, Lazy, Linear Time, Functional Pearl, Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, ICFP '02,pp.36-47,2002

4) Wouter Swierstra: Data types á la carte,Journal of Functional Programming, Vol.18, pp.423-436, 2008

5) Oleg Kiselyov, et al：Extensible effects: an alternative to monad transformers,Haskell '13 Proceedings of the 2013 ACM SIGPLAN symposium on Haskell, pp.59-70, 2013