情報処理学会論文誌　プログラミング　Vol. 4　No. 2　77–91 (Mar. 2011)

最適化 Packrat Parser の空間計算量の計算手法の提案

水島 宏太[†1]　前田 敦司[†1]　山口 喜教[†1]

従来主流の構文解析手法である LL や LR 法の欠点を解決する構文手法の 1 つとして packrat parsing という構文解析手法が注目を集めている．packrat parsing は，parsing expression grammar（PEG）という形式文法で表現できる任意の言語を扱うことができ，これは LL 法や LR 法よりも強力である．また，packrat parsing では解析を線形時間で行うことができる．しかし，packrat parsing では，構文解析の中間結果をすべてメモ化するため，メモリ使用量が線形になってしまうという欠点がある．我々の研究では，カット演算子を PEG に導入し，プログラマがカットを適切に挿入することで，実用上，メモ化ために必要なメモリ使用量がほぼ一定である packrat parser を生成できる packrat parser 生成系を提案した．しかし，文法定義に対してカットが必要十分なだけ挿入できているかを確認するのは簡単ではなく，文法定義を入念にチェックしたうえで，実際に構文解析を行ってメモリ使用量を測定する必要があった．本論文では，カットによって拡張された PEG による文法定義を静的に解析することで，その文法定義から生成される packrat parser が必要とするメモリ使用量を推計する手法を提案する．本論文の提案手法を適用することで，実際に構文解析を行うことなしに，カットが必要十分なだけ挿入されているかどうかのチェックを機械的に行うことができる．提案手法は，先行研究で提案した packrat parser 生成系を実用で使ううえで重要な問題点を解決するものである．また，提案手法は，PEG から生成された packrat parser の空間計算量に対するカットのインパクトを測る 1 つの尺度を与えているといえる．

A Space Complexity Calculation Method of Optimized Packrat Parsers

Kota Mizushima,[†1] Atusi Maeda[†1]
and Yoshinori Yamaguchi[†1]

Packrat parsing is drawing attention of researchers as an alternative to mainstream parsing algorithms such as LL- or LR-family. Packrat parsing can handle any languages that parsing expression grammars (PEGs) can express and the set of languages is strictly larger than the one that LL- or LR-family can express. Also, packrat parsers parse any input in linear time. But packrat parsers require linear space because of memoization of all intermediate parsing results. In our previous study, we proposed to extend PEGs with cut operators to remove unneeded memoized space. If grammar writers properly insert cut into grammars, packrat parser generators can generate packrat parsers that only require almost constant space for memoization in practice. However, checking whether we have sufficient cut operators inserted in grammars is not an easy problem. Grammar writers have to check grammars carefully and measure heap usage by actually running the generated parser. In this presentation, we propose a static anaylysis method for PEG grammars with cut operators to automatically estimate the amount of space used by the generated parsers. Applying our method, grammar writers can automatically check whether sufficient cut operators are inserted without actually running the parser. Our method resolves an important and practical problem of the packrat parser generator proposed in our previous study. And also, our method provides a measure to the impact of cut operators on the space complexity of packrat parsers generated from extended PEGs.

1. はじめに

Ruby や Perl などといったスクリプト言語と呼ばれる種類のプログラミング言語が，現在，Web アプリケーションの開発をはじめとする広い分野で一般的に使われるようになっている．スクリプト言語の定義はあまり明確でないが，スクリプト言語は C や Java といった従来の言語と比べて，文法が複雑である傾向があり，yacc[1)]（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) などの構文解析アルゴリズムを用いたパーザでは解析するのが困難である．

各スクリプト言語の処理系は ad hoc な手法をと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．たとえば，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 8,000 行以上に及ぶ．また，たとえば，Ruby などのスクリプト言語は，以下のように式を埋め込み可能な文字列リテラルを持っているが，

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

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

†1 筑波大学システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba

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 1 つとして packrat parsing という構文解析アルゴリズムが存在し，近年注目を集めている．Packrat parsing は PEG と呼ばれる形式文法が取り扱える範囲の文法を扱うことができ，これは LALR(1) より強力であるうえに，字句解析器を必要としないアルゴリズムであるため，複雑な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と考えられる．一方で，packrat parsing にはメモ化のために必要な記憶領域が $O(n)$ になるという欠点が存在する．我々の先行研究では，この欠点を提案するためにカット演算子を PEG に導入して，パーザ生成系のユーザがカット演算子を PEG による文法定義に手動で挿入することで，メモ化のために必要とする記憶領域の大きさが有界であるようなパーザ生成系を提案した．また，実験の結果，実用的な文法に対してそのようなパーザを生成できることを確認した．

しかし，我々の先行研究では，カット演算子を PEG による文法定義にどの程度挿入すれば，メモ化のために必要とするメモリ領域の大きさが有界になるかについては自明ではなく，パーザ生成系のユーザは実際に生成されたパーザを大きなサイズの入力に対して走らせてみることでしかそのことを確かめることができなかった．本論文では，この問題点を解決するために，カットが挿入された PEG を解析して，その PEG から，メモ化のために必要とする記憶領域の大きさが有界になるようなパーザを生成できるか否かを判定するための手法を提案する．

本論文の残りの構成は次のようになっている．まず，2 章では，packrat parsing の概要について説明する．特に，従来の構文解析手法に比べて packrat parsing がどのような利点および問題点を持っているかについて説明する．3 章では，packrat parsing の理論的な基盤となっている形式文法である PEG の概要を説明する．ここでは，PEG を構成する演算子について，その非形式的なセマンティクスおよび BNF と異なる点について説明する．4 章では，packrat parsing の問題点である，必要とする記憶領域が $O(n)$ になるという問題点を解決するために提案したカット演算子について説明する．特に，カット演算子を加えることで，どのようにセマンティクスが変化するか，カット演算子を導入することでどのような利点および問題点が発生するかについて説明する．5 章では，生成されたパーザがメモ化のために必要とする記憶領域の大きさが有界になるかどうかが明確ではないというカット演算子の問題点を解決するために本論文で提案する，カット付き PEG の静的解析手法について説明する．ここでは，提案手法を導入するにあたって導入した，*bounded* 修飾子や有界な式，有界なメモリ量で解析可能な式といった概念について説明する．7 章では，提案手法の有効性を確かめるために行った実験について述べる．それに続く 8 章では，7 章における実験の過程で得られた，*bounded* 修飾子の付加方法に関する知見について説明する．最後に，9 章において，まとめを行う．

2. Packrat Parsing（Ford02）

Packrat parsing [2] は Ford が 2002 年に発表した構文解析アルゴリズムである．Parsing expression grammar（PEG）[3] という形式文法で表現できる任意の文法を線形時間で解析することができる．これは，決定的な任意の LR(k) 言語および一部の文脈依存言語を含むため，現在主流である yacc が生成する LALR(1) パーザよりも強力である．また，packrat parsing は字句解析を必要としないアルゴリズムであり，前述の式を埋め込み可能な文字列といったものも簡単に解析できる．

Packrat parsing の概念はシンプルであり，単に backtrack recursive descent parsing（以下 backtrack parsing）に対してメモ化を追加したものである．Backtrack parsing では構文規則を，解析対象の部分文字列を引数にとる構文解析関数として実装する．関数は解析が成功したか失敗したかを示す情報を返し，成功した場合には非終端記号にマッチした部分を入力から除いた残りの部分文字列も返す．構文規則の右辺に複数の選択肢があった場合は，最初の選択肢をまず試し，成功した場合はその結果を返し，失敗した場合はバックトラックして次の選択肢を試すという動作を行う．Packrat parsing ではそれに加えて，各部分文字列に対する解析結果をメモ化しておき，同じ部分文字列に対して同じ解析関数が呼び出された場合はメモ化しておいた結果を返すという動作を行う．Backtrack parsing では，入力の長さに対して最悪の場合解析に指数関数時間かかるのに対して，packrat parsing ではメモ化によって線形時間で解析を行うことができる．

しかし，packrat parsing は，すべての解析結果についてメモ化を行うため，必要な記憶領域が入力長に対して線形に増大するという問題点がある．この点は必ずしも問題になるわけではないが，プログラムが機械的に生成した巨大なソースコードや，XML などのように，サイズが巨大になりうる種類の入力を扱う場合に問題となる．このため，巨大な XML ファイルの解析などには packrat parsing には向かないとされている [2]．2 つ目の問題点は，packrat parsing はバックトラックやメモ化を行うために，LALR(1) などに比べて一般に実行性能が悪いという点である．この点も，同様に巨大なサイズのファイルを解析するなど

の場合に問題になる．

3. Parsing Expression Grammar（Ford04）

Packrat parsing 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 についても簡単な説明を行っておく．PEG は，BNF に類似した記法を用いて文法を表現する．PEG は任意個の構文規則の集合 R および開始記号 S（厳密には開始式であり，任意の式をとることができるが，ここでは非終端記号のみをとることとする）からなっており，構文規則は $N \leftarrow e$ という形をとる．N は非終端記号であり，e は式（parsing expression）を表す．**図 1** は PEG の式の主要な構成要素である．

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．これは，e_1 / e_2 が単に e_1 と e_2 のどちらかにマッチするという定義ではなく，最初に e_1 にマッチするかどうか試して，失敗したらバックトラックして e_2 にマッチするか試すという**動作**を表しているためである．また，$*$ は正規表現における繰返しと異なり，1 度繰返しにマッチしたら，後続の式が失敗してもバックトラックを行わない点に注意する必要がある．これは，$e_1 *\ e_2$ という式があったとき，e_1* にマッチした後に e_2 で失敗しても，e_1 にバックトラックして繰返しをやり直すことはなく，$e_1 *\ e_2$ 全体が失敗するということである．このため，たとえば，`"a"` $*$ `"a"` という式は**どんな入力にもマッチしない**．この動作は，最近の言語における正規表現ライブラリでサポートされている強欲な繰返し演算子と呼ばれる演算子の動作に類似している．& と ! は先読みを行う演算子であり，e にマッチするかを試して，前者は成功する場合に，後者は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という特徴がある．

ε	:	空文字列
`"`s`"`	:	文字列リテラル
N	:	非終端記号
$e_1\ e_2$	:	式の列
e_1 / e_2	:	優先度付き選択
$e*$	:	0 回以上の繰り返し
$\&e$	:	And-predicate
$!e$	:	Not-predicate

図 1　PEG を構成する式
Fig. 1　Expressions constituting PEG.

4. カット演算子（Mizushima2008）

我々の研究では，packrat parsing の問題点を改善するための手法として，PEG へのカット演算子（以下カット）の導入を提案した[4]．カットは Prolog[5] から借用した概念であり，$e_1 \uparrow e_2$ / e_3 という形で，/ の左辺に挿入するか，繰返し演算子の本体に $(e_1 \uparrow e_2)*$ という形で挿入できる演算子として定義する．カットの動作はおおまかに次のように定義される．

- $e_1 \uparrow e_2$ / e_3:
  （1） e_1 が評価される．
  （2） （1）で e_1 が失敗した場合，e_3 が評価される．それ以外の場合，e_2 が評価されるが，このとき，たとえ e_2 が失敗しても e_3 は評価されない．カットが挿入されていなければ，e_2 が失敗した場合，e_3 が評価される．
- $(e_1 \uparrow e_2)*$:
  （1） e_1 が評価される．
  （2） （1）で e_1 が失敗した場合，バックトラックして式全体が**成功**する．そうでなければ e_2 が評価される．
  （3） （2）で e_2 が失敗した場合，式全体が必ず**失敗**する．そうでなければ，（1）に戻って評価を繰り返す．カットが挿入されていなければ，e_2 が失敗した場合も，バックトラックして式全体が成功する．

  ここで，$e*$ という式は N と $N \leftarrow e\ N$ / ε（ただし，N はすでにある非終端記号の名前と衝突しない新しい名前を持った非終端記号であるものとする）という式に展開することができるため，$(e_1 \uparrow e_2)*$ という式も同様にして，N と $N \leftarrow e_1 \uparrow e_2\ N$ / ε という式に展開することができる．

これらの動作は，式の評価時にカットを通過した後は，バックトラックして他の選択肢を試行しないといい換えることができる．

次に，カット（↑ 記号で表記）を使って定義した**図 2** の算術式の PEG に入力 `"a+b+a;"` が与えられた場合を例として，カットがどのように動作するかについて説明する．なお，説明のために，packrat parser の状態を，PEG 中の特定の位置を指し示す `<X>` と入力中の解析位置を表す整数 i（0 オリジン）の組 (`<X>`, i) と，バックトラックのために使うスタック，メモ化テーブルの 3 つからなるものとして簡略化し，関数の呼び出しスタックは省略する．

M ← E";";
E ← <E1> P "+" <E2> ↑ E / <E3> P;
P ← "a" / "b"

図 2　算術式の PEG
Fig. 2　PEG that expresses mathematical expressions.

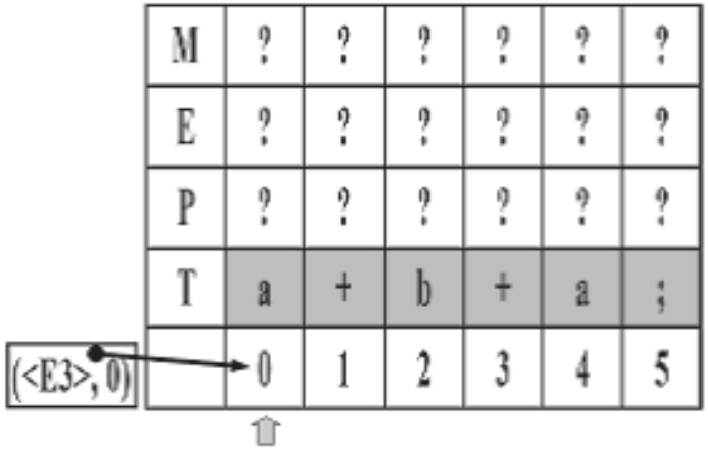


図 3　(<E1>,0) におけるバックトラック用スタックとメモ化テーブル
Fig. 3　Backtracking stack and memoization table at (<E1>,0).

まず，入力 a+b+a; に対して，パーザが M から解析を開始するものとする．パーザが (<E1>,0) 上にあるとき，バックトラック用スタックとメモ化テーブルは**図 3** のようになる．なお，テーブルについては，行の添え字が非終端記号を，最後の行が入力中の位置（0 オリジン）を，T の行が入力文字列を構成する各文字を，それ以外の表の要素が解析の中間結果を表すものする．また，表の要素には，対応する部分が未解析の場合は ? が，そうでない場合は残りの文字列を解析する場合の開始位置を表す添え字が入っているものとする．ブロック矢印は入力文字列中のどこを解析しているかを表し，表の中で塗り潰されている箇所は残りの文字列を表している．

図 3 は，P "+" E を評価する前にバックトラックに備えて (<E3>,0) をスタックにプッシュしなければならないことを示している．この時点でバックトラック用スタックが空ではなく，位置 0 がスタックにプッシュされているため，パーザはメモ化用テーブルのための領域を解放することができない．しかし，(<E2>, 2) において，バックトラック用スタックおよびメモ化テーブルは**図 4** のようになる．

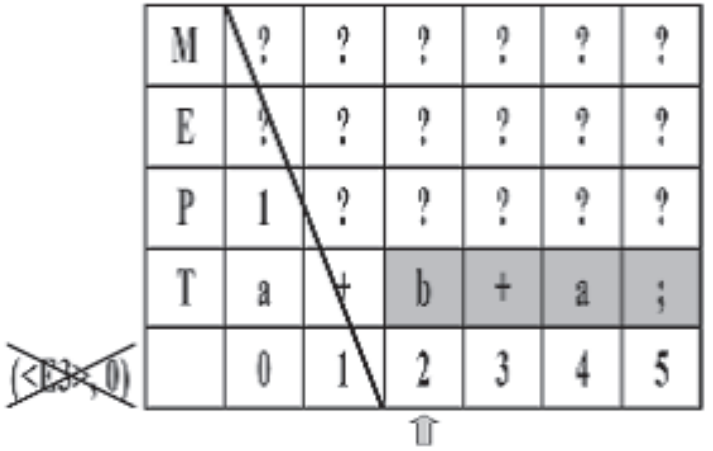


図 4　(<E2>,2) におけるバックトラック用スタックとメモ化テーブル
Fig. 4　Backtracking stack and memoization mable at (<E2>,2).

図 4 は，カットの評価によって，この地点でバックトラック用スタックが空になったということを示している．したがって，パーザは位置 2 より前には決してバックトラックしないため，メモ化テーブルのうち，斜線を引いた箇所のための領域を解放して再利用することができる．

カットを導入することで，2 章であげた 2 つの問題点を改善できる．まず第 1 の問題点について考える．/ の左辺の選択肢を試す際には，バックトラックに備えて，文字列中の現在の位置をスタックにプッシュするという動作を行うことになる．一方，カットを通過した場合はバックトラックは起きず，プッシュした情報は不要となるのでスタックに保存した情報をポップすることができる．カットを通過した場合や選択肢の片方の評価を終えた場合などで，スタックが空になった場合，パーザはその時点で解析中の位置 i より前には決してバックトラックしないため，文字列中の 0 から $i-1$ 番目について，メモ化のための記憶領域を除去できる．次に第 2 の問題点について考えると，カットを適切に挿入することでバックトラックの回数を削減できるうえに，メモ化のために保持するデータ量を削減できるために，

ガベージコレクションによるオーバヘッドを減らすことができると考えられる．

4.1　カットの性質

この節では，前節で述べたカットによるメモリ効率の改善が，具体的にどのような性質を持つのかについて述べる．まず，前節のカットを挿入した算術式の PEG について，M に対して入力を構文解析することを考える．カットが入っているため，P `"+"` にマッチした後，現在の入力位置までのメモ化領域を除去することができる．P にマッチする文字列の長さはつねに 1 であるため，2 文字読むたびに現在の入力位置までのメモ化領域を除去できる．つまり，メモ化に必要なメモリ領域は O(1) である．

次に，算術式に括弧を許すように拡張した以下の PEG について考える．この場合，P の最大長は入力中に現れる括弧で囲まれた算術式の最大長によって決まる．そのため，入力文字列中に現れる括弧で囲まれた算術式の最大の長さを s とすると，メモ化に必要なメモリ領域は O(s) になる．

$$
\begin{aligned}
M &\leftarrow E\texttt{";"};\\
E &\leftarrow P\ \texttt{"+"}\ \uparrow\ E\ /\ P;\\
P &\leftarrow \texttt{"("}\ E\ \texttt{")"}\ /\ \texttt{"a"}\ /\ \texttt{"b"};
\end{aligned}
$$

ここで，E 中に現れる P をくくり出すとともに，`"("` の直後にカットを入れて以下のようにすると，`"("` まで解析した時点で他の選択肢にバックトラックする可能性がなくなるので，メモ化に必要なメモリ領域は最初のものと同様に O(1) になる．

$$
\begin{aligned}
M &\leftarrow E\texttt{";"};\\
E &\leftarrow P\ (\texttt{"+"}\ \uparrow\ E\ /\ \varepsilon);\\
P &\leftarrow \texttt{"("}\ \uparrow\ E\ \texttt{")"}\ /\ \texttt{"a"}\ /\ \texttt{"b"};
\end{aligned}
$$

このように，カットを挿入することによってどの程度メモリ効率を改善できるかは，文法の性質とどの程度カットを挿入するかによって変化する．一般的なプログラミング言語などの文法においては，式や文など，実用的にはサイズが有界であるような要素の長さによってメモ化のために必要な領域の大きさが決まるようにカットを挿入して，実用的な入力についてメモ化に必要な領域の空間計算量が実質的に O(1) になるようにすることが可能である．この節ではメモ化に必要な記憶領域の除去について述べたが，パーザが保持している入力文字列のために必要な記憶領域も同様に除去することができる．なお，以後，単に必要なメモリ領域が有界，有界なメモリ領域などというときは，主にメモ化のために追加で必要なメモリ領域について述べている点に注意されたい．カットを十分に挿入された PEG から生成されたパーザであっても，n 重にネストするような要素を解析する場合は，関数呼び出しのために最悪 O(n) のスタック領域を必要とする可能性がある．

4.2　カットの効果

PEG をカットによって拡張したうえで，手動で PEG による文法定義に対してカットを挿入することで，実用的な文法において，メモ化に必要な空間計算量が実質的に O(1) にできることおよび実行効率を改善できることを確かめるために，我々の先行研究において実験を行った[4]．

実験のために，カットによって拡張された PEG を認識できる packrat parser 生成系 *Yapp* を開発した．Java 1.4 および XML のサブセットの PEG を用い，それらにカットを手動で挿入することで，生成されたパーザのメモリ効率および実行効率がどのように改善できるかを調べた．その結果，Java 1.4 の PEG および XML のサブセットの PEG から生成されたパーザの両方において，入力のサイズにかかわらず，ほぼ同じで十分に小さいメモリ量で構文解析を完了することを確かめることができた．また，カットを挿入した PEG から生成されたパーザはそうでないものに比べて，特にヒープサイズが小さい場合に顕著な実行効率の向上が見られた．

4.3　カットの問題点

前節で，PEG をカットによって拡張することで，実用的な文法については有界なメモリ領域で構文解析を行えるパーザを生成できることを述べたが，カットには大きな問題点が 2 つ存在する．

1 つ目の問題点は，カットは文法定義を書く者が手動で挿入しなければならないという点である．カットには，使い方によっては，文法の意味を変えてしまうことがある．これは，カットの挿入によって，本来は起こるバックトラックが起きなくなってしまうことがあるためである．たとえば，4 章における算術式の PEG をベースに，カットを挿入する位置だけを変えた以下の PEG について考える．

$$
\begin{aligned}
E &\leftarrow P\ \texttt{"+"}\ \uparrow\ E\ /\ P\ \texttt{"-"}\ \uparrow\ E\ /\ P\\
P &\leftarrow \uparrow\texttt{"a"}\ /\ \texttt{"b"}\ /\ \texttt{"c"}
\end{aligned}
$$

ここで，P の最初の選択肢で `"a"` の前にカットが挿入されているため，`"a"`,`"a+a"`, `"a-a"`,... という形以外の式を解析できなくなってしまう．我々の研究におけるカットは，純粋に最適化のために導入したものであり，このように文法の意味を変えてしまう使い

方ができてしまうのはカットの欠点である．この問題を解決するために，我々の研究では，意味を変えない範囲でカットを自動的に挿入する手法を提案した[6] が，本論文ではこの点については取り扱わない．

2 つ目の問題点は，カットをどの程度挿入すれば，生成されたパーザが有界なメモリ量で構文解析を行えるかどうかを形式的に判定する基準が存在しないことである．4.1 節で述べたように，カットを挿入することによってどの程度メモリ効率が改善できるかは，どの程度カットを挿入するかに依存するが，どの程度挿入すれば十分かは，パーザ生成系のユーザが直観で判断するしかない．その判断が妥当かどうかを知るためには，生成されたパーザを，いくつかの異なるサイズの入力に対して走らせ，個々のファイルごとに構文解析を行うのに必要な最小のヒープサイズを求めて，入力のサイズの変化に従って最小のヒープサイズがどのように変化するのかを調べる必要があるが，これには少なくない時間がかかる．我々の先行研究[4),6)] では，Java プログラムを生成するパーザ生成系を用いてこのような測定を行ったが，Java の場合，Java VM が使用する最大のヒープサイズをユーザが指定することはできるものの，そのうちでメモ化のために使われているヒープ領域がどの程度かを測定するのは容易ではない．そのため，我々の研究では，最大ヒープサイズを指定した二分探索を用いて，構文解析のために必要な最小のヒープサイズを求めたが，これは 1 回の実験につき少なくとも数十分（データセットのサイズによる）かかった．

実用でカットを使う場合には，カットを挿入 → カットが十分挿入されたか確かめる（必要なヒープサイズの変化を確かめる）→ カットを挿入 →…というステップを繰り返すことになるため，カット付き PEG を取り扱えるパーザ生成系を実用に使ううえで，これは大きな問題である．以降の章では，この問題点を解決するために，カットによって拡張された PEG による文法定義を解析し，有界なメモリ量で構文解析を行える程度に十分にカットがされているかを自動的に判定するための手法について取り扱う．なお，ここではメモ化のために追加で必要とするヒープサイズのみを問題としており，処理系が関数呼び出しのために消費するスタックサイズについては議論の対象としないことにする．

5. 提 案 手 法

カットが十分に挿入されているかを自動的に判定するために，我々は，生成されたパーザのバックトラック用スタックのサイズが 0 であるかどうかと，ある PEG の式に対して，入力文字列がその式に対して属するかどうかを判定するために走査する必要がある接頭辞の長さが有界であるかどうか（以降，このような式を有界な式と呼ぶ）という 2 点に着目する．カットなどによってバックトラック用スタックのサイズが 0 になったときに，その時点での解析位置より前のメモ化テーブルのための領域を解放できるのだから，解析中につねにバックトラック用スタックのサイズがつねに 0 であれば解析位置が 1 進むごとにメモ化テーブルのための領域を解放することができるから，つねに有界なメモリ量で解析を行うことができる．実際には，バックトラック用スタックのサイズがつねに 0 であるということはありえないが，そのような（スタックのサイズが 1 以上）場合でも，解析している式の長さが有界であれば有界なメモリ量で解析できると考えられる．

ある文脈においてバックトラック用スタックのサイズが 0 かそうでないかは，PEG の式を再帰的に探索することで簡単に調べることができるが，ある式が有界かどうかを機械的にうまく判定できるとは限らない．たとえば，Java 1.4 の PEG における識別子を表現した次の規則について考える．

```
Identifier    ←  !Keyword Letter LetterOrDigit* Spacing?;
Letter        ←  [a-z] / [A-Z] / [_$];
LetterOrDigit ←  [a-z] / [A-Z] / [0-9] / [_$];
```

ここで，非終端記号 Identifier が表現する言語に属する文字列（`foo`，`bar` など）は理論上は任意の長さをとりうる．たとえば，10,000 文字の長さを持った識別子といったものを考えることができる．一方で，そこまで巨大な識別子は**実用上**まずありえないことを，文法定義を作成するユーザは知っている．しかし，そのような知識をパーザ生成系が学習なしに得ることは困難であるため，提案手法では，このような場合に，ある式が有界であることをユーザがパーザ生成系に明示的に教えるというアプローチをとることにした．次の節では，ある式が（実用上）有界であることをユーザがパーザ生成系に教えるための方法として導入した *bounded* 修飾子について述べる．

5.1 bounded 修飾子

ある文法規則を参照する非終端記号やそれ以外の通常の式について，それが有界であることをパーザ生成系に教えるために，*bounded* 修飾子を導入する．*bounded* 修飾子は次の 3 つの形のいずれかをとる．

- *bounded* 規則：構文規則ごとに付加し，付加された規則を参照する非終端記号が有界であることを示す．たとえば，

```
bounded Identifier  ←  !Keyword Letter LetterOrDigit* Spacing?;
```

という規則は，非終端記号 Identifier が有界であることを示す．

- *bounded* ブロック：構文規則の集まりに付加し，付加されたブロックの中に含まれる規則を参照するすべての非終端記号が有界であることを示す．たとえば，

```
                    bounded{
Identifier      ←   !Keyword Letter LetterOrDigit * Spacing?;
Letter          ←   [a-z] / [A-Z] / [_$];
LetterOrDigit   ←   [a-z] / [A-Z] / [0-9] / [_$];
                    }
```

  は，非終端記号 Identifier，Letter，LetterOrDigit のすべてが有界であることを示す．*bounded* ブロックは多数の規則を *bounded* にする手間を削減するためのシンタックスシュガーであり，*bounded* ブロックに含まれるすべての規則に *bounded* 修飾子を付加したのと同じである．

- *bounded* 式：特定の式に付加し，付加された式が有界であることを示す．たとえば，

```
Identifier  ←   !Keyword Letter bounded {LetterOrDigit*} Spacing?;
```

  は，式 LetterOrDigit* が有界であることを示す．
  bounded 修飾子によって，理論上は任意の長さをとりうるが，**実用上は**有界であると見なしてよい規則や式をユーザがパーザ生成系に教えることができる．

5.2　有界な式の判定

bounded 修飾子をもとにして，ある式が有界かどうかを次の手続き Bounded によって判定することができる．ただし，Q は *bounded* 修飾子が付加された式（*bounded* 規則の場合，その規則を参照する非終端記号）の集合であり，V は探索の過程で訪れた非終端記号の集合である．

$$
\begin{aligned}
\mathrm{Bounded}(e) &= B(e, \emptyset) \\
B(e, V) \text{ if } e \in Q &= \mathrm{true} \\
B(N, V) \text{ if } N \in V &= \mathrm{false} \\
B(N, V) &= B(e, V \cup \{N\})\ (N \leftarrow e) \\
B(e_1/e_2, V) &= B(e_1, V) \wedge B(e_2, V) \\
B(e_1 \uparrow e_2/e_3, V) &= B(e_1, V) \wedge B(e_2, V) \wedge B(e_3, V) \\
B(e_1 e_2, V) &= B(e_1, V) \wedge B(e_2, V) \\
B(e*, V) &= \mathrm{false} \\
B((e_1 \uparrow e_2)*, V) &= \mathrm{false} \\
B(\varepsilon, V) &= \mathrm{true} \\
B(\texttt{" "}, V) &= \mathrm{true} \\
B(\&e, V) &= B(e, V) \\
B(!e, V) &= B(e, V)
\end{aligned}
$$

これらの式の意味は，次のように理解することができる．手続き B の 1 行目は，ある式 e について e が *bounded* 修飾子が付加された式であるならば有界であるということを，2，3 行目は再帰的に定義された非終端記号は（*bounded* 修飾子が付加されない限り）有界でないということを，4，5 行目はカット付きであるかどうかにかかわらず，優先度付き選択は，それを構成するすべての式が有界である場合にのみ有界であるということを示している，といった具合である．

5.3　有界なメモリ量で解析可能な式の判定

前の節で定義した手続き Bounded を使って，**有界なメモリ量で解析可能な**式を判定する手続き SBounded を次のようにして構成することができる．C は現在解析中の規則の名前を，D はダミーの非終端記号（既存の規則とかぶらない任意の名前）を表すものとする．手続き S は，有界なメモリ量で解析可能な式であるかどうかを判定するとともに（結果が $\emptyset$ の場合に有界なメモリ量で解析可能），そうでない場合は原因と思われる式と式が現れた規則名の集合を抽出する．これは，ある PEG による文法定義に対して有界なメモリ量で解析可能でないと判断された場合に，ユーザが文法定義のどの部分を修正すればよいのかを提示するために使われる．

$$
\begin{aligned}
\mathrm{SBounded}(e) &= S(e, \emptyset, D) = \emptyset \\
S(e, V, C) \text{ if } e \in Q &= \emptyset \\
S(N, V, C) \text{ if } N \in V &= \emptyset \\
S(N, V, C) &= S(e, V \cup \{N\}, N)\ (N \leftarrow e) \\
S(e_1/e_2, V, C) \text{ if Bounded}(e_1) &= S(e_2, V, C) \\
S(e_1/e_2, V, C) &= \{(e_1, C)\} \cup S(e_2, V, C) \\
S(e_1 \uparrow e_2/e_3, V, C) \text{ if Bounded}(e_1) &= S(e_2, V, C) \cup S(e_3, V, C)
\end{aligned}
$$

$$
\begin{aligned}
S(e_1 \uparrow e_2 / e_3, V, C) &= \{(e_1, C)\} \cup S(e_2, V, C) \cup S(e_3, V, C) \\
S(e_1 e_2, V, C) &= S(e_1, V, C) \cup S(e_2, V, C) \\
S(e*, V, C) \text{ if Bounded}(e) &= \emptyset \\
S(e*, V, C) &= \{(e, C)\} \\
S((e_1 \uparrow e_2)*, V, C) \text{ if Bounded}(e_1) &= S(e_2, V, C) \\
S((e_1 \uparrow e_2)*, V, C) &= \{(e_1, C)\} \cup S(e_2, V, C) \\
S(\varepsilon, V, C) &= \emptyset \\
S(\texttt{" "}, V, C) &= \emptyset \\
S(\texttt{\&}e, V, C) \text{ if Bounded}(e) &= \emptyset \\
S(\texttt{\&}e, V, C) &= \{(e, C)\} \\
S(\texttt{!}e, V, C) \text{ if Bounded}(e) &= \emptyset \\
S(\texttt{!}e, V, C) &= \{(e, C)\}
\end{aligned}
$$

これらの式は基本的に先ほどの Bounded 手続きと同様に読むことができるが，S の定義の 4〜5 行目と 6〜7 行目，9〜10 行目と 11〜12 行目の違いに注意する必要がある．これらの行は，カットが挿入されていない場合には，/ の左辺の式および繰返し（∗）対象の式は必ず有界でなければならないが，カットが挿入されている場合は，カットを通過した後の式は有界でなくても，有界なメモリ量で解析可能な式であればよいということを意味している．定義より，有界な式である条件よりも有界なメモリ量で解析できる式である条件の方が緩いため，カットを適切に挿入することで，パーザ生成系はより正確に判定を行うことができる．

また，先読み演算子（& および !）を使った式については，それらが有界なメモリ量で解析できる式であるためには，先読み対象である式が必ず有界でなければならないという点にも注意する必要がある．これは，先読み演算子は，先読み対象の式が成功しても失敗しても入力を消費しないためである．

この手続きは直観的には，解析を開始する時点でのバックトラック用スタックのサイズを 0 としたときに，スタックサイズが 1 以上の文脈で参照される式，つまり失敗時にバックトラックが発生する可能性がある式の場合は，それ自体が有界な式でなければならず，そうでない文脈，つまりスタックサイズが必ず 0 の文脈で参照される場合，有界な式でなくても，有界なメモリ量で解析できる式であればよいということを示している．この手続きは，再帰的に PEG の式を探索して，構文解析時にバックトラック用スタックのサイズがどのように変化するかを，実際に構文解析を行わずにシミュレートしているともいえる．

5.4 実　行　例

提案手法を実装したパーザ生成系に，2 つの異なる PEG を入力として与えた結果を以下に示す．カットの挿入の有無によって有界なメモリ量で解析できるかどうかの判定が変化していることが分かる．また，5.3 節の手続きによって，有界なメモリ量で解析できない場合，原因となったと考えられる式も提示されていることが分かる．

- 入力：

$$
\begin{aligned}
M &\leftarrow E\texttt{";"}; \\
E &\leftarrow P\ (\texttt{"+"} \uparrow E \,/\, \varepsilon); \\
P &\leftarrow \texttt{"("}\ E\ \texttt{")"} \,/\, \texttt{"a"} \,/\, \texttt{"b"};
\end{aligned}
$$

- 出力：

```
A parser generated from this grammar may require unbounded space
because the following expressions are unbounded:
E in P
```

- 入力：

$$
\begin{aligned}
M &\leftarrow E\texttt{";"}; \\
E &\leftarrow P\ (\texttt{"+"} \uparrow E \,/\, \varepsilon); \\
P &\leftarrow \texttt{"("} \uparrow E\ \texttt{")"} \,/\, \texttt{"a"} \,/\, \texttt{"b"};
\end{aligned}
$$

- 出力：

```
A parser generated from this grammar requires only bounded space.
```

6. 関連研究との比較

この章では本研究と関連研究との比較を行う．まず，2 章で述べた packrat parsing の問題点を改善する研究としては，Grimm によるものが存在する[7]．Grimm の研究では，packrat parser 生成系 *Rats!*を提案している．*Rats!*は PEG 風の文法定義を入力として与えると，Java 言語によるパーザを自動的に生成する．*Rats!*では，メモ化テーブルのカラムを複数のチャンクと呼ばれる複数のオブジェクトに分割し，必要になった段階で各チャンクの中にメモ化領域を確保することで不要なオブジェクトの確保を削減する Chunks 最適化や，ユーザが指定した特定の規則についてメモ化の対象としないようにする Transient 最適化などの

いくつかの最適化を行うことで，LALR(1) パーザ生成系などと比べても，実行性能の点でさほど劣らない効率の良いパーザを生成することが可能になっている．一方で，*Rats!*の各種最適化は空間計算量は通常の packrat parser と同じ O(n) のままであり，係数が異なるだけである．これは，packrat parsing では，メモ化のために構文規則の数 m（入力によって変化しないため，定数と見なすことができる）と入力長 n に対して，一般にサイズ cmn（c は適当な定数）のテーブルが必要であるのに対して，*Rats!*の各種最適化は c および m の値を減らすものの，n についてはそのままであるからである．

また，別の研究として PEG パーザ生成系 *Mouse* が存在する[8]．通常の packrat parser がすべての解析結果をメモ化するのに対して，*Mouse* が生成したパーザは小さな固定長のキャッシュのみを必要とする．そのため，*Mouse* はメモ化のために O(n) の記憶領域を必要としない．しかし，そのため，*Mouse* が生成したパーザは入力を線形時間で解析できることを保証しない．また，*Mouse* ではバックトラックの可能性があるため，解析が完了するまで入力文字列をすべて記憶しておく必要があるため，入力文字列を保持しておくために O(n) の記憶領域が必要である．

これらの研究と比較すると，我々のパーザ生成系によってカットが十分に挿入された PEG から生成され，有界なメモリ量で解析できると判定されたパーザは線形時間の解析を保証しながらも，有界なメモリ量で解析を行うことができるという利点がある．また，*Rats!*が提案している各種の最適化はカットと直交するため，容易に併用することが可能である．また，*Mouse* と異なり，我々のパーザ生成系は，カットのおかげで入力文字列をすべて保持しておく必要がなく，途中で要らなくなった部分文字列を解放して，そのための領域を再利用することができる．

7. 実　　験

提案手法の効果を検証するための実験を行った．まず実験のために，我々が先行研究で実装したカットの機能を持つ packrat parser 生成系 *Yapp* を拡張して提案手法を実装し，カット付き PEG を解析してパーザがメモ化のために必要とするメモリ量が有界であるかどうかを判定できるようにした．次に，カットが十分に挿入された Java 言語（Java 1.4），XML のサブセットおよび JSON の PEG による文法定義[⋆1]を *Yapp* に与えて，どの程度 *bounded* 修飾子を付加すれば，これらの文法定義から生成されたパーザが有界なメモリ量で解析できることを判定できるかを確かめた．なお，Java の文法定義については，PEG for Java 1.5 をもとに，Java 1.5 で追加された文法を取り除いたものになっており，XML の文法定義については，Extensible Markup Language（XML）1.0[9] をもとに作成した最小限のサブセットになっている．JSON の文法定義は，ECMA-262[10] の仕様書をもとにしたもので，ほぼフルセットの仕様を実装している．

参考のために，以下にカットを挿入した Java の PEG の一部（**図 5**），XML のサブセットの PEG の一部（**図 6**）および JSON の PEG（**図 7**）の一部を例示する．構文規則 MemberDecl は Java 言語のクラスのメンバ宣言を，Element は XML の要素を，JSON は JSON ファイル全体を表現している．

実験の結果，各々の PEG において必要とされる *bounded* 修飾子の数は**表 1** のようになった．表中の各行は，元の PEG の文法定義中に明示的に付加した *bounded* 式，*bounded* ブ

```
MemberDecl ←
  Type Identifier FormalParameters ↑ ...
/ VOID Identifier FormalParameters ↑ ...
/ Identifier FormalParameters ↑ ...
/ &INTERFACE ↑ InterfaceDeclaration
/ &CLASS ↑ ClassDeclaration
/ Type VariableDeclarator
  (COMMA ↑ VariableDeclarator)∗
```

図 5　Java の PEG の一部
Fig. 5　A fragment of Java PEG.

```
Element ←
"<" NAME (S Attribute) ∗ S? (
  ">" ↑ Content "</" NAME S? ">"
/ "/>"
)
```

図 6　XML の PEG の一部
Fig. 6　A fragment of XML PEG.

⋆1 我々の先行研究[6] における実験の結果，有界なメモリ量で解析できることが確認できた PEG による文法定義を使用．

```
JSON ← S Object EOT;
Object ← LBRACE RBRACE / LBRACE Members RBRACE;
Members ← Pair (!(RBRACE) ↑ COMMA Pair)∗;
```

図 7　JSON の PEG の一部
Fig. 7　A fragment of JSON PEG.

表 1　Java/XML/JSON の PEG において必要な *bounded* 修飾子の数
Table 1　The number of *bounded* quallifiers needed in Java/XML/JSON PEG.

	Java	XML	JSON
規則の数	182	24	26
bounded 式	1	0	0
bounded ブロック	3	0	0
bounded 規則	5	8	4
bounded 規則（ブロックに含まれる規則を含む）	120	8	4

ロック，*bounded* 規則の数を表している．また，最後の行は，*bounded* ブロックの中に含まれる規則も含めてカウントした場合の *bounded* 規則の数である．

表 1 を見ると，XML および JSON の PEG については，比較的少数の規則を *bounded* にするだけでパーザが使用するメモリ量が有界であることを推定できたが，Java の PEG においては，多くの範囲を *bounded* ブロックで囲む必要があったことが分かる．これは，必ずしも Java のように規模の大きな文法の場合，多くの *bounded* 修飾子が必要ということを意味するわけではない．というのは，今回の実験では，*bounded* 修飾子を付加するのにそれほど時間をかけていないため，比較的粗く見積もって，*bounded* 修飾子を付加しているからである．たとえば，Java の PEG についていえば，字句的な要素に相当する規則をひとまとめにして，*bounded* ブロックで囲んでいる．また，Java の文（Statement）および式（Expression）を表す規則も *bounded* であると見なして，*bounded* 修飾子を付加している．

bounded 修飾子を付加するのにかかった時間は，XML および JSON の PEG については 10 分程度，Java の PEG については 1〜2 時間程度であったが，パーザのメモリ使用量を見積もるためにかけるコストとしては十分現実的であるといえる．次の章では，Java/XML/JSON の PEG に対して具体的にどのような判断に基づいて *bounded* を付加していったのかを具体的な例をあげながら述べる．

8. ケーススタディ：bounded 修飾子の付加方法に関する考察

本章では，7 章の実験において，Java/XML/JSON の PEG に *bounded* 修飾子を付加していく過程で得られた知見について説明を行う．特に，それぞれの文法定義において，どのように構文規則に対して *bounded* 修飾子を付加していったかを，具体的に例をあげて説明していく．

8.1 字句構造を表現する規則

実験において，*bounded* 修飾子を付加する対象の規則としてまず最初に候補としたのは，字句構造を定義した規則，つまり，字句解析をベースとしたパーザの構文定義においては終端記号に相当する規則である．これらの規則が表現する言語の文字列長が入力の長さに比例して長くなるようなケースはほとんど考慮する必要がないため，安全に *bounded* 修飾子を付加することができる．たとえば，Java の PEG では，**図 8** のように字句構造を表す非常に多数の構文規則がひとまとめにして定義されており，これらの規則については何も考えずに *bounded* ブロックで囲むことですべてを *bounded* と見なすことができる．なお，図中における ... は，一部分が省略されていることを示している．表 1 を見ると，Java の PEG に対する *bounded* ブロックの付加実験では，*bounded* ブロックの数は少ないにもかかわらず非常に多数の規則が *bounded* になっているが，これは，Java の PEG では字句構造を表現する規則の数が非常に多いからである．

また，XML や XML の PEG でも，同様に字句構造を表す一連の規則群が存在し（**図 9** および **図 10**），これらについても安全に *bounded* を付加することができる．ただし，図を見れば分かるように，XML および JSON の PEG に関しては，*bounded* を付加しても問題ない場合でも，あえて *bounded* を付加していないケースが存在する．これについては後述する．

8.2 非常に短い文字列を表現すると推定される規則

字句構造を表現したものではないが，非常に短い文字列を表現すると容易に推定される構文規則が存在しうる．そのような規則についても，字句構造を表す規則に準じるものとして，比較的安全に *bounded* を付加する対象とすることができる．たとえば，Java の PEG については，**図 11** の部分については，型名などを表現する規則であり，実用上は非常に短い長さにしかなりえないと考えられるため，これらの規則群に対しても，*bounded* ブロックによってまとめて *bounded* を付加した．JSON や XML の PEG については，後述するように，SBounded 手続きの情報をもとに，少しずつ *bounded* を付加するという手法をとっ

```
...
//=============================================================
//  Lexical Structure
//=============================================================
//-------------------------------------------------------------
//  JLS 3.6-7  Spacing
//-------------------------------------------------------------
bounded {
Spacing
    = ( [ \f\t\r\n]+              // WhiteSpace
      / "/*" (!"*/" _)* "*/"      // TraditionalComment
      / "//" (![\r\n] _)* [\r\n]  // EndOfLineComment
      )* ;
...
//-------------------------------------------------------------
//  JLS 3.8  Identifiers
//-------------------------------------------------------------

Identifier  = !Keyword Letter LetterOrDigit* Spacing?;

Letter = [a-z] / [A-Z] / [_$] ;
LetterOrDigit = [a-z] / [A-Z] / [0-9] / [_$] ;
...
ASSERT    = "assert"       !LetterOrDigit Spacing? ;
BREAK     = "break"        !LetterOrDigit Spacing? ;
CASE      = "case"         !LetterOrDigit Spacing? ;
CATCH     = "catch"        !LetterOrDigit Spacing? ;
CLASS     = "class"        !LetterOrDigit Spacing? ;
CONTINUE  = "continue"     !LetterOrDigit Spacing? ;
DEFAULT   = "default"      !LetterOrDigit Spacing? ;
DO        = "do"           !LetterOrDigit Spacing? ;
ELSE      = "else"         !LetterOrDigit Spacing? ;
...
SL_EQU          =   "<<="     Spacing?;
SR              =   ">>"![=>] Spacing?;
SR_EQU          =   ">>="     Spacing?;
STAR            =   "*"!"="   Spacing?;
STAR_EQU        =   "*="      Spacing?;
TILDA           =   "~"       Spacing?;

EOT = !_ ;
}
```

図 8　Java の PEG における字句構造を表す規則群
Fig. 8　Rules representing lexical structure in Java PEG.

たために，このような種類の規則に対してあえて *bounded* を付加することはなかったが，Java の PEG の場合と同様にすることは可能である．

8.3　多くのケースで比較的短い文字列を表現すると推定される規則

8.1 節と 8.2 節で述べたような種類の規則については，あまり深く検討せずに *bounded* を付加することができるが，それでもパーザが必要とするメモリ量が有界と判定できない場合は，実用上稀に非常に長くなることがあるが，多くのケースでは短い文字列を表現するような規則についても *bounded* を付加することを検討する必要がある．たとえば，Java の PEG については，8.1 節と 8.2 節で述べた種類の規則ほぼすべてに対してカットを挿入しても，SBounded 手続きでは有界であると判定することができなかった．そこで，Java の PEG については，Java のブロック文を表現する BlockStatement 規則や式を表現する Expression 規則などについても *bounded* を付加することにした．

しかし，前の 2 つの場合と異なり，このケースでは規則が表現する文字列の長さが実用上十分に短いことがあまり自明でないため，*bounded* を付加するときには注意が必要である．たとえば，プログラミング言語の処理系においては，非常に巨大な switch 文（条件分岐）

```
...
bounded VersionNum = ([a-zA-Z0-9_.:] / "-")+ ;
bounded Misc       = COMMENT / S ;
EncodingDecl       = S "encoding" Eq ("'" EncName "'"
                   / "\"" EncName "\"") ;
bounded EncName    = [A-Za-z] ([A-Za-z0-9._] / "-")* ;
bounded S          = (" " / "\t" / "\r" / "\n")+ ;

LETTER             = [a-zA-Z] ;
DIGIT              = [0-9] ;
NAME_CHAR          = LETTER / DIGIT / "." / "-" / "_" / ":" ;
bounded NAME       = (LETTER / "_" / ":") NAME_CHAR* ;
bounded ENTITY_REF = "&" NAME ";" ;
bounded CHAR_REF   = "&#x" [0-9a-fA-F]+ ";" / "&#" [0-9]+ ";" ;
REFERENCE          =  ENTITY_REF / CHAR_REF ;
EOT                = !_ ;

bounded ATT_VALUE  = "\"" ([^<&"] / REFERENCE)* "\""
             / "'"  ([^<&'] / REFERENCE)* "'" ;
...
```

図 9　XML の PEG における字句構造を表す規則群
Fig. 9　Rules representing lexical structure in XML PEG.

```
...
Number = (INT FRAC EXP / INT EXP / INT FRAC / INT) S ;
INT = ("-")? ([1-9] [0-9]* / "0");
FRAC = "." [0-9]+ ;
EXP = E [0-9]+ ;
E = "e+" / "e-" / "E+" / "E-" / "e" / "E" ;
TRUE = "true" S ;
FALSE = "false" S ;
NULL = "null" S ;
EOT = !_ ;
COMMA = "," S ;
COLON = ":" S ;
LBRACE = "{" S ;
RBRACE = "}" S ;
LBRACKET = "[" S ;
RBRACKET = "]" S ;
S = ( [ \f\t\r\n]+
          / "/*" (!"*/" _)* "*/"
          / "//" (![\r\n] _)* [\r\n]
          )* ;
```

図 10　JSON の PEG における字句構造を表す規則群
Fig. 10　Rules representing lexical structure in JSON PEG.

がしばしば利用されるため，SBounded 手続きでは有界であると判定されたにもかかわらず，そのような（巨大な switch 文を利用する）プログラムがパーザに対する入力として与えられた場合，メモ化のために多くのヒープを必要とする，といった事態が起こりうる．

8.4　SBounded 手続きの結果をもとにした改善

8.1 節において，字句構造を表現する規則が *bounded* の付加候補になると述べたが，XML や JSON の PEG においては，表 1 を見れば分かるように，字句構造を表現する規則の総数よりも少ない数の *bounded* しか付加されていない．これは，JSON や XML の PEG の規模が比較的小さいため（XML についてはサブセットであるため），最初は *bounded* を挿入せずに，SBounded 手続きを利用して *Yapp* が出力するメッセージをヒントに，少しずつ *bounded* を付加していくことで，*bounded* を付加する数を必要最小限に抑えたためである．たとえば，JSON の PEG を *Yapp* に与えると，**図 12** のように，空白文字の連続を表現した規則 `S` におけるコメントが不定の長さをとりうるために，必要なメモリサイズが有界と判定できないと報告されたため，それをもとに規則 `S` に *bounded* にするという作業を行った．

規則 `S` に *bounded* を付加した JSON の PEG を再度処理系に与えると，**図 13** のように，別の規則 `INT` が不定の長さをとりうるため，必要なメモリサイズが有界と判定できないと報告されたため，それをもとに規則 `INT` を *bounded* にするという作業を行った．

このように，*Yapp* のメッセージをもとに *bounded* を少しずつ付加していくことで，必要な *bounded* の数を最小限にすることができた．XML の PEG についても，*Yapp* のメッ

```
...
//-------------------------------------------------------------
//  Types and Modifiers
//-------------------------------------------------------------
bounded {
Type
    = (BasicType / ClassType) Dim*
    ;

ReferenceType
    = BasicType Dim+
    / ClassType Dim*
    ;

ClassType
    = Identifier (DOT Identifier)*
    ;

ClassTypeList
    = ClassType (COMMA ClassType)*
    ;

Modifier
    = ( "public"
      / "protected"
      / "private"
      / "static"
      / "abstract"
      / "final"
      / "native"
      / "synchronized"
      / "transient"
      / "volatile"
      / "strictfp"
      ) !LetterOrDigit Spacing?
    ;
...
}
```

図 11　Java の PEG における型名などを表す規則群
Fig. 11　Rules that include rules representing type names in Java PEG.

```
A parser generated from this grammar may require unbounded space
because the following expressions are unbounded:
("/*" ((!("*/") .))* "*/") in S
```

図 12　JSON の PEG において，bounded を付加しない場合のメッセージ
Fig. 12　Displayed message in JSON PEG if no bounded qualifiers are added.

```
A parser generated from this grammar may require unbounded space
because the following expressions are unbounded:
INT in Number
```

図 13　JSON の PEG において，規則 S に bounded を付加した場合のメッセージ
Fig. 13　Displayed message in JSON PEG if a bounded qualifier is added to rule S.

セージをもとに同様にして *bounded* を付加していくという作業を行った．Java の PEG についても同様に，*Yapp* のメッセージをヒントに *bounded* を付加することができたが，最初にある程度の *bounded* を付加してから *Yapp* に与えるようにした．これは，Java では字句構造を表現する規則が大量に存在するため，*bounded* を何も与えない状態から少しずつ *bounded* を付加していく方法では手間がかかるためである．

8.5　判定の精密さと bounded の付加コストのトレードオフ

8.3 節であげた例のように，*bounded* を付加しすぎることで，有界性の判定が実際よりも粗くなるケースが存在する．より一般的には，*bounded* には，構文規則に付加するためにかかる時間と，有界性に関する解析の正確さの間にある種のトレードオフが存在する．可能な限り精密に有界性の解析を行う場合，まず，明らかに有界であるような規則に関して

のみ最初は *bounded* を付加してから，SBounded 手続きの結果をもとに，有界でない原因である規則を観察し，対象となる規則が言語の性質上有界と見なして問題なさそうならば，*bounded* を付加し，そうでないならば規則の右辺の式に適切にカットを挿入することで問題の解消を試みる，というステップを繰り返すことになる．SBounded 手続き自体は対話的に利用する分には十分に短い時間で終了するので，SBounded 自体の実行時間は考慮しないとして，PEG の扱いに十分に慣れた文法定義作成者であれば，1 つの規則を観察するのに数分程度あれば十分と考えられる．規則が 100 以上あるような比較的規模の大きな文法であっても，経験的には，その多くは，明らかに十分短い文字列を表現するものであり，実際に人間が詳細に観察して *bounded* と見なしてよいかを判定し，必要ならばその中にカットを挿入する必要がある残りの規則は多くても数十個程度である．1 つの規則の処理に 5 分くらいかかるとすれば，デバッグの手間を含めても，数時間程度で十分精密に *bounded* を付加し終えることができると考えられる．

今回の実験では，Java の PEG は比較的規模が大きいことと，実験ですでに Java の PEG から生成されたパーザの有界性を確認できているため，追加でさらにカットを挿入する手間をかけないようにするために，ブロック文や式などを表現する規則にまで *bounded* を付加するという，比較的粗い見積りを行ったが，すでに述べたような条件を考えると，Java のようなそれなりの規模の文法を持つ言語であっても，精密に *bounded* を付加することは，十分現実的な時間で可能であると考えられる．

9. ま と め

本研究では，我々の先行研究で提案したカット付き PEG からパーザ生成系によって生成されるパーザについて，メモリ使用量が有界であるかどうかを見積もるための客観的な手法が存在しないという欠点を改善するために，カット付き PEG による文法定義を解析し，メモリ使用量が有界であるかどうかを判定するための手法を提案した．

提案手法では，特定の規則および式に，それらが表現する文字列の長さが有界であることをパーザ生成系に教えるための *bounded* 修飾子を導入し，それを利用して，任意の式について，それが表現する文字列が有界なメモリ量で解析できる式かどうかを判定している．また，Java，XML のサブセット，JSON のカット付き PEG について実際に *bounded* 修飾子を付加し，現実的なコストで既存のカット付き PEG について，有界なメモリ量で構文解析を行えることを判定できるように *bounded* 修飾子を付加することができた．パーザを実際に繰り返し走らせることなく，有界なメモリ量で構文解析を行えることを確信できるようになったのは本研究の大きな貢献であると考えている．

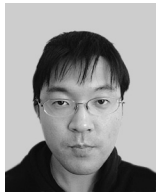
今後の課題としては，解析対象となる規則が有界なメモリ量で解析できない場合，どの規則がその原因であるかをパーザ生成系のユーザに正しく提示するための手法の開発があげられる．現在のところは，有界なメモリ量で解析できないと判定された式とその式が現れた規則の集合をアドホックに集めてユーザに提示しているが，より正確に原因となる箇所を提示することで，より短い時間で文法定義を修正できるようになると考えられる．また，カットをうまく挿入できそうな箇所を静的な解析で見つけて，ユーザにヒントとして提示する機能などがあれば，提案手法の実用性はより高まると考えられる．

参 考 文 献

1) Johnson, S.: Yacc: Yet Another Compiler Compiler, *UNIX Programmer's Manual*, Holt, Rinehart and Winston, New York, NY, USA, pp.353–387 (1979).
2) Ford, B.: Packrat Parsing: Simple, Powerful, Lazy, Linear Time, *Proc. 2002 International Conference on Functional Programming* (2002).
3) Ford, B.: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation, *Symposium on Principles of Programming Languages* (2004).
4) Mizushima, K., Maeda, A. and Yamaguchi, Y.: Improvement technique of memory efficiency of Packrat Parsing, *IPSJ Trans. Programming*, Vol.49, No.SIG 1(PRO 35), pp.117–126 (2008).
5) Colmerauer, A. and Roussel, P.: The birth of Prolog, *The 2nd ACM SIGPLAN conference on History of programming languages*, pp.37–52, ACM Press (1993).
6) Mizushima, K., Maeda, A. and Yamaguchi, Y.: Packrat parsers can handle practical grammars in mostly constant space, *PASTE '10: Proc. 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, New York, NY, USA, pp.29–36, ACM (2010).
7) Grimm, R.: Better Extensibility through Modular Syntax, *Proc. ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pp.19–28 (2006).
8) Redziejowski, R.: Mouse: from Parsing Expressions to a practical parser, *Concurrency Specification and Programming Workshop* (2009).
9) Bray, T., Paoli, J. and Sperberg-McQueen, C.: Extensible Markup Language (XML) 1.0 (4th Edition), W3C Recommendation (2006).
10) International, E.C.M.A.: ECMA-262: ECMAScript Language Specification, ECMA (European Association for Standardizing Information and Communication Systems), 5th edition (2009).
11) Hopcroft, J. and Ullman, J.: *Introduction to Automata Theory, Languages and*

Computation, Addison-Wesley (1979). オートマトン言語理論計算論 I，サイエンス社 (1984).
12) Erjavec, T.: The IJS-ELAN Slovene-English Parallel Corpus, *International Journal of Corpus Linguistics*, Vol.7, No.1, pp.1–20 (2002).
13) Redziejowski, R.: Some aspects of Parsing Expression Grammar, *Fundamenta Informaticae*, Vol.85, No.1-4, pp.441–454 (2008).
14) Redziejowski, R.: Applying classical concepts to Parsing Expression Grammar, *Fundamenta Informaticae*, Vol.93, No.1-3, pp.325–336 (2009).
15) Becket, R. and Somogyi, Z.: DCGs + Memoing = Packrat Parsing But is it worth it?, *Practical Aspects of Declarative Languages* (2008).
16) Organization, I.S.: Syntactic metalanguage — Extended BNF (1996). ISO/IEC 14977.
17) Parr, T.J. and Quong, R.W.: ANTLR: A Predicated-LL(k) Parser Generator, *Software Practice and Experience*, Vol.25, pp.789–810 (1994).
18) Redziejowski, R.: Parsing Expression Grammar as a primitive recursive-descent parser with backtracking, *Concurrency Specification and Programming Workshop* (2006).
19) Redziejowski, R.: Parsing Expression Grammar for Java 1.5. http://www.romanredz.se/papers/PEG.Java.1.5.txt
20) javacc: JavaCC Home. https://javacc.dev.java.net/
21) A repository of JavaCC grammars. https://javacc.dev.java.net/servlets/ProjectDocumentList?folderID=110
22) JSON in Java. http://www.json.org/java/

(平成 22 年 9 月 27 日受付)
(平成 22 年 12 月 28 日採録)

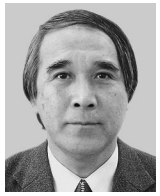


水島　宏太

1983 年生．2006 年筑波大学第三学群情報学類卒業．2008 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入学．プログラミング言語の構文解析に関する研究に従事．

前田　敦司（正会員）

1994 年慶應義塾大学大学院理工学研究科単位取得退学．博士（工学）（慶應義塾大学 1997 年）．1997 年電気通信大学大学院情報システム学研究科助手．2000 年筑波大学電子・情報工学系講師．2004 年筑波大学大学院システム情報工学研究科助教授（2010 年准教授）．プログラミング言語の実装，ガーベッジコレクション，スクリプト言語，パターンマッチング等に興味を持つ．日本ソフトウエア科学会，ACM 各会員．



山口　喜教（正会員）

1972 年東京大学工学部電子工学科卒業．同年通商産業省工業技術院電子技術総合研究所入所．計算機方式研究室長等を経て，1999 年筑波大学電子・情報工学系教授．博士（工学）（東京大学 1993 年）．現在，筑波大学システム情報工学科教授．高級言語計算機，並列計算機アーキテクチャ，並列実時間システム，ネットワーク侵入検知システム等の研究に従事．1991 年情報処理学会論文賞，1995 年市村学術賞受賞．著書『データ駆動型並列計算機』（共著）．IEEE Computer Society，ACM，電子情報通信学会各会員．