PEGのパラメタ付き拡張Macro PEGの提案

○水島 宏太[1,a)]

概要：PEG は，2004 年に Ford によって発表された解析的形式的文法の一つである．PEG は非常にシンプルであるにも関わらず，広い範囲の文法を解析することができる（一部の文脈依存言語を含む）．また，Packrat Parsing という構文解析アルゴリズムによって，PEG で表現可能な任意の言語は線形時間で解析できるという好ましい性質を持っている．PEG の解析木は必ず一意に定まるため，プログラミング言語の構文解析器など，非自然言語の構文解析に向いている．
PEG はその強力さにも関わらず，いくつかの問題がある．まず，PEG の規則は再利用性性が低い．また，現実のプログラミング言語の文法要素を扱うためには必ずしも十分な能力を持っていない. たとえば，複数の修飾子が順不同で出現するというパターンを認識するためには，通常の PEG では全てのパターンを書き下す必要があり，修飾子の数が増えると文法の規模が爆発的に増加してしまう．本発表では，Macro Grammar を参考に，PEG がパラメタを取れるように拡張した Macro PEG を提案する．Macro PEG では，先ほど挙げたような文法を簡潔に記述することができる．また，本発表では，Macro PEG とパーザコンビネータとの関係について考察する．パーザコンビネータとは，主に関数型プログラミング言語で用いられる手法であり，関数合成によってパーザを組み立てる．最後に，本発表では，Macro PEG の表現能力について考察を行う．

Macro PEG: PEG with macro-like rules

Kota Mizushima[1,a)]

Abstract: PEGs are analytic formal grammars invented by Ford in 2004. Despite of its simplicity, PEGs can recognize wide-ranged grammars, including some context sensitive languages. Also, packrat parsing enables linear time parsing for any language expressed by PEGs. Since PEGs don't have ambiguity, they are well suited for non-natural language parsing, especially programming language's parsing.
Although PEGs are powerful, they have some problems. At first, rules of PEG have low reusability. And they don't have enable power to handle all pratical programming langauges. Considering so-called "modifiers" in several programming languages: there is a sequence of modifiers and each modifier occurs only once in the sequence. To recognize such modifier sequences, the size of the grammar may increase explosively. In this presentation, I propose Macro PEG, inspired by Macro grammar. Macro PEGs can express 'permutation languages' concisely comparison with traditional PEGs. Also, I consider the relation between Macro PEGs and parser combinators. Parser combinators are a technique that build parsers by composing smaller parsers. At the last, I consider about Macro PEGs' expressiveness.

1. はじめに

Ruby[1] や Python[2] といったスクリプト言語，あるいはそれらに文法的な影響を受けたと思われる Scala[3] や Kotlin などの言語が Web アプリケーションの開発 (Ruby, Python, Scala) や Android アプリケーション開発 (Kotlin) などの分野をはじめとした分野で使われるようになってきている．これらの言語の特徴として，改行に意味を持たせたり，字句要素が構文要素を含むなど総じて文脈依存性が高いということができる．このような文脈依存性が高い文

[1] 株式会社ドワンゴ
DWANGO Co., Ltd.
[a)] kota_mizushima@dwango.co.jp

法に対して，yacc[4](bison) などの一般的に使用されているパーザ生成系が生成する LALR(1) 構文解析アルゴリズムを用いたパーザは文脈自由文法を想定しているため，解析するのが困難である．

各プログラミング言語の処理系は ad hoc な手法を取ることでこの問題に対応しているが，そのために文法の記述が煩雑になり，文法定義のメンテナンスが困難になるという問題がある．例えば，プログラミング言語 Ruby の処理系はパーザ生成系として yacc を使用しているが，その文法定義は手書きの字句解析器を含めて 8000 行以上におよぶ．また，Scala，Go，Kotlin は手書きの構文解析器を使っている．

```
puts "1 + 2 = {1 + 2}" # 3 が表示される
```

図 1 式が埋め込まれた文字列リテラル（Ruby）
Fig. 1 String literal in which an expression is embedded (Ruby)

```
println(s"1 + 2 = ${1 + 2}") // 3 が表示される
```

図 2 式が埋め込まれた文字列リテラル（Scala）
Fig. 2 String literal in which an expression is embedded (Scala)

```
println("1 + 2 = ${1 + 2}") // 3 が表示される
```

図 3 式が埋め込まれた文字列リテラル（Kotlin）
Fig. 3 String literal in which an expression is embedded (Kotlin)

Ruby（**図 1**)、Scala（**図 2**)、Kotlin（**図 3**）は式を埋め込み可能な文字列リテラルを持っているが，文字列リテラルは一般的に字句要素 (トークン) であり，式は構文要素である．つまり，字句要素の中に構文要素が埋め込まれることになるが，このような機能は字句解析をベースとした構文解析アルゴリズムでは，字句解析器に状態を持たせて，パーザから字句解析器の状態を変更するなどの，ad hoc な手法を使わなければ解析を行うことができない．

この問題点を解決可能な構文解析アルゴリズムの一つとして PEG という形式文法とその実装である packrat parsing という構文解析アルゴリズムが存在する．Packrat parsing が取り扱える文法である PEG は LALR(1) より強力である上に，字句解析器を必要としないアルゴリズムであるため，文脈依存性が高い文法を持った言語の構文解析器を実装するのに向いている．

たとえば，式を埋め込み可能な文字列リテラルを持った言語は PEG で**図 4** のように表現することができる．単純化するため，リテラルとして整数と文字列のみを持ち，演算として加算および減算のみを許すようにしている．字句解析と構文解析が分かれていることを前提としたアルゴリズムでは，先に述べたように，空白の解釈が文脈によって異なったり，文字列リテラルが要素として式を持つような言語を表現することは難しい．

```
Spacing <- " " / "\t" / "\r" / "\n"
Expression <- Primary (
  "+" Spacing Primary
  "-" Spacing Primary
)*
Primary <- "(" Spacing Expression ")" Spacing /
               NumberLiteral Spacing /
               StringLiteral Spacing
NumberLiteral <- "0" | [1-9][0-9]*
StringLiteral <- "\""
  ("#{" Expression "}" / !"\"" .)*
"\""
```

図 4 式が埋め込まれた文字列リテラルを表現する PEG
Fig. 4 A PEG that expresses String literal in which an expression is embedded

PEG は字句解析を別途必要としないため，このような言語を表現するのに向いている．

一方で，この例に端的に表れているように，PEG の規則だけでは，いわゆるトークンに相当する文字列が出現するたびに，Spacing という規則が出てくる．この場合，トークンと Spacing をセットで再利用したいのだが，PEG だけではそれは困難である．

また，PEG で取り扱うことが容易ではない，あるいは不可能であるような文法が存在する．たとえば，回文を表現する言語は PEG で取り扱うことが困難あるいは不可能である． 他には順列を表す言語を表現することも容易でない．順列を表す言語の例としては，XML の属性やプログラミング言語のアクセス修飾子などがある．このような問題点を解決するために，PEG を拡張して規則が引数を取れるようにした Macro PEG を実験的に提案する．Macro PEG では，文法の再利用性の問題を改善できる．また，Macro PEG では回文や順列を表す言語を表現することも容易である．

本発表の残りの構成は次のようになっている．まず，2 節では，本発表の前提となる parsing expression grammar(PEG) について説明する．非形式的なセマンティクスおよび EBNF と異なる点について説明する．3 節では，PEG の表現力を豊かにするための拡張として本発表で提案する Macro PEG の概要を説明する．いくつかの例を通して，Macro PEG が既存の PEG の表現力の問題点をどのように解決するかを見ていく．4 節では，Macro PEG とパーザコンビネータの対応関係を示す．5 節では，Macro PEG においても表現が困難であるような文法にはどのよ

うなものがあるかを述べる．6 節では関連研究について述べる．最後に，7 節においてまとめをおこなう．

2. Parsing Expression Grammar

Macro PEG は parsing expresssion grammar (PEG) と呼ばれる形式文法をベースにしているため，PEG についてまず説明を行う．PEG は $G = (V_N, V_T, R, e_s)$ の四つ組からなる．それぞれ次の意味を持つ：

- V_N: 非終端記号の集合
- V_T: 終端記号の集合
- R: 規則の集合．$r \in R$ において，r は $A \leftarrow e$ となる．また，$A \in V_N$ であり，e は parsing expression と呼ばれる式である．
- e_s: 開始式．ある PEG と文字列のマッチングを行う際には，e_s の評価からマッチングが始まる．

parsing expression は**図 5** のような要素から構成されている．parsing expression は EBNF[5] に類似した記法を用いて文法を表現していることがわかる．

"" or ε	:	空文字列
" "	:	文字列リテラル
[ ]	:	文字クラス
N	:	非終端記号
$e_1\ e_2$	:	連接
e_1 / e_2	:	優先度付き選択
$e*$	:	0 回以上の繰り返し
$e+$	:	1 回以上の繰り返し
$e?$	:	0 回または 1 回の出現
$\&e$	:	And-predicate
$!e$	:	Not-predicate

図 5 Parsing expression (1)
Fig. 5 Parsing expressions (1)

ここで，BNF の | と異なり，/ には順序に意味があるという点に注意する必要がある．つまり，PEG において e_1 / e_2 と e_2 / e_1 は等価ではない．& と ! はともに先読みを行う演算子であり，e にマッチするかを試して，前者はマッチが成功する場合に，後者はマッチが失敗する場合にマッチする演算子であるが，ともに入力を消費しない．

PEG は決定的文脈自由言語に加えて，いくつかの文脈依存言語を扱うことができる．次の図 6 は，文脈自由言語ではないが PEG で取り扱うことができる言語の例である．

```
S <- &(A !"b") "a"+ B !.
A <- "a" A? "b"
B <- "b" B? "c"
```

図 6 $a^n b^n c^n$ を表現する PEG
Fig. 6 PEG describing $a^n b^n c^n$

PEG は packrat parsing という手法を用いて線形時間で解析できることが知られている [6].

3. Macro PEG

Macro PEG は，PEG の各規則が任意個の引数を取れるようにしたものである．これは，文脈自由文法の規則を引数を取れるように拡張した Macro Grammar[7] に影響を受けたものである．

Macro PEG は PEG と同様に $G = (V_N, V_T, R, e_s)$ の四つ組からなる．それぞれ次の意味を持つ：

- V_N: 非終端記号の集合
- V_T: 終端記号の集合
- R: 規則の集合．$r \in R$ において，r は $A(P_1, ..., P_n) \leftarrow e$ となる（$n = 0$ である場合，PEG の規則と同じ意味）．また，$A \in V_N$ であり，e は parsing expression とである．$P_1, ..., P_n$ は規則の仮引数であり，何らかの一意な識別子が割り当てられている．
- e_s: 開始式．

引数付き規則は $N(e_1, ..., e_n)$ のようにして呼び出すことができる．**図 7** は図 5 を拡張したものである．Parsing expression としては引数を伴った規則の呼び出し以外は拡張されていないことがわかる．

"" or ε	:	空文字列
" "	:	文字列リテラル
[ ]	:	文字クラス
N	:	非終端記号
$N(e_1, ..., e_n)$	:	引数を伴った規則の呼び出し (e_n は parsing expression)
$e_1\ e_2$	:	連接
e_1 / e_2	:	優先度付き選択
$e*$	:	0 回以上の繰り返し
$e+$	:	1 回以上の繰り返し
$e?$	:	0 回または 1 回の出現
$\&e$	:	And-predicate
$!e$	:	Not-predicate

図 7 Parsing expression (2)
Fig. 7 Parsing expressions (2)

規則の呼び出し $N(e_1, ..., e_n)$ は名前呼び出しで評価されるものとする．すなわち，呼び出し時点では引数は評価されず，呼び出し先で引数の参照があったときに初めて引数が評価される．引数を値呼び出しで評価する方法も考えられるが，今回はその方法を取らなかった．

Macro PEG は PEG のように $G = (V_N, V_T, R, e_s)$ の四つ組で表現することができる．ただし，$r \in R$ における r は $N(P_1, ..., P_n) \leftarrow e$ という形を取る．また，parsing expression は図 7 の形を取る．

3.1 Macro PEG の性質

ある Macro PEG G が受理する言語を $L(G)$ とおく．すると，$L(G)$ は次の演算に対して閉じていると考えられる．つまり，和集合，積集合，補集合に関して閉じている．

- $L(G_1) \cup L(G_2)$
- $L(G_1) \cap L(G_2)$
- $L(\overline{G})$

このことに関する証明は，文献 [8] と同じになるので省略する．また，文献 [8] における PEG に関する性質の多くは Macro PEG でもそのまま成立する．たとえば，任意の G に対して，$L(G) = \emptyset$ かどうかの判定は，PEG がそうであるように，自明に決定不能である．

3.2 式を含む文字列リテラルを表現する PEG の改善

Macro PEG では，parsing expression を規則に適用することで新しい parsing expression を生成することができる．この特性を利用して，図 4 の PEG を改善したものが**図 8** である．

```
Spacing <- " " / "\t" / "\r" / "\n"
Token(terminal) <- terminal Spacing

Expression <- Primary (
  Token("+") Primary
  Token("-") Primary
)*
Primary <-
  Token("(") Expression Token(")")  /
  Token(NumberLiteral) /
  Token(StringLiteral)
NumberLiteral <- "0" | [1-9][0-9]*
StringLiteral <- "\""
  ("#{" Expression "}" / !"\"" .)*
"\""
```

図 8 式を埋め込み可能な文字列リテラルを表現する Macro PEG
Fig. 8 A Macro PEG that expresses String literal in which an expression is embedded

ここで，引数を取る規則 `Token(terminal)` を定義し，`terminal` に続いて `Spacing` が続くような式をまとめることができている．また，`Token` という名前をつけることによって可読性も向上している．

別の例として，ある文字列をセパレータとして構文要素が繰り返しあらわれるような parsing expression を表現したいとする．Macro PEG ではこのような parsing expression を**図 9** のようにして表現することができる．引数付き規則 `repsep` がある `separator` で区切られた `element` の 1 回以上の繰り返しを表現している．

このように，Macro PEG を用いるとより複雑な parsing

```
repsep(separator, element) <-
  element (separator element)*
Data <- repsep(",", StringLiteral)
StringLiteral <- "\"" (!"\"" .)* "\""
```

図 9 カンマ区切りの文字列リテラルの列を表す Macro PEG
Fig. 9 A Macro PEG that expresses comma separated String literals

expression を再利用の単位とすることができるようになる．

3.3 回文を表現する Macro PEG

```
S <- X !.
X <- "a" X "a" / "b" X "b"  / |""
```

図 10 長さが偶数であるような回文を表現することを意図した PEG
Fig. 10 PEG intended to describe palindromes

CFG で表現可能だが，PEG で容易に表現ができない言語のひとつに回文がある．直観的には**図 10** の PEG で回文が表現可能なように思えるが，これは PEG の優先度付き選択のためにうまく意図通りに動作しない．具体的には，この PEG は文字列"abbaabba"を受理しない（これは長さが偶数の回文であるため意図通りに動けば受理されるはずである）．回文を表現可能な PEG が存在する可能性はあるが，現在のところ回文を表現可能な PEG は発見されていない．

Macro PEG では**図 11** のようにして長さが偶数である回文を表現することができる（開始式は S とする）．

```
S    <- P("") !.
P(r) <- "a" P("a" r) / "b" P("b" r) / r
```

図 11 長さが偶数であるような回文を表現する Macro PEG
Fig. 11 Macro PEG describing palindromes

この Macro PEG は次のようにして動作する．まず，規則 P は引数 r として，これまでにマッチした文字列の逆順にマッチする Macro PEG を保持すると仮定する．S の中では規則 $P(r)$ を空文字列を引数として呼び出す（まだ文字列とのマッチングを行っていないため）．$P(r)$ の中では，a，b，c のいずれかの文字列に対してマッチングを行い，マッチした場合その文字列を r の前にくっつけて再帰的に P を呼び出す．いずれの文字列に対してもマッチしなかった場合，残りの文字列はこれまでマッチした文字列を逆順にしたものになるので，引数 r とのマッチングを行う．このようにして，Macro PEG では長さが偶数であるような回文を表現することができる．

先に述べたように，回文は CFG では容易に表現できるが，PEG で表現するのは困難な（あるいは表現できない）言語であり，Macro PEG の言語クラスである Macro PEL は Context Free Language(CFL) を真に含む可能性がある．また，仮に Macro PEL が CFL を真に含むのならば，任意の CFG を解析するアルゴリズムで線形時間で解析可能なものは現在のところ存在しないことから，任意の Macro PEG で表現される言語を線形時間で解析する事はできないと考えられる．なお，文献 [8] では，PEG が線形時間で解析できることをもって，CFG で表現できるが PEG で表現できない言語が存在するであろうということを考察している．

3.4 順列言語を表現する Macro PEG

アルファベットに対して，それを構成する要素が 0 回または 1 回出現する順列のような言語を Macro PEG で表現することができる．これは，通常の PEG でも表現することができるが，全てのパターンを列挙する必要があるため PEG で表現するのはあまり現実的でない．順列言語は，プログラミング言語の修飾子，XML[9] の属性などの解析において応用することができる．

Macro PEG によって，文字 a,b,c からなる順列言語を**図 12** のようにして表現することができる：

```
M(A) <- !(A)
       ( "a" M(A / "a")
      / "b" M(A / "b")
      / "c" M(A / "c")
      / ""
      )
```

図 12 順列言語を表現する Macro PEG
Fig. 12 Macro PEG describing permutations

3.5 ネストしたクラスを表現する Macro PEG

Java では，クラスをネストすることができるが，その際に，static なクラスの内部には非 static なクラスを定義することができないという制約がある．static なクラスは非 static なクラスと比較すると，外側のクラスの this を参照できないという違いがある．

```
TopLevel <- ClassDeclaration(!"") *
ClassDeclaration(S) <-
  (  !S ClassBody(!"")
  / "static" ClassBody("")
  )
ClassBody(S) <-
  ( MemberDeclaration(S)
  / ClassDeclaration(S)
  )*
```

図 13 ネストしたクラスを表現する Macro PEG
Fig. 13 Macro PEG describing nested classes

このような制約は，Macro PEG を用いて**図 13** のように書くことができる．なお，

- TopLevel: トップレベルの宣言
- ClassDeclaration: クラス定義

- ClassBody: 修飾子を除いたクラス定義
- MemberDeclaration: クラス以外のメンバー（フィールド，メソッド）定義

であるものとする．また，引数 S は現在のコンテキストが static なクラスかどうかを表すものとする．非 static なコンテキストでのみ非 static なネストしたクラスを書くことができるため，$!S$ をフラグとして，非 static なクラス定義の前に付加している．

このように，Macro PEG を用いることで，PEG では表現できない，あるいは困難な言語を表現することができる．

4. Macro PEG とパーザコンビネータの関係

Macro PEG はパーザコンビネータを使って簡潔に表現することができる．パーザコンビネータとは，個々のパーザを値（オブジェクトあるいは関数）とみなして，パーザ同士を合成することで複雑な言語のパーザを構築するための技法である．**図 14** はプログラミング言語 Scala[3] を用いた Macro PEG のパーザコンビネータによる実現である．個々のパーザの実装については本題からはずれるので，[10] を参照されたい．

このコードを用いて，回文を表現する Macro PEG は **図 15** のようにして表現することができる．

`def` は関数の定義，演算子`/`は優先度付き選択，`~`は連接，`"a".s` は文字列リテラルを表現している．規則 P に相当する関数 P が引数として `Parser[Any]` をとっており，これによって引数を取る規則が実現されている．

このパーザコンビネータを実装するにあたって特に工夫はしていないが，単純に `Parser[T]` 型の値を引数として渡せるようにするだけで名前呼び出しと同じセマンティクスが実現されている．このことから，引数として任意個の `Parser[T]` 型の値をとれるようにした関数を，Macro PEG における引数付き規則と同様にみなすことができるといえる．

```
object Parsers {
  type Input = String
  case class ~[+A, +B](_1: A, _2: B)
  abstract sealed class ParseResult[+T] {
    def drop: ParseResult[Any] = this match {
      case ParseSuccess(_, next) =>
        ParseSuccess(None, next)
      case ParseFailure(_, next) =>
        ParseFailure("", next)
    }
  }
  final case class ParseSuccess[+T](
    result: T, next: Input
  ) extends ParseResult[T]
  final case class ParseFailure(
    val message: String, next: Input
  ) extends ParseResult[Nothing]

  abstract sealed class Parser[+T] {self =>
    def apply(input: Input): ParseResult[T]
    def /[U >: T](b: Parser[U]): Parser[U] =
      this | b
    def ~[U](b: Parser[U]): Parser[T ~ U] =
      Sequence(this, b)
    def ? : Parser[Option[T]] = Option(this)
    def * : Parser[List[T]] = Repeat(this)
    def + : Parser[List[T]] =
      (this ~ Repeat(this)).map{
        case a ~ b => a :: b
      }
    def unary_! : Parser[Any] = Not(this)
    def and: Parser[Any] = !(!(this))
    def map[U](function: T => U): Parser[U] =
      Mapping(this, function)
  }
  def any: AnyParser.type = AnyParser
  def string(literal: String): StringParser =
    StringParser(literal)
  implicit class RichString(
    val self: String
  ) extends AnyVal {
    def s: StringParser = StringParser(self)
  }
  def rule[T](parser: => Parser[T]):
    RuleParser[T] =
      RuleParser(() => parser)
}
```

図 14 パーザコンビネータによる Macro PEG の実装
Fig. 14 A Macro PEG implementation by parser combinator

5. Macro PEG で表現する事が困難な文法

Macro PEG によって PEG の規則の再利用性を向上させ，表現力の問題を改善できるが，Macro PEG でも表現することが容易でない文法も存在する．

たとえば、変数宣言が前方にない場合、その変数への参照を構文解析の失敗とみなすような文法を考える．このような文法は C 言語の typedef 宣言にみられるものであるが，Macro PEG でも簡単には表現できない．また，Python や Haskell に代表される，インデントの有無によって解析結果が変わる文法を表現するのも苦手である．

6. 関連研究

PEG に対して，現実的なプログラミング言語を扱うための拡張を行った例としては，文献 [11] がある．文献 [11]

```
object Palindrome {
  def S: Parser[Any] = rule {
    P("".s) ~ !any
  }
  def P(r: Parser[Any]): Parser[Any] = rule {
    "a".s ~ (P("a".s ~ r)) /
    "b".s ~ (P("b".s ~ r)) /
    r
  }
}
```

図 15 回文を表現するパーザコンビネータ
Fig. 15 A parser combinator describing palindromes

では，主に

- C の typedef のような文脈依存の構文
- Python や Haskell のようなインデントベースの構文
- C#における文脈依存キーワード

を対象として，それらを扱うための PEG に対する拡張を提案している．文献 [11] では主に，実際のプログラミング言語を扱うために必要な機能という観点でいくつかの拡張が定義されているが，本発表では，最初に PEG の規則に引数を持たせるという拡張を考え，それからどのような応用があるかを考察している点に違いがある．文献 [11] で提案されている拡張と本発表で提案した拡張で，受理する言語にどのような違いがあるかは不明である．また，Macro PEG は，和集合，積集合，補集合に対して閉じているが文献 [11] の拡張をほどこした PEG に対してそのような性質が成り立つかは不明である．この研究では，Macro PEG では取り扱いが難しい文法のいくつかについて容易に取り扱えるようになっているため，Macro PEG の改善として，これらの拡張を追加することが考えられる．

また，Packrat Parser のメモ化機構を利用して PEG に対して左再帰のサポートを追加した研究に文献 [12] がある．文献 [12] においては，PEG への左再帰のサポートによってどのようなセマンティクスの変更があるかについて言及がない．また，文献 [13] によって，サポートされない左再帰のパターンがあることが示されている．そのため，文献 [12] によって左再帰のうちどの程度のパターンがサポートされるかは不明瞭である．

7. まとめ

本研究では，PEG をベースに，規則が任意個のパラメタを取れるように拡張した Macro PEG を提案した．Macro PEG は，引数を名前呼び出しで評価する．その結果として，

- 規則の再利用性が向上した
- PEG では表現が困難/不可能であった文法について定義できるようになった

ことを示した．

また，Macro PEG は，ちょうど対応するパーザコンビネータを定義することができる．これは一般のパーザコンビネータに対して制約を付けたものと考えることができる．

Macro PEG の計算量については不明な点が多いが，Macro PEG では任意の文脈自由言語を表現できる可能性が高いため，計算量についても，良くても多項式時間が必要であり，線形時間での解析を行うことは困難だと考えられる．

今後の研究の方向性としては，

- Macro PEG の最悪計算量の解明
- Macro PEG のサブセットとして，明らかに線形時間で解析できるものを考える
- Macro PEG のサブセットとして，PEG への変換が定義可能なものを考える

といったことが挙げられる．

謝辞 Macro PEG で回文を表現する方法についての議論に付き合ってくださった@chiguri 氏，Macro PEG の様々な性質についての議論に付き合ってくださった古賀智裕氏に感謝します．

参考文献

[1] Flanagan, D. and Matsumoto, Y.: *The Ruby Programming Language*, O'Reilly, first edition (2008).
[2] Rossum, G.: Python Reference Manual, Technical report, Amsterdam, The Netherlands, The Netherlands (1995).
[3] Odersky, M., Spoon, L. and Venners, B.: *Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition*, Artima Incorporation, USA, 2nd edition (2011).
[4] Johnson, S.: Yacc: Yet Another Compiler Compiler, *UNIX Programmer's Manual*, Holt, Rinehart, and Winston, New York, NY, USA, pp. 353–387 (1979).
[5] Organization, I. S.: *Syntactic metalanguage – Extended BNF* (1996). ISO/IEC 14977.
[6] Ford, B.: Packrat Parsing: Simple, Powerful, Lazy, Linear Time, *Proceedings of the 2002 International Conference on Functional Programming* (2002).
[7] Fischer, M. J.: Grammars with macro-like productions, *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, Vol. 0, pp. 131–142 (online), DOI: http://doi.ieeecomputersociety.org/10.1109/SWAT.1968.12 (1968).
[8] Ford, B.: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation, *Symposium on Principles of Programming Languages* (2004).
[9] Bray, T., Paoli, J. and Sperberg-McQueen, C.: *Extensible Markup Language (XML) 1.0 (Fourth Edition), W3C Recommendation* (2006).
[10] : Macro PEG Combinators. https://github.com/kmizu/macro_peg_combinators.
[11] Matsumura, T. and Kuramitsu, K.: A Declarative Extension of Parsing Expression Grammars for Recognizing Most Programming Languages, *Journal of Information Processing*, Vol. 24, No. 2, pp. 256–264 (online), DOI: 10.2197/ipsjjip.24.256 (2016).
[12] Warth, A., Douglass, J. R. and Millstein, T.: Packrat Parsers Can Support Left Recursion, *Proceedings*

of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM '08, New York, NY, USA, ACM, pp. 103–110 (online), DOI: 10.1145/1328408.1328424 (2008).

[13] 佳章白田, 真人木山, 評, 芦原.: 同一入力位置で複数発生する左再帰へ対応した Packrat Parsing の設計と実装. 情報処理学会論文誌プログラミング（PRO）. Vol. 4, No. 2, pp. 104–115 (2011).

[14] Donovan, A. A. and Kernighan, B. W.: *The Go Programming Language*, Addison-Wesley Professional, 1st edition (2015).

[15] Grimm, R.: Better Extensibility through Modular Syntax, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pp. 19–28 (2006).

[16] Redziejowski, R.: Some aspects of Parsing Expression Grammar, *Fundamenta Informaticae 85, 1-4*, pp. 441–454 (2008).

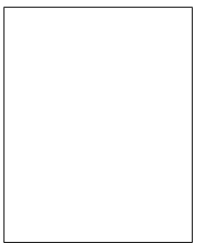
[17] Mizushima, K., Maeda, A. and Yamaguchi, Y.: Packrat parsers can handle practical grammars in mostly constant space, *PASTE '10: Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, New York, NY, USA, ACM, pp. 29–36 (online), DOI: http://doi.acm.org/10.1145/1806672.1806679 (2010).

[18] Parr, T. J. and Quong, R. W.: ANTLR: A Predicated-LL(k) Parser Generator, *Software Practice and Experience*, Vol. 25, pp. 789–810 (1994).

[19] BAARS, A. I., LOH, A. and SWIERSTRA, S. D.: FUNCTIONAL PEARL Parsing permutation phrases, *Journal of Functional Programming*, Vol. 14, pp. 635–646 (online), DOI: 10.1017/S0956796804005143 (2004).

[20] Redziejowski, R.: Parsing Expression Grammar as a primitive recursive-descent parser with backtracking, *Concurrency Specification and Programming Workshop* (2006).

[21] Hutton, G. and Meijer, E.: Monadic Parser Combinators, Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham (1996).



水島 宏太

1983 年生．2005 年筑波大学第三学群情報学類卒業．2011 年筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻博士後期課程修了．博士（工学）．2014 年 株式会社ドワンゴに入社．2016 年 8 月現在，現職．プログラミング言語，構文解析，プログラミング教育などに興味を持つ．情報処理学会，ACM 各会員