Packrat Parsingのメモリ効率の改善手法

○水島 宏太[†] 前田 敦司[†] 山口 喜教[†]

Packrat parsing は，バックトラック付き再帰下降構文解析にメモ化を組合せた構文解析法であり，parsing expression grammar(以下 PEG) で記述できる広範囲の文法を入力サイズの線形時間で解析することができるという特徴を持っている．しかし，packrat parsing には，メモ化を行うために入力サイズに比例する記憶領域を要するという欠点がある．本論文では，カット演算子を PEG に加えることにより，不要な記憶領域を削除する機能を持つ packrat パーザ生成系を提案する．カットは Prolog から借用した概念であり，構文規則中でカットより後ろでバックトラックが不要な場合にそのことを処理系に指示し，不必要なバックトラックが起こらないようにするための機能である．カットの評価によって，ある入力位置より前にバックトラックしないということが判明した場合，その入力位置より前にバックトラックしたときのために使用されているメモ化領域を動的に削除し，再利用することができる．我々は，構文規則中にカットを適切に挿入することで，多くの実用的文法について有界なメモリ量で解析できるパーザを生成可能と考えている．一方，カットによるメモリ効率の改善は，パーザ生成系のユーザが手動でカットを挿入しなければいけないという問題点があるが，本論文ではこの問題点を解決するために，解析する言語の構文の意味を変えない範囲でカットを自動的に挿入する手法と，不要なメモ化が行われている箇所を検出して必要な記憶領域を静的に削減するための手法についても提案する．

Improvement technique of memory efficiency of Packrat Parsing

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

Packrat parsing is a parsing technique that combines memoization with backtracking recursive descent parser. It can handle any grammars that can be expressed in powerful grammar notation called parsing expression grammar (PEG). Generated parsers can analyze its input in linear time. However, to memoize intermediate result, packrat parsing requires storage area proportional to the input size. In this paper, we propose the packrat parser generator system that disposes unnecessary storage area by adding the notion of a cut operator to PEG. Cut operators is the notion we borrowed from Prolog that can be used by programmers to 'cut off' an alternative execution branch of a choice point in a syntax rule when the alternative should not be tried for suppressing unnecessary backtracking. When an alternative is removed by execution of a cut operator (and thus the parser can make sure that no backtracking can occur before a particular input positon), the memoization storage kept against backtracking can be deallocated and reclaimed dynamically. We believe that, for most pratical grammars, parsers which only use bounded memory size can be generated by appropriate insertion of cut operators in syntax rules. Although memory efficiency that can be achieved by hand-insertion of cut operators would be valuable, it is awesome and error-prone. In this paper, we also propose a technique to reduce required storage area by statically detecting syntax rules which memoization is unnecessary and another technique for automatically inserting cut operators without changing meanings of syntax rules.

1. はじめに

近年，Web アプリケーションの開発をはじめとする分野で，Ruby などの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急速に普及してきている．スクリプト言語は C や Java といった言語と比べて，多くの場合文法が複雑であり，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 な手法を取ることで対応しているが，そのために文法の記述が煩雑になり，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，その入力ファイルのサイズは 6000 行以上

† 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

におよぶ (Ruby1.8.6)．また，スクリプト言語には式を埋め込める文字列など，字句要素に構文要素を埋め込む機能を持っているものが多いが，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 ad hoc な手法を使わなければ解析を行うことができない．

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] という形式文法をベースにしており，決定的な任意の LR(k) 言語および，一部の文脈依存言語を入力の長さに対して線形時間で解析することができる．Packrat parsing を使えば，yacc が生成する LALR(1) パーザよりも広範囲の言語を解析することができる．また，packrat parsing のアルゴリズムは字句解析を前提としないアルゴリズムであるため，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，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取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 には二つの問題点が存在する．一つ目の問題点は，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点である．この点は，プログラムが機械的に生成した巨大なソースコードなどを解析する場合に問題になる．Ford の packrat parsing の論文[2] では，大規模な 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．二つ目の問題点は，packrat parsing はバックトラックやメモ化を行うために，LALR(1) などに比べて一般的に実行性能が悪いという点である．この点も，同様に大規模なファイルの解析を行う場合に問題になる．

3. Parsing Expression Grammar

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 parsing expression という式を表す．以下は PEG の式の主要な構成要素である．

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

ここで，BNF の | と違って / 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無く，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り返しと異なり，一度繰り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，例えば $e_1 *\ e_2$ 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り返しをやり直すことは無く，$e_1 *\ e_2$ 全体が失敗するということである．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り返し演算子と呼ばれる演算子の動作に類似している．& は先読みを行う演算子であり，式 e にマッチするかを試して，マッチした場合に成功するが，入力を消費しない．! は否定先読みを行う演算子であり，e にマッチするかを試して，マッチしなかった場合に成功するが，入力を消費しない．

4. 提 案 手 法

4.1 カット演算子

本研究では，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) の導入を提案する．カットは Prolog[4] から借用した概念であり，本研究では $e_1 \uparrow e_2 \,/\, e_3$ という形で，/ の左辺に挿入できる演算子として定義する．カットは次のように動作する．まず最初に，e_1 のマッチを試行する．マッチに成功した場合，次に e_2 のマッチを試行する．e_1 のマッチに失敗した場合，e_1 のマッチの開始位置にバックトラックして，e_3 のマッチを試行する．$e_1\ e_2 \,/\, e3$ との違いは，e_2 のマッチに失敗した場合に，バックトラックして e_3 のマッチを試さずに式全体が失敗することである．つまり，カットによってバックトラックを抑制することができるといえる．

ここで，カット (図の中では ↑ 記号で表記) を使って定義した次の算術式の PEG に入力`"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．

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

まず，入力`"a+"`が与えられると，最初は上から E(`"a+"`)，P(`"a+"`) の順番で構文規則が呼び出され，P(`"a+"`) の中で`"a"`にマッチするため，マッチに成功したという情報と残りの文字列`"+"`が E に返される．E の中では，P が成功したため，それに続く`"+"`に対して`"+"`のマッチを試みる．このマッチも成功し，成功したという情報と残りの文字列`""`が得られる．マッチが成功したため，それに続いて E(`""`) を呼び出されるが，これは失敗する．ここで通常ならば，E 中の次の選択肢 P `"-"` $\uparrow$ E を試すことになるが，マッチした`"+"`の直後に ↑ が挿入されているため，他の選択肢を試さずに失敗したという情報を返す．

カットを導入することで，2 節で挙げた二つの問題点を改善できる可能性がある．まず第一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．カットを通過した際にはここでの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 番目について，メモ化のための記憶領域を除去できる．次に第二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，また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ーヘッドを減らすことができると考えられる．

4.2 カットの定性的な性質

前節で，カットによってメモリ効率を改善できるという事を述べたが，どの程度改善できるのかについては述べなかった．この節では，カットによるメモリ効率の改善が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，E 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 `"+"` または 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常に 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，メモ化に必要な領域の空間計算量は $O(1)$ 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，メモ化に必要な領域の空間計算量は $O(s)$ になる．

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

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無くなるので，メモ化に必要な領域の空間計算量は最初のものと同様に $O(1)$ になる．

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

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，通常はサイズが一定以上の大きさにならない要素の長さによって必要なメモ化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と考えている．なお，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．

4.3 不要なメモ化の除去

本研究では，不要なメモ化を除去するための手法についても提案する．Packrat parsing では全ての非終端記号に対してメモ化を行うが，文法によっては静的にある非終端記号のメモ化が不要であると判定することができる場合がある．例えば，以下の PEG(開始記号は N_0 とする) について考えると，N_0 の解析中に N_2 および N_3 の解析が失敗した場合はバックトラックが発生せず，N_0 の解析そのものが失敗するため，N_2 および N_3 の結果をメモ化する必要は無いといえる．

$$
\begin{array}{lcl}
N_0 & \leftarrow & N_1 \uparrow N_2 \ / \ N_3 \\
N_1 & \leftarrow & \texttt{"a"} \\
N_2 & \leftarrow & \texttt{"b"} \\
N_3 & \leftarrow & \texttt{"c"}
\end{array}
$$

メモ化が不要な非終端記号を完全に検出することは困難であると考えられるため，判定は保守的に行う．大まかな方針としては，ある非終端記号について，それが開始記号から直接・間接的に参照されていないか，参照されている全ての文脈においてバックトラックせず，呼び出し元に失敗を返すような場合に，それをメモ化が不要であると判定する．具体的には，ある非終端記号 N について，N の中で直接参照されている非終端記号の集合 A_N の中で，失敗時にバックトラックせず，N 自体の解析が失敗するような非終端記号の集合 B_N を以下のようにして求めることができる (e_N は非終端記号 N が持つ式を表す)．

$$
\begin{array}{lcl}
B_N & = & B(e_N) \\
B(e_1 \uparrow e_2 \ / \ e_3) & = & B(e_2) \cup B(e_3) \\
B(e_1 \ / \ e_2) & = & B(e_2) \\
B(e_1 \ e_2) & = & B(e_1) \cup B(e_2) \\
B(e*) & = & \emptyset \\
B(N) & = & \{N\} \\
B(\texttt{"}s\texttt{"}) & = & \emptyset \\
B(\varepsilon) & = & \emptyset
\end{array}
$$

ここから，A_N の中で失敗時にバックトラックする非終端記号の集合 C_N を $A_N - B_N$ と定義することができる．C_N の閉包 C_N^* と非終端記号の集合 V，開始記号 S に対して，$V - C_S^*$ を求めることで，明らかにメモ化が不要な非終端記号の集合を求めることができる．

5. 関連研究との比較

2 節で述べた 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[5]．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!* では処理系によるいくつかの自動的な最適化と，ユーザによる明示的な最適化を行うことができるようになっており，LALR(1) パーザ生成系などと比べても，実行性能および記憶領域の両方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．

この節では，*Rats!*が提案している最適化と，本研究による最適化がどのような関係を持つかについて考察を行う．なお，最適化の名称は文献 5) のものをそのまま使用している．

5.1 Transient

Transient 最適化は，構文規則に対して人手で transient という修飾子を付加することで，対象となる構文規則の解析結果をメモ化しないようにする最適化である．この最適化によって，メモ化のためのテーブルのカラムからフィールドを削除できるため，メモリ効率を改善することができる．しかし，メモ化した結果が再利用される構文規則に対して付加すると，実行効率が悪化する危険性がある．場合によっては，解析にかかる時間計算量のオーダが $O(n)$ より悪化する可能性もある (このことは文献 5) においても述べられている)．一方，カットによるメモ化領域の除去は，メモ化した結果が決して再利用されないことが判明した領域に対してのみ行われるため，時間計算量のオーダは変化しない．また，その性質からわかるように transient 最適化によるメモリ効率の改善は係数レベルの改善であり，空間計算量のオーダは変化しない．それに対してカットは適切に挿入することによって空間計算量のオーダを改善することができる．Transient 最適化はカットと直交しており，併用することが可能である．

5.2 Nontransient

Nontransient は，transient が指定されていない構文規則に対して，ある条件を満たす場合に処理系が自動的に transient を付加する最適化である．この最適化の利点は，人手で transient 修飾子を付加する手間が必要無いことである．処理系が自動的にメモ化しない規則を決定するという点で本研究の不要なメモ化の除去と類似している．どのような構文規則を transient をみなすかについては，現時点では他の構文規則からの参照が 1 箇所しか存在しない場合にのみ transient

とみなすということが文献 5) において述べられている．そのため，どのような構文規則を transient とみなすかという点で本研究における不要なメモ化の除去手法とは異なると言える．Nontransient 最適化もカットと直交しており，併用することが可能である．

5.3 Chunks

Chunks 最適化はメモ化に使用するテーブルのカラムを複数のチャンクと呼ばれる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アロケーションを削減するものである．文献 5) において，chunks はヒープ利用率の改善のために最も重要な最適化であると述べられている．Transient と同様にこの最適化も係数レベルでメモリ効率を改善するものであり，空間計算量のオーダは変化しない．Chunks 最適化もカットと直交するものであり，併用することが可能である．

5.4 Repeated

Repeated 最適化は transient な構文規則中に現れる，繰り返し演算子を使った繰り返しに関して，それを右再帰に展開せず，単なるループとして処理する最適化である．この最適化によって，関数呼び出しの回数を減らせるため，実行性能を向上させることができる上に，繰り返しを右再帰に展開しないことによってメモ化のための領域を減らすことができる．また，右再帰に展開する場合と違って処理系のスタックを消費しないため，スタックオーバーフローが起こるのを抑制することができる．Repeated 最適化も係数レベルでメモリ効率を改善するものであり，空間計算量のオーダは変化しない．上で述べた他の最適化同様に，repeated 最適化もカットと直交するものであり，併用することが可能である．なお，repeated 最適化には厳密な根拠があるわけではなく，文法定義によっては時間計算量が悪化する危険性がある．例えば，以下のような文法定義を考える．

$$A \leftarrow (\&((a\ /\ b)*)\ (a\ /\ b))*$$

ここで，構文規則 A はメモ化する必要が無いため，transient とみなすことができる (*Rats!*でも nontransient 最適化によって transient とみなされる)．そのため，repeated 最適化の対象となる．しかし，A の中に現れる二つの繰り返しの両方を単なるループに展開してしまうと，一文字読むごとに残りの文字列についてマッチし続ける限り先読みを行うため，入力文字列の長さ n に対して時間計算量が $O(n^2)$ になってしまう☆.

5.5 考　　察

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，定数とみなすことができる) と入力長 n に対して，一般にサイズ mn のテーブルが必要である．この節で述べた *Rats!*のメモリ効率に関する最適化は，いずれも空間計算量を係数レベルで改善するものであり，そのため，通常の packrat parser と同様に空間計算量は $O(n)$ のままである．このことは，*Rats!*においても大規模なファイルの構文解析は向いていないということを示している．4.2 節で述べたように，カットは空間計算量をオーダのレベルで改善できるため，カットを適切に挿入することで，大規模なファイルの構文解析に適用できる可能性があるという点が，*Rats!*の最適化手法と比較した場合の提案手法の利点である．また，これらの最適化はいずれもカットと直交するものであるため，カットと併用することでさらにメモリ効率を改善できる可能性がある．

6. 実　　験

カットがメモリ効率および実行効率にどのような影響を与えるのかを検証するための実験を行った．まず実験のために，カットの機能および不要なメモ化の除去機能を持つ Java による packtrat parser 生成系 *Yapp* を開発した (カットの自動挿入については未実装)．実験 1 と 2 では実装した *Yapp* に Java 言語 (Java 1.4) および XML のサブセットの文法定義を与えて，カットを挿入しない場合と挿入した場合のパーザを生成し (不要なメモ化の除去は未使用)，それらのパーザと *Rats!*(バージョン 1.11.0) から生成されたパーザで，実行効率およびメモリ効率を測定した．実験 3 では Java 言語および XML のサブセットの文法定義に対して，提案した不要なメモ化の除去手法がどの程度効果があるのかを調べた．なお，Java の文法定義については，*Rats!*版は処理系に付属の Java 1.4 のパーザをそのまま使用し，*Yapp* 版については文献 6) の著者の PEG for Java 1.5 を元に，Java 1.5 で追加された文法を取り除いたものになっている．XML の文法定義については，*Rats!*による XML の文法定

☆ 現在の Yapp の実装でも，長い繰り返しでスタックオーバーフローが発生するのを防ぐために繰り返しをループに展開しているため，同様の危険性がある．ただし，今回の実験においては文法定義が上のような形にはなっていないため，問題にはならない．この問題を解決するには，ループに展開しつつ，繰り返しの解析結果を適切にメモ化するように実装する必要がある．

義が存在しなかったため，*Rats!*版と *Yapp* 版の両方とも，Extensible Markup Language (XML) 1.0[7] を参考に，実験用の入力データを解析できる最小限の機能のみを実装したサブセットになっている．なお，*Rats!* 版の XML パーザについては，transient にしても性能が低下しないと考えられる規則については手動で transient を付加してある (構文規則 24 個中 6 個に付加)．また，*Rats!*版の Java パーザおよび XML パーザは，*Rats!*の最適化オプションは特に指定していない．最適化オプションを指定しない場合，実験に使用したバージョンの *Rats!*では文献 5) の Table 3. に書かれている最適化の全てが有効な状態になる☆ため，5 節で論じた 4 個の最適化についても有効になっている．実験のために使用した XML および Java パーザは，*Rats!*版と *Yapp* 版のどちらも，入力文字列が文法に合致するかどうかを検査するだけのもの (recognizer) であり，抽象構文木の構築は行っていない．

これらの文法定義の性質について，次のような事が挙げられる．まず，Java の文法定義は構文規則の数が多く (183 個) 複雑な文法である．また，カット有りの Java の文法では，一つの文あるいは式の最大の長さに比例したヒープサイズが必要になるようにカットを挿入した．一方，XML のサブセットの文法定義は構文規則の数が少なく (24 個)，比較的単純な文法である．また，カット有りの XML の文法では，コメントまたは要素中のテキストの最大長に比例したヒープサイズが必要になるようにカットを挿入した．参考のために，以下にカットを挿入した Java の PEG の一部 (**図 1**) と XML のサブセットの PEG の一部 (**図 2**) 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表現しており，構文規則 Element は XML の要素を表現している．

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

図 1 Java の PEG の一部

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

図 2 XML の PEG の一部

実験のためにパーザに与える入力としては，Java パーザについては *Rats!*の処理系に付属のベンチマーク用 Java プログラム (41 ファイル) を使用した．ベンチマーク用の Java プログラムは，Java のソースコードジェネレータや暗号化アルゴリズム (Blowfish など) の実装などで構成されている．一方，XML パーザについては，スロベニア語と英語の並列コーパスである，IJS-ELAN corpus Version 2.0[8] のデータ 48 ファイルからサイズが 2MB 未満である 38 ファイルを抽出して使用した．使用した IJS-ELAN corpus のデータはファイルサイズが大きい (1MB 以上のものが存在する) 現実に有り得る入力であるため，巨大な XML に対するパーザの性能を評価するために適切なデータであると言うことができる．

- 実験環境
  - CPU: Intel Core2 Duo 2.4GHz
  - RAM: 2.0GB
  - OS: Windows XP Professional
  - Java 実行環境: JDK1.6.0

6.1 実　験　1

カットによってメモリ効率がどれだけ改善されるかを確かめるために，生成されたパーザがあるサイズのファイルを解析するのにどれだけのヒープを使用するのかを計測した．パーザが必要とする最小のヒープサイズを求めるために，JVM の起動オプションである，`-Xms` と`-Xmx` を使用した．これらのオプションはそれぞれ，JVM が使用する最小ヒープサイズと最大ヒープサイズを指定するものである．このオプションを使用して JVM が使用するヒープサイズを指定できる．例えば，`java -Xms64m -Xmx64m Main` とすると，Main の実行時に JVM が使用するヒープサイズを 64MB に指定してプログラムを実行することになる．これを利用して，二分探索法によって各ファイルに対して正常に構文解析を終了することができる最小のヒープサイズを求めた．実験の結果は**図 3** と**図 4** のようになった．ここで，グラフの横軸は構文解析の対象である各ファイルのサイズ (KB) を，縦軸はファイルを解析するのに必要な最小のヒープサイズ (MB) を表しており，個々の点が各ファイルに対応している．

☆ 文献 5) の Table 3. に未出の最適化である-Oerrors2 および-Oleft1 は無効になっている．

なお，実験 1 と 2 ともに，1KB=2^{10}B，1MB=2^{20}B の意味である．なお，`-Xms` オプションで指定できる最小のヒープサイズが 2MB であるため，この実験では，パーザが必要とする最小のヒープサイズが 2MB 未満の場合の違いは検出できない．

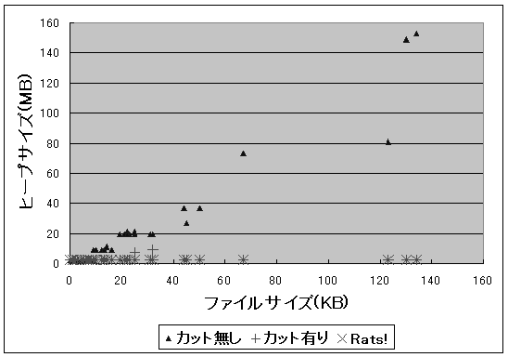


図 3 Java プログラムの構文解析に必要な最小ヒープサイズ

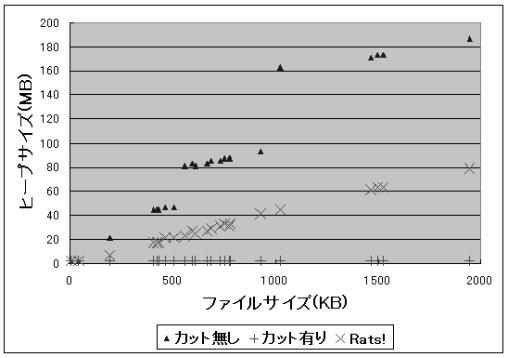


図 4 XML ファイルの構文解析に必要な最小ヒープサイズ

まず，Java プログラムの構文解析については，カット無しに比べてカット有りの場合の方がメモリ効率が改善しているものの，*Rats!*と比べた場合は，カット有りでも有意な差が見られず，両方とも入力のサイズに対してほぼ $O(1)$ の空間計算量であるかのように見える．これは，Java パーザの実験に使用したファイルのサイズが大きいものでも 200KB に満たないものであることと，5 節で述べたように *Rats!*が空間計算量を係数レベルで改善するための様々な最適化を行っているために，JVM の `-Xms` オプションで指定できる最小のヒープサイズである 2MB でも構文解析を行うことができたことが原因であると考えられる．また，サイズが 20KB から 40KB の間に，*Rats!*に比べてカット有りの方が明らかにヒープ使用量が多いファイルが 2 つ存在するが，これらのファイルについて調べてみたところ，共通する特徴として，一つの式や文が非常に巨大であることがわかった．今回実験に使用した PEG ではカットを十分に挿入し切れていないため，一つの文や式の長さが巨大である場合，その式や文の解析を終えるまでメモ化領域を削除することができないことが影響したものと考えられる．

一方，XML ファイルの構文解析については，カット無しと *Rats!*のどちらと比べても，カット有りの方が大幅にメモリ効率が改善されており，入力長が増えても必要なヒープサイズはほぼ変わっていない．これは，カット有りの場合では空間計算量は XML 中のテキストやコメントの最大長によって決まる一方で，入力に使用したファイルの性質として XML の要素の数は多いものの，テキストやコメント一つ辺りの長さは短いために入力のサイズが増大しても必要なヒープ使用量が増大せず，カット有りのパーザの空間計算量が実質的に $O(1)$ であったことと，入力のサイズが大きい (1MB 以上) ために，空間計算量の差が明確にあらわれたためであると言える．

これらの結果から，カットを適切に挿入することによって，実用的な入力に対して空間計算量がほぼ $O(1)$ であるようなパーザを生成することが可能であり，生成したパーザを大規模な入力に適用することが可能である事がわかる．

6.2 実　験　2

カットによって実行効率がどのように改善されるかを調べるために，生成されたパーザの構文解析速度が，設定したヒープサイズによってどのように変化するかを計測した．ヒープサイズの指定には，実験 1 と同じように JVM の起動オプション`-Xms` と`-Xmx` を使用した．例えば，`java -Xms5m -Xmx5m ...` とすることによって，ヒープサイズを 5MB に指定して構文解析を行った．構文解析の速度は，上記で述べたベンチマーク用のデータの全ファイルを解析するのにかかった時間とファイルの合計サイズから求めた (10 回計測した平均値を使用)．実験の結果は**図 5** と**図 6** のようになった．なお，グラフの横軸は構文解析時に指定したヒープサイズ (`-Xms` と`-Xmx` により指定) を，縦軸はそのときの構文解析の速度を表している．また，縦軸の値が 0 であるデータは，対応するヒープサイズにおいて構文解析を完了することができなかったことを表している．

実験の結果からはまず，カットを挿入することによって構文解析の速度が大幅に改善していることがわかる．これは，必要とするヒープサイズが小さくなったことによって，ガベージコレクションのオーバヘッドが削

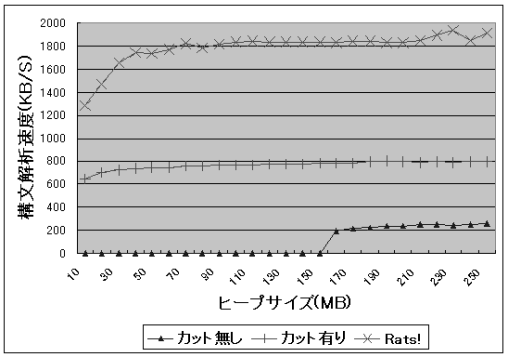


図 5 Java プログラムの構文解析の速度

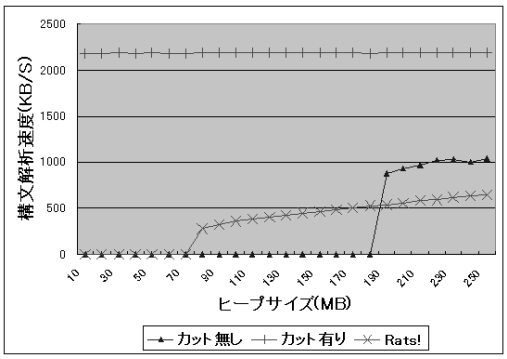


図 6 XML ファイルの構文解析の速度

減されたことやデータがキャッシュにヒットしやすくなったことなどによると考えられる．また，Java プログラムの解析では *Rats!*の方が 2〜3 倍程度高速であるのに対し，XML ファイルの解析では *Yapp* の方が 3 倍程度高速であることがわかる．原因についてはいくつか考えられるが，まず，Java プログラムの文法定義において *Rats!*版は処理系に付属のものをそのまま使用したために，*Rats!*において性能が出るような文法定義であったのが，XML の文法定義では同じ PEG を元にして実装したため，*Rats!*で性能が出るような文法定義になっていなかったという事が考えられる．別の原因としては，メモ化領域のデータ構造が *Rats!*と *Yapp* とで異なるため，それによって性能が変化したということも考えられる．また，これらの結果から，カット有りのパーザが様々な最適化を行っている *Rats!*に比べても極端に遅いわけではなく，十分実用に耐え得ることがわかる．

6.3 実　験　3

4.3 節で提案した不要なメモ化の除去手法を Java の PEG(カット無し, あり) および XML のサブセットの PEG に適用し，どれだけの非終端記号のメモ化が不要になったかを調査した．実験の結果は，**表 1**(Java) および**表 2**(XML) のようになった．

表 1 Java の PEG においてメモ化される非終端記号の数

	適用前	適用後
カット無し	183	176
カット有り	183	176

表 2 XML の PEG においてメモ化される非終端記号の数

	適用前	適用後
カット無し	24	21
カット有り	24	18

この結果から，カットを挿入することによって，不要なメモ化の除去手法の効果を改善できる可能性があること，どちらの PEG においても 10 個未満の少数の非終端記号のメモ化が除去可能になっていることがわかる．他の言語の文法定義についても同じ程度の数の非終端記号しか除去できないとすると，この手法の効果は，文法の規模が大きくなり複雑になるほど相対的に小さくなると言える．そのため，ある程度以上の規模の文法定義にこの手法だけを適用してもあまり効果は見込めないと考えられる．

7. 提案手法の改良案

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．例えば，4.1 節の算術式の PEG のカットを挿入する位置だけを変えた以下の文法定義について考える．

$$E \leftarrow P \texttt{"+"} \uparrow E \mid P \texttt{"-"} \uparrow E \mid P$$
$$P \leftarrow \uparrow \texttt{"a"} \mid \texttt{"b"} \mid \texttt{"c"}$$

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",...` という形以外の式以外を解析できなくなってしまう．本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，このような文法の意味を変えるようなカットは意図していないし，望ましく無いと考える．

この節では，上記のカットの問題点を改善するために，カットを自動挿入するための手法について考察を行う．文法の意味が変わらない範囲でカットを処理系が自動挿入できれば，カットを手動で挿入する危険無しに，同等の効率を達成できる可能性がある．これは，Γ_a,Γ_b,Δ を任意の PEG の式として，カットを $\Gamma_a\Gamma_b/\Delta$

という形の PEG の式に対して挿入して，$\Gamma_a \uparrow \Gamma_b/\Delta$ としても意味が変わらないかどうかを判定する問題と考えることができる．カットを挿入しても意味が変わらないことを判定するには，Γ_a と Δ の受理する言語が disjoint であるかどうか，つまり以下の式が満たされているかを判定する必要がある．なお，PEG が受理する言語 (PEL) の定義は，与えられた入力文字列に対して式が成功することであり，全ての入力を消費する必要は無い (例えば ε という PEG の式が受理する言語は任意の文字列を含む集合になる) ことが文献 3) で述べられているため，判定すべき条件はこの式で良いと言える．しかし，この問題 (PEL の disjointness の判定) は決定不能問題であることが示されている[3]．なお，$L(\Gamma_a)$ は Γ_a を受理する言語 (文字列の集合) を表している．

$$L(\Gamma_a) \cap L(\Delta) = \emptyset$$

そのため，プログラムによってカットを自動挿入可能かを判定するには，より保守的な判定を用いる必要がある．具体的には，Δ を分割した Δ_a と Δ_b があったとして ($\Delta = \Delta_a\Delta_b$)，式 $\Gamma_a\Gamma_b/\Delta_a$ にカットを挿入して $\Gamma_a \uparrow \Gamma_b/\Delta_a$ としても意味が変わらないならば，同様に式 $\Gamma_a\Gamma_b/\Delta$ にカットを挿入して $\Gamma_a \uparrow \Gamma_b/\Delta$ としても明らかに意味が変わらないことと，Γ_a と Γ_b が正規言語であるとき，問題を正規言語の disjointness を判定する問題に変換できることを利用する (正規言語の disjointness については，それを判定する手法が存在する)[9]．まず，式が次の形をしているとする．ここで，Δ_a の受理する言語が正規言語になるように Δ を分割する．また，ある PEG の受理する言語が正規言語であるかどうかの判定が一般に決定可能であるかどうかは不明であるため，明らかに Γ_a と Δ_a の受理する言語が正規言語になるように分割する (例えば，式`"if"`の受理する言語は明らかに正規言語である)．

$\Gamma_a\Gamma_b/\Delta_a\Delta_b$($\Gamma_a$ Δ_aは正規言語を受理する式)

Γ_a と Δ_a はともに正規言語を受理する式であるため，次の式が満たされるかどうかを判定できる．

$L_R(R(\Gamma_a)) \cap L_R(R(\Delta_a)) = \emptyset$
$R(e)$ ： $L(e)$ を包含する言語を表す正規表現
$L_R(r)$ ： 正規表現 r が受理する言語

$L_R(R(e))$ は $L(e)$ を包含しているため，上の式を満たしているならば次の式も満たしていると言える．

$$L(\Gamma_a) \cap L(\Delta_a) = \emptyset$$

この式が満たされているということは，式 $\Gamma_a\Gamma_b/\Delta_a$ にカットを挿入して $\Gamma_a \uparrow \Gamma_b/\Delta_a$ としても意味が変わらないということであるため，式 $\Gamma_a\Gamma_b/\Delta$ にカットを挿入して，$\Gamma_a \uparrow \Gamma_b/\Delta$ としても意味が変わらないと言うことができる．

ただし，この自動挿入手法には，ある式の 0 回以上の繰り返しに適切にカットを挿入できないという問題点が存在する．PEG においては，ある式 e の 0 回以上の繰り返しを，$E = e\ E\ /\ \varepsilon$ という形で表現するが ($e*$ はこれと意味的に同じである)，このとき，ε という必ずマッチする式が / の右辺にあるため，E も必ずマッチする式になる．そのため，人手でカットを挿入する場合は E の直前にカットを挿入して $E \leftarrow e \uparrow E\ /\ \varepsilon$ としても意味が変わらないことがわかる．一方，自動挿入手法を適用することを考えた場合，/ の右辺の式 ε が受理する言語は，任意の文字列を含む集合であり，$L_R(R(\varepsilon))$ も同様に任意の文字列を含む集合になるため，カットを挿入することができない (厳密に言えば，e または e を分割したプレフィクスの受理する言語が空集合である場合にはカットを挿入できる可能性があるが，通常は，受理する言語が空集合になる式を記述する意味は無い)．プログラミング言語などの文法において，ある要素の 0 回以上の繰り返しは頻出するため，カットを自動挿入する別のアルゴリズムを組み合わせるなどして，この問題点を解決すれば，自動挿入手法の実用性は高まると考えられる．例えば，5.4 節では repeated 最適化には時間計算量が悪化する場合がある事を示したが，繰り返しにカットを自動的に挿入することができれば，そのカットによってメモ化領域を削除できる場合に，repeated 最適化と同様の効果を時間計算量が悪化する危険性無しに達成できる可能性がある．

8. ま と め

本研究では，packrat parsing の改善手法として，カットの導入を提案した．カットをユーザが適切に用いることによって，packrat parsing による構文解析の実行性能の向上や必要な記憶領域の削減の効果が得られることを，実験によって示すことができた．特に，提案手法によって，今まで packrat parsing が苦手であるとされていた大規模な XML ファイルの構文解析を効率良く行える事を示せたのは大きいと考えている．

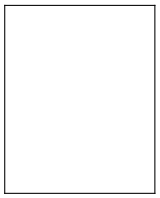
今後の課題としては，まず，提案したカット自動挿入手法の改良が挙げられる．7 節で言及したように，現状の手法だけでは一般的なプログラミング言語の文法に対して十分ではないと考えられるためである．次

に，カット自動挿入手法を実装し，*Rats!*などとの性能比較を行い，カットの自動挿入がどの程度有効であるかを確かめる必要がある．また，今回提案した不要なメモ化の除去手法は，あまり効果が見られなかったため，より多くの不要なメモ化を除去できる手法を考えたい．

参 考 文 献

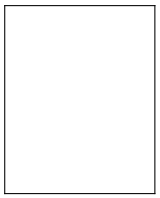
1) Johnson, S.: Yacc: Yet Another Compiler Compiler, *UNIX Programmer's Manual*, Holt, Rinehart, and Winston, New York, NY, USA, pp.353–387 (1979).
2) Ford, B.: Packrat Parsing: Simple, Powerful, Lazy, Linear Time, *Proceedings of the 2002 International Conference on Functional Programming* (2002).
3) Ford, B.: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation, *Symposium on Principles of Programming Languages* (2004).
4) Colmerauer, A. and Roussel, P.: The birth of Prolog, *The second ACM SIGPLAN conference on History of programming languages*, ACM Press, pp.37–52 (1993).
5) Grimm, R.: Better Extensibility through Modular Syntax, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pp. 19–28 (2006).
6) Redziejowski, R.: Parsing Expression Grammar as a primitive recursive-descent parser with backtracking, *Concurrency Specification and Programming Workshop* (2006).
7) Bray, T., Paoli, J. and Sperberg-McQueen, C.: *Extensible Markup Language (XML) 1.0 (Fourth Edition),*
W3C Recommendation (2006).
8) Erjavec, T.: The IJS-ELAN Slovene-English Parallel Corpus, *International Journal of Corpus Linguistics*, Vol.7, No.1, pp.1–20 (2002).
9) Hopcroft, J. and Ullman, J.: *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley (1979). (訳：オートマトン言語理論計算論 I，サイエンス社 (1984)).

(平成？年？月？日受付)
(平成？年？月？日採録)



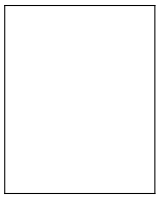
水島　宏太

昭和 58 年生．平成 18 年筑波大学第三学群情報学類卒業．同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前期課程入学．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 (2007 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などに興味を持つ．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などを経て，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などの研究に従事．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)」．IEEEComputer Society, ACM，電子情報通信学会各会員．