情報処理学会論文誌　プログラミング　Vol. 4　No. 3　84–93 (June 2011)

更新検知を用いて左再帰に対応する Packrat Parser の実装

後藤 勇太[†1]　白田 佳章[†2]
木山 真人[†2]　芦原 評[†2]

構文解析法で Packrat Parsing という手法がある．Packrat Parsing は，再帰下降構文解析にメモ化を組み合わせた手法であり，バックトラックや無限先読みを用いた解析において，線形時間で解析可能である．しかし，左再帰を含む文法は解析不可能である．そこで，従来は左再帰を含む文法を解析する際，左再帰部分を等価な右再帰に変換し，解析を行っていた．だが文法の変換を行うと構文木の構造が変化してしまう．また，特定の左再帰は変換できない．たとえば，閉路が存在する文法である．よって，この手法では解析できない文法がある．Warth らは，左再帰を含む文法を，右再帰への変換なしに解析を可能にした．しかし，Warth らの手法では，同一の入力位置で左再帰が複数発生する文法において，特定の入力の解析に失敗する．また，構文木の構造が意図したものとは異なるという問題がある．そこで本研究では，更新検知を用いて，左再帰を含む文法を右再帰への変換なしに解析でき，かつ従来手法の 2 つの問題点に対応する Packrat Parser を提案・実装し，評価を行った．

Implementation of Packrat Parser with Update Detection to Parse Left Recursive Grammars

Yuta Goto,[†1] Yoshiaki Shirata,[†2] Masato Kiyama[†2] and Hyo Ashihara[†2]

Packrat Parsing is a kind of parsing method. Packrat Parsing is a combination of Recursive Descent Parsing and memoization that can parse backtracking and unlimited look-ahead in linear parse time. However, Packrat Parsing cannot parse left recursive grammars. Thus, traditional method transforms left recursive grammars into right recursive grammars. Unfortunately, syntax tree is changed by the transforming. Moreover, particular left recursive grammars cannot be transformed. Traditional method cannot parse particular grammars. Warth, et al. made possible to support left recursive grammars without transforming in Packrat Parsing. However, the method cannot parse some grammars that have multiple left recursions at an input position. Furthermore, syntax tree is be unintended consequence when the method parses particular grammars. This paper presents impremention and evaluation of Packrat Parser with Update Detection that possible to support left recursive grammars without transforming, and grammars that have multiple left recursions at an input position.

1. はじめに

構文解析法に Packrat Parsing [1] という手法がある．Packrat Parsing は，再帰下降構文解析にメモ化を組み合わせた手法である．バックトラックや無限先読みを用いた Parsing Expression Grammar（PEG）[2] の解析において線形時間で解析可能という利点がある．一方，左再帰を含む文法を解析できないという欠点がある．左再帰とは，文法においてある非終端記号からその非終端記号自身を左端に含む状態を指す．左再帰は，直接左再帰と間接左再帰の 2 種類に分類される．直接左再帰は，ある非終端記号が直接その非終端記号自身を呼び出す状態である．たとえば以下のような文法は直接左再帰を含む．

$$S \leftarrow Sb/a \tag{1}$$

一方，間接左再帰は，ある非終端記号が複数の非終端記号を経由して，間接的に自分自身を呼び出す状態である．以後，説明の簡略化のため，間接左再帰において，自分自身を呼び出す非終端記号を head rule，経由される非終端記号を involved rule と呼称する．以下のような文法は間接左再帰を含む．

$$\begin{aligned} S &\leftarrow A \\ A &\leftarrow Sb/a \end{aligned} \tag{2}$$

左再帰を含む文法を通常の再帰下降構文解析法で解析すると，無限再帰に陥ってしまう．左再帰に対応するため，Pappy [3] や Rats! [4] などの手法では，左再帰を含む文法をほぼ等価な右再帰を含む文法に変換して解析を行う．たとえば，(1) の文法を右再帰を含む文法に変換すると以下のようになる．

†1 熊本大学工学部情報電気電子工学科
Faculty of Engineering, Department of Computer Science and Electrical Engineering, Kumamoto University
†2 熊本大学大学院自然科学研究科
Graduate School of Science and Technology, Kumamoto University

$$S \leftarrow aS'$$
$$S' \leftarrow bS'/b \qquad (3)$$

このように変換すると左再帰を含む文法を解析できる．しかし，文法を変換すると構文木の形が変化するという問題がある．Warth ら[5] は，左再帰を含む文法を，右再帰への変換なしに解析できる Packrat Parser を実装した．Warth らの手法は，文法の変換を行わないため，従来の構文木の形が変化するという問題は解決している．しかし，この手法では，同一の入力位置で head rule が異なる左再帰を含む文法が解析できない．さらに特定の文法と入力で，意図した構文木が得られないという問題がある．

そこで本稿では，同一の入力位置で head rule が異なる左再帰を含む文法を解析でき，かつ意図した構文木が得られる手法について提案し，評価を行う．2 章では，Warth らの手法とその問題点について述べる．3 章では，提案手法について述べる．4 章では，提案手法の評価および考察を述べる．最後に，5 章で本稿のまとめを述べる．

2. 従 来 手 法

本章では，Warth らが提案した従来手法とその問題点について述べる．

2.1 従来手法のアルゴリズム

まず，従来手法のアルゴリズムについて述べる．従来手法では，左再帰が発生した際，発生した左再帰の head rule の解析を保留し，次の選択肢に移り解析を進める．その後，左再帰以外の解析結果を得るとメモにその情報を書き込み，その解析結果を用いて保留しておいた左再帰の解析を進める．

従来手法のアルゴリズムの擬似コードを文献 5) より再録し，簡潔に説明する．このアルゴリズムは大きく分けて APPLY-RULE プロシージャ，SETUP-LR プロシージャ，LR-ANSWER プロシージャ，GROW-LR プロシージャ，RECALL プロシージャの 5 つのプロシージャからなる．

APPLY-RULE プロシージャを**図 1** に示す．APPLY-RULE プロシージャは rule の適用に用いる．適用する rule の解析結果がメモにあれば，メモからその情報を取得する．メモになければ，その rule を LR データを用いて LRStack にプッシュする．LRStack は LR データが格納されるスタックであり，左再帰の情報を得るために用いる．LR データのデータ型を以下に示す．

LR : (*seed* : AST, *rule* : RULE, *head* : HEAD, *next* : LR)

AST 型の *seed* は解析の結果を Match，MisMatch，FAIL のいずれかで表す．解析に成功した場合は Match，失敗した場合は MisMatch，保留の場合は FAIL である．RULE 型の *rule* は適用した rule の情報を保持する．HEAD 型の *head* は rule に関連する左再帰の情報を保持する．以下に HEAD のデータ型を示す．

```
01.APPLY-RULE(R,P)
02.   let m = RECALL(R,P)
03.   if m = NIL
04.     then let lr = new LR(FAIL,R,NIL,LRStack)
05.          LRStack ← lr
06.          m ← new MEMOENTRY(lr,P)
07.          MEMO(R,P) ← m
08.          let ans = EVAL(R.body)
09.          LRStack ← LRStack.next
10.          m.pos ← Pos
11.          if lr.head ≠ NIL
12.            then lr.seed ← ans
13.                 return LR-ANSWER(R,P,m)
14.            else  m.ans ← ans
15.                 return ans
16.     else  Pos ← m.pos
17.          if m.ans is LR
18.            then SETUP-LR(R,m.ans)
19.                 return m.ans.seed
20.            else  return m.ans
```

図 1　APPLY-RULE プロシージャ
Fig. 1　APPLY-RULE procedure.

HEAD : (*rule* : RULE, *involvedSet* : SETofRULE, *evalSet* : SETofRULE)

rule は発生した左再帰の head rule を保持する．*involvedSet* は発生した左再帰の involved rule の集合を保持する．rule を適用した後 LRStack をポップする．

SETUP-LR プロシージャを**図 2** に示す．SETUP-LR プロシージャは，発生している左再帰の情報を LRStack から得るために用いる．LRStack から得た左再帰の情報をメモの HEAD データに格納する．

LR-ANSWER プロシージャを**図 3** に示す．LR-ANSWER プロシージャは，発生した左再帰の解析を行う．解析を保留しておいた左再帰を再度解析するとき GROW-LR プロシージャを実行する．

GROW-LR プロシージャを**図 4** に示す．GROW-LR プロシージャは，解析を保留しておいた左再帰を再度解析するときに実行する．左再帰が発生している部分の解析を繰り返

```
01.SETUP-LR(R,L)
02.   if L.head = NIL
03.     then L.head ← new HEAD(R,{},{})
04.   let s = LRStack
05.   while s.head ≠ L.head
06.     do s.head ← L.head
07.        L.head.involvedSet ← L.head.involvedSet ∪ {s.rule}
08.        s ← s.next
```

図 2　SETUP-LR プロシージャ
Fig. 2　SETUP-LR procedure.

```
01.LR-ANSWER(R,P,M)
02.   let h = M.ans.head
03.   if h.rule ≠ R
04.     then return M.ans.seed
05.     else  M.ans ← M.ans.seed
06.           if M.ans = FAIL
07.             then return FAIL
08.             else  return GROW-LR(R,P,M,h)
```

図 3　LR-ANSWER プロシージャ
Fig. 3　LR-ANSWER procedure.

```
01.GROW-LR(R,P,M,H)
02.   HEADS(P) ← H
03.   while TRUE
04.     do
05.        Pos ← P
06.        H.evalSet ← COPY(H.involvedSet)
07.        let ans = EVAL(R.body)
08.        if ans = FAIL or Pos ≤ M.pos
09.          then break
10.        M.ans ← ans
11.        M.pos ← Pos
12.   HEADS(P) ← NIL
13.   Pos ← M.pos
14.   return M.ans
```

図 4　GROW-LR プロシージャ
Fig. 4　GROW-LR procedure.

```
01.RECALL(R,P)
02.   let m = MEMO(R,P)
03.   let h = HEADS(P)
04.   if h = NIL
05.     then return m
06.   if m = NIL and R ∉ {h.rule} ∪ h.involvedSet
07.     then return new MEMOENTRY(FAIL,P)
08.   if R ∈ h.evalSet
09.     then h.evalSet ← h.evalSet \ {R}
10.          let ans = EVAL(R.body)
11.          m.ans ← ans
12.          m.pos ← Pos
13.   return m
```

図 5　RECALL プロシージャ
Fig. 5　RECALL procedure.

し行い，入力の読み込み位置（POSITION）を可能な限り読み進める．また，実行する際 HEADS データに現在の POSITION における HEAD データを格納する．HEADS のデータ型は以下のとおりである．

HEADS:POSITION → HEADS

POSITION を可能な限り読み進んだら HEADS の内容を消去する．

RECALL プロシージャを**図 5** に示す．RECALL プロシージャはメモから解析結果を取得するのに用いる．また，左再帰が発生している場合はその左再帰の involved rule を解析するときに，メモから情報を取得せずに解析を進める．

2.2　従来手法を用いた解析例

従来手法を用いて (2) の文法について，入力を “abb” として実際に解析する．解析の様子を**図 6** に示す．まず，図 6 (a) のように S→A→S の間接左再帰が発生する．そのまま解析を続けると無限再帰になるため，ここで左再帰の解析を保留する．バックトラックして解析を進めると図 6 (b) のようになる．入力 1 トークン目の “a” が Match となるので，保留しておいた左再帰の解析が可能になる．GROW-LR プロシージャを実行し，左再帰の解析を進めると図 6 (c) のようになる．GROW-LR プロシージャは POSITION を可能な限り読み進めようとするためさらに解析を進める．図 6 (d) のようになり，これ以上 POSITION を読み進めることができないため解析を終了する．すべての入力を読み込めたため，この解析は成功となる．

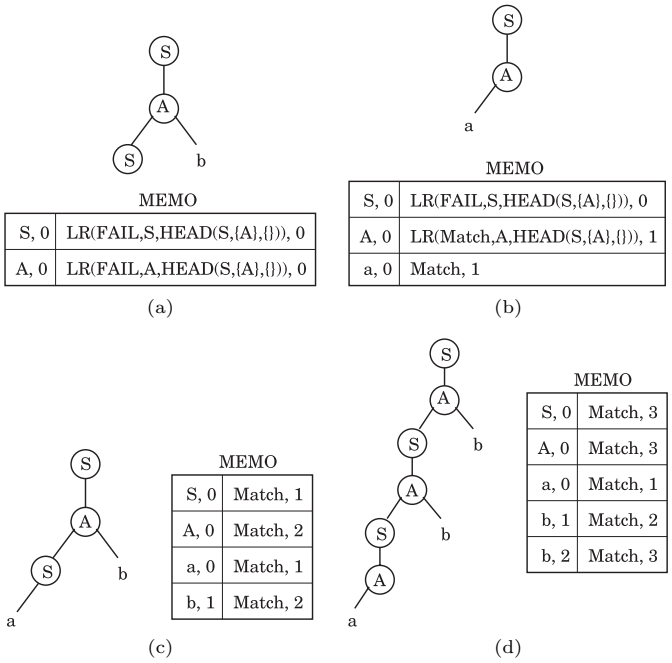

図 6　従来手法の解析成功例
Fig. 6　Case of success with traditional method.

2.3　従来手法の問題点

従来手法には以下の 2 つの問題点がある．

(1)　解析が異常終了する文法が存在する．

(2)　特定の文法と入力の解析で構文木が意図しない木になる．

まず，問題点 (1) について解説する．以下の文法について考える．この文法は，同じ POSITION で head rule が異なる左再帰が発生する文法である．

$$S \leftarrow Ab/b$$
$$A \leftarrow Aa/Sa \tag{4}$$

入力を “baab” として実際に解析する．解析の様子を**図 7** に示す．まず図 7 (a) のように A→A の直接左再帰が発生するので，左再帰の情報をメモに書き込む．バックトラックすると図 7 (b) のように今度は S→A→S の間接左再帰が発生する．ここでメモに左再帰の情報を記録すると，その前に発生した A→A の直接左再帰の情報は，S→A→S の間接左再帰の情報によって上書きされる．次に，図 7 (c) のように，“b” が Match となるので S の左再帰が解析可能になる．そのため，GROW-LR プロシージャを実行し解析を続ける．その後，図 7 (d) で A→A の直接左再帰が再度発生する．メモに左再帰の情報があるため，SETUP-LR プロシージャを実行する．SETUP-LR プロシージャは LRStack から左再帰の情報を得ようとする．しかし，このとき LRStack は空になっている．これは，メモから左再帰の情報を得るときは，LRStack がプッシュされないためである．空の LRStack から情報を取得しようとするので，この解析は異常終了する．

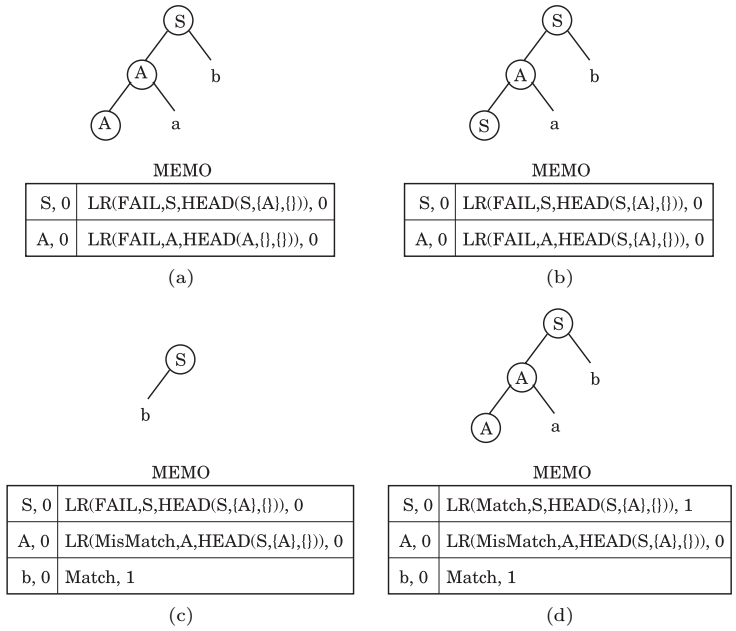

図 7　従来手法の解析失敗例
Fig. 7　Case of failure with traditional method.

次に，問題点 (2) について説明する．以下の文法について入力を “aa” として解析する．解析の様子を**図 8** に示す．

$$S \leftarrow Aa/a$$
$$A \leftarrow S \tag{5}$$

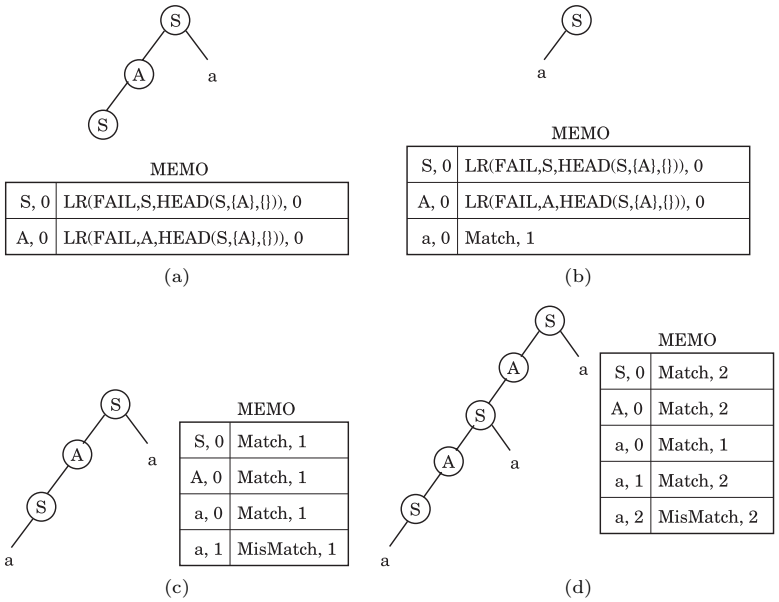

図 8　従来手法の問題点
Fig. 8　Issue of traditional method.

まず，図 8 (a) のように S→A→S の間接左再帰が発生する．バックトラックして解析を進めると図 8 (b) のようになる．“a” が Match となり，POSITION が読み進められるので S の左再帰が解析可能になる．次に，GROW-LR プロシージャを実行すると図 8 (c) のようになる．この解析は Match となる．GROW-LR プロシージャは POSITION を可能な限り読み進めようとするため，さらに解析を進め図 8 (d) のようになる．すると，入力の 3 トークン目には文字列が存在しないため MisMatch となる．入力をすべて読み込んだため，解析は成功となり終了する．ここで図 8 (d) の構文木を見ると，木の末端が “aaa” となっている．つまり，“aaa” という入力が解析されたことになっている．しかし，実際の入力は “aa” であるため構文木は正しくない．これは，従来手法の GROW-LR プロシージャが POSITION を可能な限り読み進めようとするために起こる．このように従来手法は GROW-LR プロシージャにより POSITION が進みすぎて，意図した木とは異なる構文木になるという問題点がある．

```
01.APPLY-RULE(R,P)
02.   let m = MEMO(R,P)
03.   if (m ≠ NIL and R = terminal) or R ∈ Call
04.      then Pos ← m.pos
05.           return m.ans
06.      else  return UPDATE-MEMO(R,P)
```

図 9　提案手法の APPLY-RULE プロシージャ
Fig. 9　Proposal APPLY-RULE procedure.

3. 提 案 手 法

本章では，提案手法について述べる．前章で確認した従来手法の問題点を以下に示す．

(1)　空の LRStack から情報を得ようとする場合がある．

(2)　GROW-LR プロシージャ実行時に POSITION が進みすぎる場合がある．

これらの問題点を解決するアルゴリズムについて説明する．

3.1　提案手法のアルゴリズム

問題点 (1) に対応するため，LRStack を使用せずに解析できるように改良する．よって，メモに左再帰の情報を記録しない．また，問題点 (2) に対応するため，GROW-LR プロシージャを改良する．提案手法では，従来手法で用いていた LRStack，HEADS，データ型 LR，データ型 HEAD，RECALL プロシージャ，SETUP-LR プロシージャ，LR-ANSWER プロシージャは使用しない．新たに，現時点での POSITION の最大値を格納する変数 maxPos，GROW-LR プロシージャの実行前の POSITION を格納する変数 oldPos，適用した rule を格納する Call スタック，GROW-LR プロシージャを実行中かどうかを判定する growState，メモを更新する UPDATE-MEMO プロシージャを使用する．なお，growState は GROW-LR プロシージャが実行中なら TRUE，実行中でなければ FALSE となる．提案手法は，APPLY-RULE プロシージャ，UPDATE-MEMO プロシージャ，GROW-LR プロシージャの 3 つのプロシージャからなる．

3.1.1　問題点 (1) への対応

提案手法で用いる改良した APPLY-RULE プロシージャを**図 9** に示す．従来手法の問題点 (1) を解決するために，メモが左再帰の情報を用いないように変更する．また，メモを更新する際は，新たに追加した UPDATE-MEMO プロシージャを実行する．提案手法の APPLY-RULE プロシージャは以下の処理を実行する．

メモに解析結果があり，かつ Call にその rule がある場合はメモから情報を取得する．そ

```
01.UPDATE-MEMO(R,P)
02.   Call.push(R)
03.   if MEMO(R,P) = NIL
04.      then let m = MEMOENTRY(FAIL,P)
05.      else let m = MEMO(R,P)
06.   MEMO(R,P) ← m
07.   let ans = EVAL(R.body)
08.   if ans = Match
09.      then m.pos ← Pos
10.   if maxPos < m.pos
11.      then maxPos = m.pos
12.   m.ans = ans
13.   Call.pop
14.   if growState = FALSE and rule = startSymbol
15.      then while TRUE
16.                do
17.                   ans ← GROW-LR(R,P,m)
18.                   if ans = FAIL or Pos ≥ input.size
19.                      then break
20.   return ans
```

図 10　UPDATE-MEMO プロシージャ
Fig. 10　UPDATE-MEMO procedure.

れ以外の場合はメモの内容を更新するため UPDATE-MEMO プロシージャを実行する．たとえば，S→A→S の間接左再帰を解析するときは，1 度目の S と A の適用時に UPDATE-MEMO プロシージャを実行しメモを更新する．2 度目の S の適用時にメモから情報を取得する．

次に，新たに追加した UPDATE-MEMO プロシージャを**図 10** に示す．UPDATE-MEMO はメモの内容を更新する際に用いる．まず，解析する rule を Call にプッシュする．この Call にプッシュされた rule は，次にその rule を適用するときにメモから情報を取得する．Call に rule がなければさらに解析を進め，メモを更新する．maxPos の初期値は 0 であり，POSITION が進むとそのたびに更新される．メモを更新した後，その rule を Call からポップし，下記の条件をすべて満たすとき GROW-LR プロシージャを実行する．

- growState が FALSE である．
- 更新した rule の左辺が開始記号である．
- 前回の GROW-LR プロシージャの実行結果が FAIL でない．
- POSITION を最後まで読み進んでいない．

```
01.GROW-LR(R,P,M)
02.   Call.push(R)
03.   growState ← TRUE
04.   oldPos ← maxPos
05.   Pos ← P
06.   let ans = EVAL(R.body)
07.   if ans = Match and Pos > m.pos
08.      then MEMO(R,P) ← MEMOENTRY(ans,Pos)
09.   growState ← FALSE
10.   Call.pop
11.   if oldPos ≠ maxPos
12.      then return ans
13.      else  return FAIL
```

図 11　提案手法の GROW-LR プロシージャ
Fig. 11　Proposal GROW-LR procedure.

このように，改良した APPLY-RULE プロシージャと新たに追加した UPDATE-MEMO プロシージャを用いると，LRStack を必要とせず，従来手法の問題点 (1) が解決できる．

3.1.2　問題点 (2) への対応

提案手法で用いる改良した GROW-LR プロシージャを**図 11** に示す．従来手法の GROW-LR プロシージャは POSITION を可能な限り読み進める．しかし，文法によっては POSITION が進みすぎる．これは，従来手法の GROW-LR プロシージャが，左再帰が発生している部分のみを解析するためである．そこで，POSITION を 1 度だけ読み進めるようにし，その後，再度開始記号から解析するように改良する．この変更で問題点 (2) が解決できる．

GROW-LR プロシージャではまず，growState を TRUE とする．また現時点での maxPos の値を oldPos に格納する．次に再度文法を解析する．POSITION を読み進められる場合，その結果をメモに記録する．その後，growState を FALSE とする．最後に oldPos と maxPos の値を比較し，一致した場合は FAIL を返す．ここで FAIL が返されると解析失敗となり解析が終了する．oldPos と maxPos が異なれば，POSITION が読み進められるので次の解析に移る．もしここで POSITION が最後なら入力文字列をすべて解析できるので，解析成功となり解析が終了する．このように maxPos と oldPos を比較する更新検知を用いて，解析を続けるか否かを判断する．

提案手法のアルゴリズムは必ず停止する．解析の際，POSITION が進むにつれて maxPos が更新される．maxPos が入力長と一致したとき，入力がすべて解析できたと判断し，解析成功として提案手法のアルゴリズムは停止する．maxPos が更新されない場合は，その時点

で FAIL を返し，解析失敗として提案手法のアルゴリズムは停止する．

3.2　提案手法を用いた解析例

提案手法を用いて以下の文法について，入力を “baab” として実際に解析を行う．

$$S \leftarrow Ab/b$$
$$A \leftarrow Aa/Sa \tag{6}$$

まず**図 12** (a) のようにメモから A の情報を取得すると FAIL が格納されているのでバックトラックする．バックトラックして次の選択肢を解析すると図 12 (b) のようになる．この解析は FAIL となり，バックトラックして次の選択肢を解析する．図 12 (c) で入力 1 トークン目の “b” が Match となり POSITION が 1 まで読み進められる．POSITION が読み進められるので maxPos も更新され 1 となる．ここで，GROW-LR プロシージャを実行する条件がすべて満たされるので，GROW-LR プロシージャを実行し解析を続ける．GROW-LR プロシージャにより，oldPos が設定され図 12 (d) のようになる．メモから A の情報を取得すると MisMatch なのでバックトラックして図 12 (e) のようになる．ここで A が入力 2 トークン目の “a” まで Match となるので，メモが更新される．また POSITION が 2 まで読み進められるので maxPos も 2 に更新される．次に S を適用しようとするが MisMatch となり図 12 (f) のようになる．ここでは POSITION は更新されないが図 12 (e) で maxPos が更新されているので解析を続ける．そして図 12 (g) のようになり，入力文字列がすべて解析できるので解析成功となる．

4.　評価・考察

4.1　評 価 手 法

提案手法に関して以下の 3 点の評価を行う．

(1)　従来手法が対応している文法に対応するか．
(2)　従来手法と比較して解析時間が長くなっていないか．
(3)　従来手法が対応していない文法に対応するか．

(1) の評価は，従来手法の論文[5] で評価に用いられた Java のプライマリ表現[6] を利用する．評価に用いる文法を**図 13** に，その入力を**図 14** に示す．従来手法と提案手法を用いて解析を行い，その結果と構文木を比較する．(2) の評価は以下の文法を従来手法と提案手法を用いて解析を行い，実行時間を比較する．

$$S \leftarrow Aa/a$$
$$A \leftarrow S \tag{7}$$

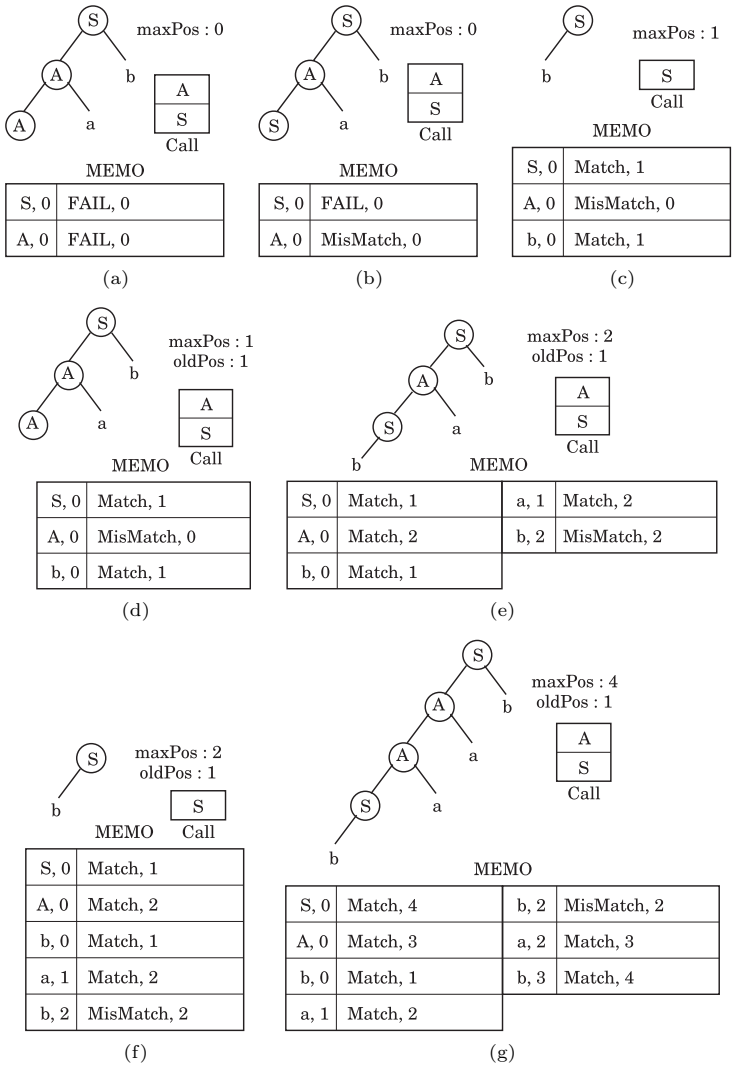

図 12　提案手法の解析成功例
Fig. 12　Case of success with proposal method.

```
Primary                          ←   <PrimaryNoNewArray>
PrimaryNoNewArray                ←   <ClassInstanceCreationExpression>
                                 /   <MethodInvocation>
                                 /   <FieldAccess>
                                 /   <ArrayAccess>
                                 /   this
ClassInstanceCreationExpression  ←   new <ClassOrInterfaceType> ()
                                 /   <Primary> . new <Identifier> ()
MethodInvocation                 ←   <Primary> . <Identifier> ()
                                 /   <MethodName> ()
FieldAccess                      ←   <Primary> . <Identifier>
                                 /   super . <Identifier>
ArrayAccess                      ←   <Primary> [ <Expression> ]
                                 /   <ExpressionName> [ <Expression> ]
ClassOrInterfaceType             ←   <ClassName> / <InterfaceTypeName>
ClassName                        ←   C / D
InterfaceTypeName                ←   I / J
Identifier                       ←   x / y / <ClassOrInterfaceType>
MethodName                       ←   m / n
ExpressionName                   ←   <Identifier>
Expression                       ←   i / j
```

図 13　Java プライマリ表現
Fig. 13　Java's Primary expressions.

```
"this"
"this.x"
"this.x.y"
"x[i][j].y"
```

図 14　Java プライマリ表現の入力
Fig. 14　Input of Java Primary expressions.

入力は "a" を繰り返したトークン列を与える．(3) の評価は以下の文法について提案手法を用いて解析を行い，解析が可能であるかを調べる．

$$S \leftarrow Ab/b \qquad A \leftarrow Aa/Sa \tag{8}$$

$$S \leftarrow Aa/a \qquad A \leftarrow S \tag{9}$$

文法 (8) の入力を "baab"，文法 (9) の入力を "aaa" とする．
従来手法と提案手法のアルゴリズムは Scala を用いて実装した．評価環境を**表 1** に示す．

表 1　評価環境
Table 1　Evaluation environment.

カーネル	Windows7 Home Premium 32 bit
CPU	Intel(R) Core(TM)2 Duo CPU P8700 2.53 GHz 2.53 GHz
メモリ	4.0 GB
scala	scala version 2.8.1

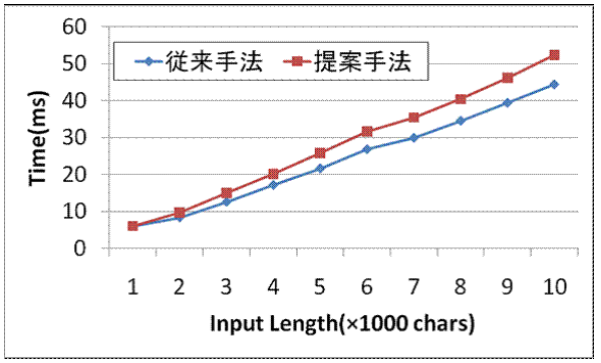

図 15　従来手法と提案手法による左再帰を含む文法の解析時間
Fig. 15　Analysis time of evaluating grramar containing left recursion with traditional method and proposal method.

4.2　評価結果

まず，評価 (1) を行った．従来手法と提案手法の解析結果，および構文木は一致した．よって，従来手法が対応している文法は提案手法でも対応できると考えられる．

次に，評価 (2) を行った．**図 15** に評価結果を示す．図 15 の横軸は入力文字列長であり，1,000 文字から 10,000 文字まで 1,000 文字単位で増加する．図 15 の縦軸は実行時間をミリ秒で表している．簡単な間接左再帰の文法の場合，従来手法と提案手法を用いたときのどちらの場合も入力文字数が増えると，それにほぼ比例して解析時間が増えることが分かる．つまり従来手法を用いて線形時間で解析可能な文法は，提案手法を用いても線形時間で解析可能であるといえる．また，提案手法は従来手法より解析時間が長くなっている．これは，提案手法の非終端記号呼び出し回数が多いためである．従来手法は左再帰の再解析を head rule から行うが，提案手法は開始記号から行う．よって，提案手法では非終端記号呼び出し回数が多くなり，解析時間が長くなると考えられる．今回の評価では簡単な文法を用いてい

るため，解析時間に大きな差は見られない．しかし，木の深い位置で左再帰が発生するような文法の場合，従来手法と提案手法の非終端記号呼び出し回数の差が大きくなる可能性がある．よって，提案手法の解析時間がより長くなる可能性がある．

最後に，評価 (3) を行った．文法 (8) と文法 (9) はどちらも解析することができた．また，構文木も正しいことが確認できた．従来手法で対応できない文法が，提案手法で対応できることが確認できた．

4.3 計　算　量

従来手法と提案手法の計算量について考察する．従来手法と提案手法のどちらも最悪の場合，両手法ともメモ化の効果が得られないため，計算量は指数時間となる．また，最良の場合，両手法とも左再帰がいっさい発生しないため，計算量は線形時間となる．よって，従来手法と提案手法の最悪と最良の計算量は一致する．しかし，文法によっては実際の解析時間が従来手法より長くなる可能性がある．なぜなら，前節で述べたように，提案手法は開始記号から再解析を行うためである．

5. 関 連 研 究

本研究と関連する研究に白田ら[7] の手法がある．アルゴリズム・対応する文法・実行効率・メモリ使用量について提案手法と比較する．

Warth らの手法で異常終了する問題に対して，白田らは新たに HEADTABLE を追加して解決した．白田らの手法は，左再帰が発生したときに，左再帰の情報を HEADTABLE に記録する．その後，左再帰の情報を HEADTABLE から取得し，異常終了する問題に対応した．一方，提案手法は，左再帰の情報を記録せず，すでに呼び出した非終端記号を呼び出そうとしたときにメモを参照する．

白田らの手法は，2.2 節で述べた POSITION が進みすぎるという問題には対応していない．また，GROW-LR プロシージャの実行中に再び GROW-LR プロシージャを実行する必要がある文法には対応していない．たとえば，2.3 節で示した文法 (4) に “baaab” を入力した場合，提案手法は正しく解析できる．しかし，白田らの手法では解析できない．

また，白田らの手法は，HEADTABLE の操作を行うため，従来手法より解析時間が長くなる．提案手法は開始記号から再解析を行うため，従来手法より解析時間が長くなる．白田らの手法は従来手法と同じ位置から再解析を行うため，4.2 節で述べた木の深い位置で左再帰が発生する文法の場合，提案手法は白田らの手法より解析時間が長くなる可能性がある．

また，白田らの手法は，HEADTABLE を追加しているため，従来手法よりもメモリ使用量が増加する．一方，本研究で提案した手法は，従来手法で使用していたデータ型をいくつか削減しているため，メモリ使用量が減少する．よって，提案手法は白田らの手法よりもメモリ使用量が少ない．

6. 結　　論

本稿では，更新検知を用いた Packrat Parser を提案し，実装した．結果，従来手法では対応できない文法に対応することができた．また，従来手法を用いて線形時間で解析できる文法は，提案手法を用いても線形時間で解析できる．

提案手法は，対応できる文法を増やすことを目的としたため，今後の課題としては解析の効率化，速度向上があげられる．

参　考　文　献

1) Ford, B.: Packrat Parsing: Simple, Powerful, Lazy, Linear Time, *ICFP '02: Proc. 7th ACM SIGPLAN International Conference on Functional Programming*, New York, NY, USA, pp.36–47 (2002).
2) Ford, B.: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation, *Symposium on Principles of Programming Languages* (2004).
3) Ford, B.: Packrat Parsing: A practical linear-time algorithm with backtracking, Master's thesis, Massachusetts Insitute of Technology (Sep. 2002).
4) Grimm, R.: Better Extensibility Through Modular Syntax, *PLDI '06: Proc. 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, New York, NY, USA, pp.38–51, ACM Press (2006).
5) Warth, A., Douglass, J.R. and Millstein, T.: Packrat Parsers Can Support Left Recursion, *ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation*, January 7–8, 2008, San Francisco, California, USA, pp.103–110 (2008).
6) Gosling, J., Joy, B., Steele, G. and Bracha, G.: *The Java Language Specification*, 3rd Edition, Addison-Wesly (2005).
7) 白田佳章，木山真人，芦原　評：同一入力位置で複数発生する左再帰へ対応した Packrat Parser の設計と実装，情報処理学会論文誌　プログラミング，Vol.4, No.2 (2011).

(平成 22 年 12 月 17 日受付)
(平成 23 年　3 月 29 日採録)

後藤　勇太

昭和 63 年生．平成 23 年熊本大学工学部情報電気電子工学科卒業．同年熊本大学大学院自然科学研究科情報電気電子工学専攻入学．現在に至る．

白田　佳章

昭和 60 年生．平成 20 年熊本大学工学部数理情報システム工学科卒業．同年熊本大学大学院自然科学研究科情報電気電子工学専攻入学．現在に至る．

木山　真人（正会員）

昭和 51 年生．平成 11 年広島市立大学情報科学部情報工学科卒業．平成 15 年広島市立大学大学院情報科学研究科情報科学専攻博士後期課程修了．同年熊本大学工学部数理情報システム工学科助手．現在，熊本大学大学院自然科学研究科情報電気電子工学専攻助教．博士（情報工学）．プログラミング言語，言語処理系，オブジェクト指向言語の高速化手法に興味を持つ．

芦原　　評（正会員）

昭和 39 年生．昭和 62 年東京大学理学部情報科学科卒業．平成 4 年東京大学大学院理学系研究科博士課程修了．博士（理学）．同年電気通信大学情報工学科助手．平成 11 年熊本大学工学部数理情報システム工学科助教授．現在，熊本大学大学院自然科学研究科情報電気電子工学専攻准教授．