左再帰に対応するPackrat Parserの実装

後藤勇太† 木山真人† 芦原 評†

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．Alessandro Warth らは，左再帰を含む文法を，右再帰への変換無しに解析を可能にした．しかし，Alessandro 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．そこで本研究では，左再帰を含む文法を右再帰への変換無しに解析でき，かつ従来手法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 to Parse Left Recursive Grammars

Yuta Goto ,† Masato Kiyama † and Hyo Ashihara†

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. Alessandro 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. This paper presents imprementation and evaluation of Packrat Parser 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．

$$S \leftarrow A$$
$$A \leftarrow Sb/a \tag{2}$$

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．

$$S \leftarrow aS'$$
$$S' \leftarrow bS'/b \tag{3}$$

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

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．Alessandro Warth ら[5] は，左再帰を含む文法を，右再帰への変換無しに解析できる Packrat Parser を実装した．Alessandro 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．

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る手法について提案し，評価を行う．2 章では，Alessandro 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，Alessandro 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成る．

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．

$$\mathrm{LR} : (\mathit{seed} : \mathrm{AST}, \mathit{rule} : \mathrm{RULE}, \; \mathit{head} : \mathit{HEAD}, \mathit{next} : \mathit{LR})$$

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

```
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

```
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

左再帰の情報を保持する．以下に HEAD のデータ型を示す．

$$\mathrm{HEAD} : (\mathit{rule} : \mathrm{RULE}, \mathit{involvedSet} : \mathrm{SETofRULE}, \; \mathit{evalSet} : SETofRULE)$$

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

```
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

```
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

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し行い，入力の読み込み位置 (POSITION) を可能な限り読み進める．以降，この入力の読み込み位置を可能な限り読み進めることを GROW すると呼ぶ．また，実行する際 HEADS データに現在の入力位置における HEAD データを格納する．HEADS のデータ型は以下の通りである．

HEADS:POSITION → HEADS

入力位置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．

```
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

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，GROW すると図 6(c) のようになる．GROW-LR プロシージャは入力位置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，これ以上入力位置を読み進めることができないため解析を終了する．全ての入力を読み込めたため，この解析は成功となる．

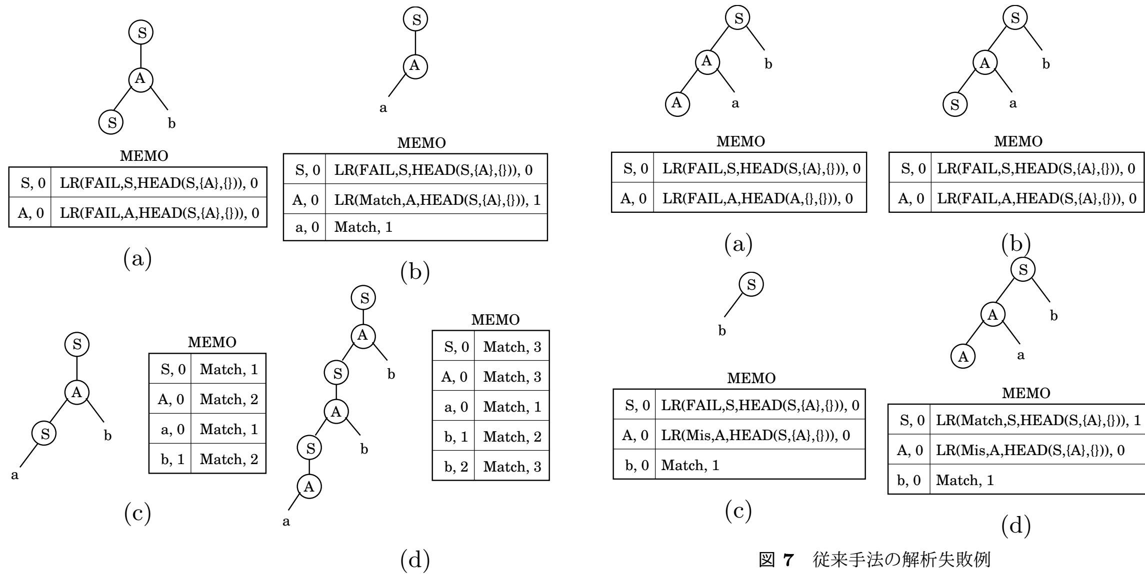

図 6　従来手法の解析成功例

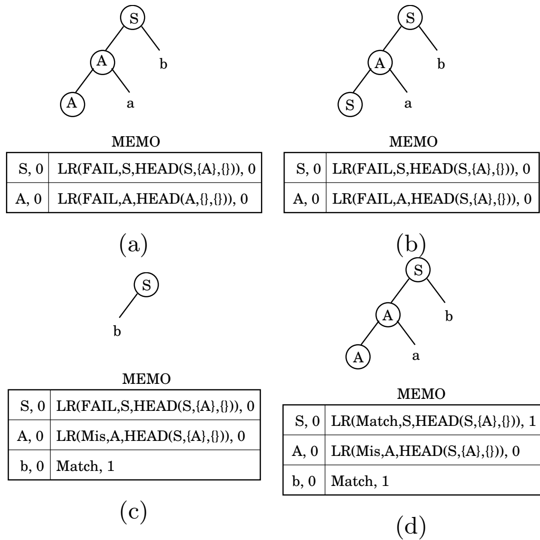

図 7　従来手法の解析失敗例

2.3　従来手法の問題点

従来手法にはある文法において，解析が異常終了する問題がある．以下の文法について考える．この文法は，同じ入力位置で head rule が異なる左再帰が発生する文法である．

$$S \leftarrow Ab/b$$
$$A \leftarrow Aa/Sa \qquad (4)$$

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こむ．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．

3. 提 案 手 法

本章では，提案手法について述べる．従来手法には，空の LRStack から情報を得ようとする問題があった．この問題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

新たなアルゴリズムを考案するために，左再帰が発生する条件について考察した．その結果，文法の非終端記号を開始記号から順次呼び出し，その後ある非終端記号が 2 度目に呼び出されたときに左再帰が発生することがわかった．そこで，1 度呼び出した非終端記号をリストに記録し，その後，リストに記録された非終端記号を呼び出した際にメモから情報を取得するように変更した．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，1 度呼び出した非終端記号を格納する Call スタック，GROW する非終端記号を格納する Grow リスト，メモを更新する UPDATE-MEMO プロシージャを使用する．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成る．

提案手法で用いる改良した APPLY-RULE プロシージャを図 8 に示す．従来手法の問題点を解決するために，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．その時，メモから取得した解析結果が FAIL の場合は，その非終端記号を Grow に追加する．Grow に追加された非終端記号は，後に GROW する条件を満たしたときに GROW される．メモに解析結果がない，または Call にその rule がない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 9** 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

```
01.APPLY-RULE(R,P)
02.  let m = MEMO(R,P)
03.  if m ≠ NIL and R ∈ Call
04.    then Pos ← m.pos
05.         if m.ans = FAIL and R is nonTerm
06.           then Grow.append(R)
07.         return m.ans
08.    else return UPDATE-MEMO(R,P)
```

図 8 提案手法の APPLY-RULE プロシージャ

```
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 m.ans ≠ Pos
11.    then m.ans ← ans
12.  if R ∈ Grow
13.    then pos < m.pos and P < m.pos
14.         ans ← GROW-LR(R,P,m)
15.  Call.pop
16.  return ans
```

図 9 UPDATE-MEMO プロシージャ

```
01.GROW-LR(R,P,M)
02.  while Call.head ≠ R
03.         Call.pop
04.  while TRUE
05.    do
06.        Pos ← P
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.  if ans = Match and Pos > M.pos
13.    then MEMO(R,P) ← MEMOENTRY(ans,
Pos)
14.  Pos ← M.pos
15.  return M.ans
```

図 10 提案手法の GROW-LR プロシージャ

を更新する．メモを更新した後，その rule が GROW に含まれている場合，GROW-LR プロシージャを実効する．

提案手法で用いる改良した GROW-LR プロシージャを**図 10** に示す．まず，Call の先頭が Grow する rule になるまで，Call をポップする．その後，rule を GROW して入力位置を可能な限り読み進める．

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．

$$S \leftarrow Ab/b$$
$$A \leftarrow Aa/Sa \qquad (5)$$

まず**図 11**(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．また，A のメモが FAIL のため，A を Grow に追加する．バックトラックして次の選択肢を解析すると図 11(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．S も FAIL のため，Grow に追加する．図 11(c) で入力 1 トークン目の “b” が Match となり S について入力位置が 1 まで読み進められる．ここで，S が Grow に含まれるため，S を GROW すると図 11(d) のようになる．A は MisMatch であるため，バックトラックする．次に，図 11(e) のように A について入力位置が 2 まで読み進められる．ここで，A は Grow に含まれるため，A を GROW する．A を GROW すると，図 11(f) のようになり，全ての入力を解析できたので成功となる．

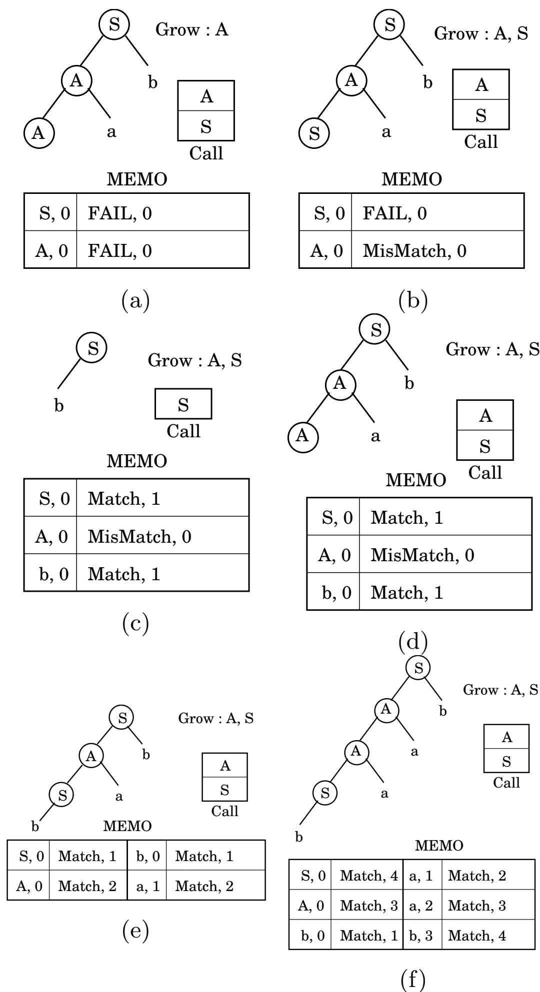

図 11　提案手法の解析成功例

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．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．

$$S \leftarrow Aa/a$$
$$A \leftarrow S \quad (6)$$

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．

$$S \leftarrow Ab/b \qquad\qquad S \leftarrow Ab$$
$$A \leftarrow Aa/Sa \quad (7) \qquad A \leftarrow Aa/Sa/a \quad (8)$$

文法 (7) の入力を “baab”，文法 (8) の入力を “aab” 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

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

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，及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 12** に評価結果を示す．図 12 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．縦軸は実行時間をミリ秒で表している．図 12 を見ると，提案手法の実行時間は従来手法とほぼ同じであることがわかる．

最後に，評価 (3) を行った．文法 (7) と文法 (8) 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．

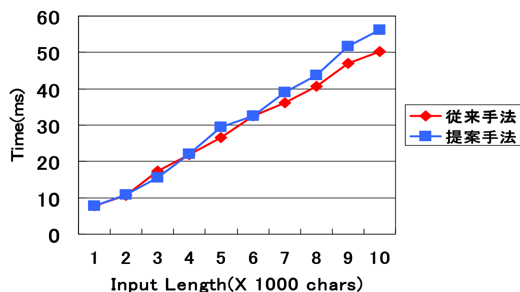

図 12　従来手法と提案手法による左再帰を含む文法の解析時間

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一切発生しないため，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．

5. 結 論

本稿では，左再帰に対応する 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，提案手法の実行効率が従来手法と同等であることが確認できた．

質 疑 応 答

Q 性能評価のグラフを見ると，一見線形のようだが，従来手法は文法によっては線形でないケースがあった．提案手法についてはどうなのか？

A 従来手法の場合と同様に，提案手法でも線形時間でないケースが存在すると考えられます．

Q その問題を解決するアイディアはあるのか？

A 今のところありません．

Q 性能評価のグラフを見ると，若干ではあるが入力文字数が増えると提案手法の方が遅くなるように見える．入力文字数がもっと増えるとその差がさらに大きくなるのではないか？

A 従来手法と提案手法で構文木の構築手順は変わらないはずなので，入力文字数による影響はないと考えています．

Q 提案手法のアルゴリズムは従来手法のものと比較してシンプルになっていると言っていたがどのくらいシンプルになったか？

A 従来手法はプロシージャが 5 つだったのに対し，提案手法は 3 つのみとなっています．また，scala ソースコードは 100 行ほど削減できました．

謝 辞

本研究を進めるにあたって，熊本大学大学院自然科学研究科情報電気電子工学専攻　芦原評准教授には貴重な御時間の中，様々な指導や助言をいただき，心から御礼申し上げます．

また，熊本大学大学院自然科学研究科情報電気電子工学専攻　木山真人助教授には本研究を進めるにあたり様々な助言をいただき，心から感謝致しております．

参 考 文 献

1) Bryan Ford.: Packrat Parsing: Simple, Powerful, Lazy, Linear Time, *ICFP '02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pp.36–47, New York, NY, USA, 2002.

2) Bryan Ford.: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation, *Symposium on Principles of Programming Languages*, 2004.

3) Bryan Ford.: Packrat Parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Insitute of Technology, September 2002.

4) Robert Grimm.: Better Extensibility Through Modular Syntax. *PLDI '06:Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2006, New York, NY, USA, ACM Press, pp.38–51.

5) Alessandro W., James R Douglass. and Todd Millstein.: 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) James Gosling, Bill Joy, Guy Steele, and Gilad Bracha.: The Java Language Specification, Third Edition. Addison-Wesly, 2005.