SGLR: 逐次型一般化 LR パーザの Prolog による実現†

沼崎浩明†† 田中穂積††

本論文では，DCG 文法に対する一般化 LR パーザ SGLR (A Sequential Generalized LR Parser) を効率良く Prolog 上に実現する手法を示す．SGLR は横型探索によって文の曖昧性を扱い，スタックの統合によって無駄な再計算を防ぐ点で富田法に従い，DCG の補強項の計算のために，すべての解析木を独立に扱う点で富田法と異なる．効率化の観点から，SGLR は以下の特徴を持つ．(1) スタックの表現形式を工夫し，すべてのスタック操作をユニフィケーションのみを用いて行うこと．(2) LR パーズ表の各エントリをパーザのプロセスとみなし，一つのホーン節で記述すること．(3) Prolog 処理系のダブルハッシュによる述語検索機能を利用して，プロセスの選択を高速に行うこと．(4)すべての解析情報を，副作用を用いずに保持すること．(5)補強項の評価以外にはバックトラックを用いないこと．規則数 339 の英語の DCG を用いた実験により，Prolog 上に実現されたパーザとして，SGLR は従来のものを凌ぐ性能を有することが確認された．また，富田法を DCG パーザとして Prolog 上に実現した場合に生ずる問題点についても検討し，SGLR の構文解析の戦略の妥当性を示す．

1. はじめに

本論文では，DCG 文法に対する一般化 LR パーザ SGLR (A Sequential Generalized LR Parser) を効率良く論理型言語 Prolog 上に実現する手法を示す．

一般化 LR 構文解析法は，LR 構文解析法[2]を，一般の CFG に適用するためのアルゴリズムである．CFG の曖昧性は LR パーズ表のエントリの競合 (conflict) として現れる．文の解析過程は，競合のあるエントリにおいて非決定的となり，縦型探索あるいは横型探索によってすべての解析木を計算することが必要となる．曖昧性によって生ずる複数の解析過程が同一の部分木を生成する場合，これは無駄な再計算となる．これを防ぐアルゴリズムとして，富田法[1]が良く知られている．富田法は，横型探索によって曖昧性を扱い，複数のスタックの共通部分を統合することによって再計算を防ぐことができる．これによって富田法は高い効率性を得ている (ただし，特殊な文法に対する計算量は Earley 法よりも多いことが知られている[3])．

我々は既に，富田法のスタック統合の手法を用いたパーザ SGLR を Prolog 上に試作し，その基本的な考え方と実現方法を示している[4]．このパーザは以下の技術的特徴により解析の効率性を得ている．

(1) LR パーズ表の各エントリをプロセスとみなし，一つのホーン節で記述する．

(2) Prolog 処理系のダブルハッシュによる述語検索機能を利用して，節の選択を高速に行う．

(3) すべての解析情報を，副作用を用いずに保持する．

(4) 補強項の評価以外には，バックトラックを用いない．

本論文では，文献[4]に示した SGLR の効率をさらに徹底化したアルゴリズムを提案し，実験によってその有効性を実証している．また，SGLR と富田法との差異を詳細に検討している．

我々は，SGLR のより一層の効率化を行うために，スタックの表現形式を大幅に変更し，すべてのスタック操作をユニフィケーションのみを用いて行うようにした (3 章参照)．それにより，スタックの統合，還元操作のプログラムの大幅な簡素化を図ることが可能になった (4.6 節参照)．

Prolog 上に実現されたパーザは，これまでに数多く提案されている．BUP[6]，SAX[7]，LangLAB[8] は DCG のパーザとして良く知られており，その中では特に SAX が優れた効率性を示すことが知られているが，SAX は先読みの情報を用いないため，探索空間が SGLR よりやや広い．AID[9] は LR 法に基づいた DCG パーザであり，縦型探索によって曖昧性を扱うが，再計算を許す点で効率性に劣る．また，Kindermann[10] らは，LR 法に基づく LFG パーザを Prolog 上に構築している．このパーザは横型探索によって曖昧性を扱っているが，スタックの先頭の共通要素を統合せず，AID と同様に再計算を許している．YAP[11] は CFG を拡張した新たな文法の枠組を提案し，その

† SGLR: A Sequential Generalized LR Parser in Prolog by Hiroaki Numazaki and Hozumi Tanaka (Department of Computer Science, Faculty of Engineering, Tokyo Institute of Technology).

†† 東京工業大学工学部情報工学科

記述力を生かして規則数を減らすとともに，効率の良い構文解析を実現している．

以下，本論文の2章では SGLR の構成を示す．3章では構造化スタックの表現形式とその操作について述べ，4章では SGLR の記述とその動作を説明する．5章では，SGLR と富田法との相違について述べ，6章では規則数 399 の英語の DCG を用いた実験によって，SGLR の効率性を示す．また，富田法を Prolog 上に忠実に実現した場合のオーバヘッドについても評価する．7章では SGLR の課題について述べ，最後の8章では本論文の結論を示す．

2. SGLR の構成

SGLR の構成を**図1**に示し，各構成部の役割を以下に示す．

- 入力インタフェース
  入力文をトップレベルのゴール列に変換するプログラム．入力文の辞書引きを行い，各語のカテゴリと引数の情報を得た後，以下で説明するカテゴリ節と統合節の並びを作り，トップレベルのゴール列とする（4.7 節参照）．
- カテゴリ節
  ある一つのカテゴリに対し，そのカテゴリをスタックに積むまでの LR 構文解析を行う節．スタックのリストを受けとり，各スタックごとに次に示すエントリ節を呼び出し，その結果のリストを返す（4.2 節参照）．

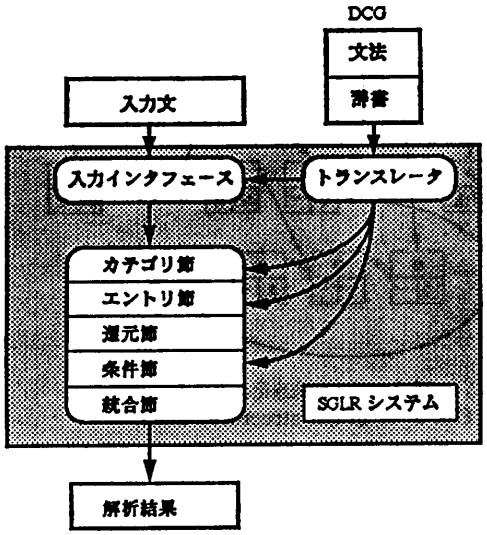


図1 SGLR システム
Fig. 1 SGLR system.

- エントリ節
  LR パーズ表の各エントリを実行する節．エントリの内容によって記述が異なる（4.3 節参照）．
- 還元節
  スタックの要素を必要な数だけ取り出し，次に示す条件節を適用する節．還元の途中でスタックの分岐点に到達した場合は，各枝に対してそれぞれ同じ条件節を適用する（4.4 節参照）．
- 条件節
  DCG 規則中の補強項を評価し，規則適用の可否を決定する節．規則の適用が失敗した場合には，空のスタックを返す（4.5 節参照）．
- 統合節
  先頭の要素が等しいスタックを統合する節．ここで，先頭の要素とは状態番号とカテゴリの引数のリストを意味する（4.6 節参照）．

3. スタックの実現

3.1 スタックの表現形式の定義

以下に，SGLR で用いるスタックの表現形式を BNF 記法を用いて定義する．

```
〈stack〉::=
    [0]|
    [〈state number〉,〈argument〉|〈branch〉]
〈branch〉::=
    〈stack〉|
    〈branch〉/〈stack〉
```

ここで，〈state number〉は LR パーズ表の状態番号，〈argument〉は DCG のあるカテゴリが持つ引数のリストである．また，リスト [⋯] 中の '|' は Prolog のリスト操作の記号であり，'/' は組み込みの演算子である．このように，スタックはリスト形式で表現され，複数のスタックの先頭の要素が等しい場合，これを統合する．

3.2 構造化スタックの例

例えば，**図2**から得られた**表1**の LR パーズ表に従って，文："I open the door with a key." の解析を "with" まで終えた時，SGLR は以下のような構造化スタックを得る．

```
[7, [p(with)]|
  [9, [vp(v, np(det, n))], 4, [np(pron)], 0]/
  [12, [np(det, n)], 8, [v], 4, [np(pron)], 0]]
```

このスタックは，先頭の要素 7, [p(with)] の部分が統合され，表現上は木構造の形をとっている．しかし，

Prolog の処理系が Structure Sharing の機構を有する場合，スタックの底の部分とカテゴリの情報が共有化され，内部表現は図3に示すような構造となる．また，この例では，カテゴリの引数として解析木の情報が与えられているが，SGLR では引数の個数とその内容については任意である．

3.3 スタックの操作

以下に示すように，スタックの操作はすべて簡単なユニフィケーションによって実現できる．

- スタック S に状態番号 N とカテゴリの引数のリスト A を積み，それを S1 とする操作．

  S1=[N, A|S]

- スタック S の先頭に積まれたカテゴリの引数のリスト A を取り出し，残りのスタックを S1 とする操作．

  [_, A|S1]=S

- スタック S の先頭の状態番号 N を参照する操作．

  [N|_]=S

- 先頭の要素が等しい二つのスタック [N, A|S1] と [N, A|S2] を統合し，それを S とする操作．

  S=[N, A|S1/S2]

- スタック S の還元が枝の分岐点に及んだかどうかを判定し，分岐点である場合には，スタックから一つの枝 S1 を取り出し，残りの枝を B1 とする操作．

  B1/S1=S

4. SGLR の記述

4.1 記号の意味の定義

SGLR の記述を明確にするために，使用するアトムと変数の意味を以下のように定義する．

アトムの意味

- pt：単語の辞書引きによって得られる語彙カテゴリ名
- nt：非終端カテゴリ名
- n：状態番号
- r：各文法規則に割り付けられた番号
- p：(規則 r の右辺にあるカテゴリ数)−1

変数の意味

- N：状態番号
- A：カテゴリの引数のリスト

(1) s(s(Np, Vp)) → np(Np), vp(Vp).
(2) np(np(Np, Pp)) → np(Np), pp(Pp).
(3) np(np(Det, N)) → det(Det), n(N).
(4) np(np(N)) → n(N).
(5) np(np(Pron)) → pron(Pron).
(6) vp(vp(V, Np)) → v(V), np(Np).
(7) vp(vp(Vp, Pp)) → vp(Vp), pp(Pp).
(8) pp(pp(P, Np)) → p(P), np(Np).
(9) pron(pron) → ['I'].
(10) v(v) → [open].
(11) det(det) → [the].
(12) n(n) → [door].
(13) p(p) → [with].
(14) det(det) → [a].
(15) n(n) → [key].

図 2 曖昧性のある文法規則
Fig. 2 An ambiguous English grammar.

表 1 LR パーズ表
Table 1 An LR parsing table.

	det	n	p	pron	v	$	np	pp	s	vp
0	sh1	sh5		sh2			4		3	
1		sh6								
2			re5		re5	re5				
3						acc				
4			sh7		sh8			10		9
5			re4		re4	re4				
6			re3		re3	re3				
7	sh1	sh5		sh2			11			
8	sh1	sh5		sh2			12			
9			sh7			re1		13		
10			re2		re2	re2				
11			sh7/re8		re8	re8		10		
12			sh7/re6			re6		10		
13			re7			re7		10		

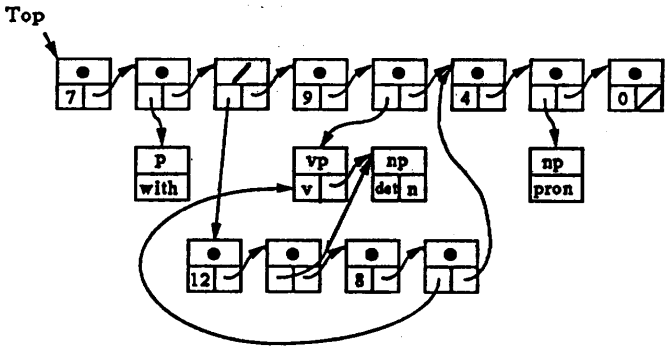


図 3 グラフ構造化スタック
Fig. 3 A graph structured stack.

- P：スタックの還元すべき要素数
- R：文法規則の番号
- S：スタック
- B：スタックの枝

- I：入力スタックのリスト
- O：出力スタックのリスト
- X：差分リストの尾部
- D：O-X の形の出力結果の差分リスト

ここで，A 0, n 1 のように名前の後に数字を付加した場合も，その意味は変わらないものとする．

4.2 カテゴリ節の記述

文法規則中のすべての語彙カテゴリ pt と非終端カテゴリ nt に対し，以下のようにカテゴリ節を与える．

```
pt([], _, O-O) :- !.
pt([S|I], A, O-X) :-
  [N|_]=S, pt_(N, S, A, O-O 1),
  pt(I, A, O 1-X).
nt(B/S, A, O-X) :- !,
  [N|_]=S, nt_(N, S, A, O-O 1),
  nt(B, A, O 1-X).
nt(S, A, D) :-
  [N|_]=S, nt_(N, S, A, D).
```

各カテゴリ節の第一引数は入力スタックのリスト，第二引数は辞書から得られたカテゴリの引数のリスト，第三引数は差分形式の出力スタックのリストである．

4.3 エントリ節

LR パーズ表の各エントリを一つのホーン節で記述し，そこでスタックを操作する．ただし，空白 (error) エントリはカテゴリごとに一つにまとめて記述する．

- shift エントリ：状態 n，カテゴリ pt のエントリが 'sh n 1' の時

```
pt_(n, S, A, [[n 1, A|S]|X]-X) :- !.
```

ここでは，スタックの先頭に辞書から得られたカテゴリの引数のリストAと新たな状態 n 1 を積んだ差分リストを出力する．

- reduce エントリ：状態 n，カテゴリ pt のエントリが 're r' の時

```
pt_(n, [_, A 0|S], A, D) :- !,
    re(S, p, r, A 0, O 1-[]),
    mg_st(O 1, O 0),
    pt(O 0, A, D).
```

ただし，p=0 の（規則 r の右辺のカテゴリが一つしかない）場合，節の二行目の還元節の呼び出しを，下の条件節の呼び出しに置き換える．

```
    cnd(r, S, A 0, O 1-[]),
```

reduce 操作では先読み語 pt を消費しないため，同じ語に対するカテゴリ節 pt を再帰的に呼び出す．

- コンフリクトのあるエントリ：状態 n，カテゴリ pt のエントリが 'sh n 1/re r 1' の時

```
pt_(n, S, A, [[n 1, A|S]|O]-X) :- !,
    [_, A 0|S 1]=S, re(S 1, p 1, r 1, A 0, O 1-[]),
    mg_st(O 1, O 0),
    pt(O 0, A, O-X).
```

また，エントリが 're r 1/re r 2' の時，

```
pt_(n, [_, A 0|S], A, D) :- !,
    re(S, p 1, r 1, A 0, O 1-O 2),
    re(S, p 2, r 2, A 0, O 2-[]),
    mg_st(O 1, O 0),
    pt(O 0, A, D).
```

上の二つの節 pt_ において，スタックの還元すべき要素数 pi=0 の場合には，reduce エントリと同様，還元節 re の呼び出しを条件節 cnd の呼び出しに置き換えることができる．

- accept エントリ：状態 n，カテゴリ $ (文の末尾) のエントリが acc の時，

```
$_(n, [_, A|_], _, [A|X]-X) :- !.
```

ここで，変数 *A* には，解析木の根 (sentence など) のカテゴリの持つ引数のリストが束縛される．

- error エントリ

```
pt_(_, _, _, O-O).
```

error エントリの各述語 'pt' の定義の末尾に置かれ，空の結果を返す．

- goto エントリ：状態 n，カテゴリ nt のエントリが 'goto n 1' の時，

```
nt_(n, S, A, [[n 1, A|S]|X]-X) :- !.
```

ここでは，reduce 後に得られたカテゴリの引数のリストAと新たな状態 n 1 を積んだ差分リストを出力する．

4.4 還元節の記述

還元節はスタックから *p* 個の要素を取り出し，規則Rに対する条件節（4.5 節参照）を呼び出す．

```
re([_, A 1|S], P, R, A, D) :- !,
  (p==1, !, cnd(R, S, (A 1, A), D)
  ; P 1 is P-1, re(S, P 1, R, (A 1, A), D)).
re(B/[_, A 1|S], P, R, A, O-X) :-
  (p==1, !, cnd(R, S, (A 1, A), O-O 1)
  ; P 1 is P-1, re(S, P 1, R, (A 1, A), O-O 1)),
  re(B, P, R, A, O 1-X).
```

4.5 条件節の記述

条件節は規則ごとに記述する．例えば，規則 r が次のように与えられている時，

```
nt(A 1, A 2)→cat1(A 11, A 12), cat 2(A 21, A 22),
```

```
{aug 1(A 11, A 21, A 1),
 aug 2(A 12, A 22, A 2)}.
```

条件節を以下のように記述する.

```
cnd(r, S, ([A 11, A 12], [A 21, A 22]), D) :-
  (aug 1(A 11, A 21, A 1),
   aug 2(A 12, A 22, A 2)), ! ,
  nt(S, [A 1, A 2], D).
```

条件節は規則中の任意の場所に記述された補強項を順次評価し，そのすべてが成功した場合，規則左辺のカテゴリ nt に対するカテゴリ節を呼び出す．それ以外は，バックトラックによって以下の節が呼び出される.

```
cnd(_, _, _, O-O).
```

この節は条件節の末尾に置かれ，常に空の結果を返して成功する.

4.6 統合節の記述

統合節は第一引数に与えられたリスト中のスタックを比較し，先頭の状態番号Nと，カテゴリの引数の情報Aが等しいものを統合する．その結果得られたスタックのリストを第二引数に返す.

```
mg_st([], []) :- ! .
mg_st([[N, A|S]|I], [[N, A|B]|O]) :-
    mk_br(N, A, S, I, B, O 1),
    mg_st(O 1, O).
mk_br(_, _, S, [], S, []) :- ! .
mk_br(N, A, S, [[N, A|S 1]|I], B/S 1, O 1) :- ! ,
    mk_br(N, A, S, I, B, O 1).
mk_br(N, A, S, [S 1|I], B, [S 1|O 1]) :-
    mk_br(N, A, S, I, B, O 1).
```

4.7 SGLR の起動

2章で述べたように，SGLR は入力文から得られるトップレベルのゴール列を呼び出すことによって起動する．3.2 節で示した例文に対するトップレベルのゴール列は，次のようになる.

```
?-pron([[0]], [pron], O 1-[]),
  mg_st(O 1, I 2),
  v(I 2, [v], O 2-[]),
  mg_st(O 2, I 3),
    ...
  n(I 7, [n], O 7-[]),
  mg_st(O 7, I 8),
  $(I 8, [], O 8-[]).
```

最初のゴール pron はカテゴリ節とマッチし，最初の入力語 'I' に対する解析を行う．その結果を統合節 mg_st によって統合し，次の語 v の解析に渡す.

また，多品詞語も容易に扱える．例えば，二つ目の入力語 open が n (n) というカテゴリを持つ時，三行目のゴールを次に置き換えるだけでよい.

```
v(I 2, [v], O 2-X), n(I 2, [n], X-[]),
```

5. SGLR と富田法との相違

SGLR は次の点で富田法と異なる.

（1） 富田法では，先頭の状態が等しいスタックを統合するが，SGLR では状態だけでなく，カテゴリの引数の同一性も調べた上で統合する.

（2） 富田法では，解析木を統合して統合共有林 (Packed Forest) を作るが，SGLR では個々の解析木を別々に保持する.

この違いは以下の理由による.

（1） 構文解析の途中で，文のある部分を支配する同一名のカテゴリが複数得られた時，それらのカテゴリが支配する部分木の構造が異なると，カテゴリの引数の情報が一般的に異なる．したがって，仮にこれらを統合 (Pack) したとしても，引数の内容を参照する際に再び分離 (Unpack) する必要が生じ[5]，統合，分離の操作がオーバヘッドとなる.

（2） Prolog では，統合共有森を，assert などの副作用を用いずに実現するのは困難である．副作用を用いて実現すると，解析の効率が大幅に低下する．これらの事実を次の実験によって示す.

6. 実　　験

SGLR の効率性を示すために，以下の条件に基づいて文の解析時間を測定し，SAX との比較を行うとともに，富田法のオーバヘッドを評価した.

- 計算機環境：Sun-3/260 SUN UNIX 4.2
- 使用言語：Quintus Prolog (Release 2.2)
- 文法：規則数/399 (CFG 換算/554) 個，非終端記号/44 個，語彙記号 (preterminal)/35 個
- 辞書：653 語
- プログラム領域の大きさ
  SGLR/2927 (KB), SAX/772 (KB)
- 測定値：全解探索に要する時間 (Stack Shift, Garbage Collection の時間を除く)
- 解析文：105 文

実験に用いた文法は，Hornby の文型パターンを用いて曖昧性を削減する補強項が組み込まれている[12]．また，実験に用いた英文は，基本5文型，命令文，疑

問文，関係詞節，従属節などを含み，単語数は4語から33語，解析で得られた木の数は1個から36個であった．

6.1 実験結果

図4に入力文の長さ（単語数）と解析時間の関係，**図5**に入力文の長さと解析木の数の関係を示す．また，105文の全解析時間，解析木一つ当たりの平均解析時間，および一単語当たりの平均解析時間を**表2**に示す．この実験により，SGLR は解析の時間的効率性において，SAX を凌ぐ性能を有することが明らかとなった（この実験に用いた SAX は，規則に対する冗長な識別子の割り付けの削減と，ダブルハッシュによる述語検索機能の利用により，初期の SAX[7] と比べて3倍程度の解析速度を得ると同時に，3割程度の空間的効率化を図っている）．

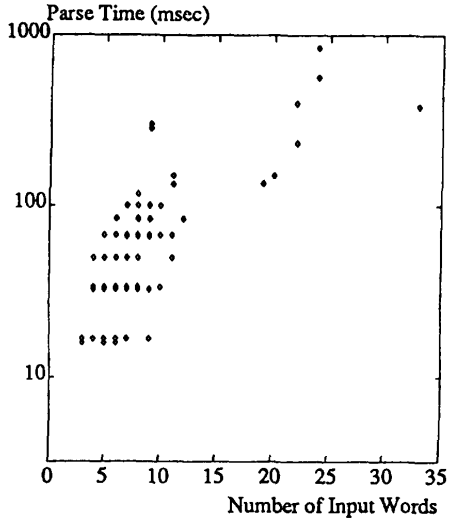


図4 解析時間
Fig. 4 Parse time.

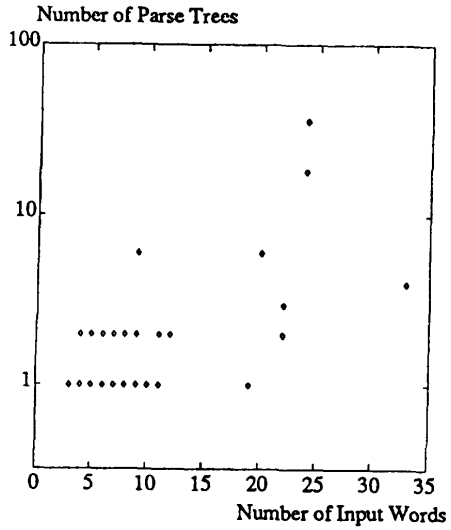


図5 解析木の数
Fig. 5 Number of parse trees.

また，参考のため SGLR においてスタックの統合を行わず，再計算を許した場合の解析時間と，文法から補強項を取り除いた場合の解析時間を**表3**に示す．この結果は，再計算による効率の低下と，CFG の構文解析における解析木の組合せ的爆発による効率の低下を顕著に示している．

6.2 富田法の評価

ここでは，富田法に基づく DCG パーザを Prolog 上に実現する際のカテゴリの統合，分離のオーバヘッドと，統合共有森の Prolog 上での実現のコストを評価する．SGLR では，統合節（4.6節）を拡張することにより，富田法に従ったスタックの統合を実現できる（先頭の状態が等しい二つのスタックのカテゴリ情報 A1, A2 を A1/A2 の形に統合する）．統合したカテゴリの内容を条件節（4.5節）で参照する場合には，それを再び分離する．上の105文に対して，一度統合したカテゴリを二度と分離しない場合と，すべてを分離した場合の解析時間を測定した．その結果，前者が 7,686 (msec)，後者が 9,950 (msec) であった．SGLR の解析時間との比率はそれぞれ 0.94 と 1.22 である．前者の値はスタック数の減少による効率の向上を示し，後者の値はカテゴリの統合，分離のオーバヘッドの影響を示している．一般に，DCG のカテゴリの引数は，そのカテゴリが支配する木構造ごとに異なるため，統合したカテゴリの引数を参照する際に，必ず分離操作が必要となる．したがって，DCG の解析には，後者の解析時間を要し，富田法は SGLR よ

表2 SGLR と SAX の解析時間
Table 2 Parse time of SGLR and SAX.

	SGLR (msec)	SAX (msec)
全解析時間	8,183	11,950
木一つ当たり	40.1	58.6
一単語当たり	9.6	14.0

表3 スタック不統合，補強項不適用の解析時間
Table 3 Parse time without merge and without augmentations.

	Merge なし (msec)	補強項なし (msec)
全解析時間	63,523	4,213,748
木一つ当たり	311.4	420.7
一単語当たり	74.3	4934.1

り効率が悪い.

また，富田法の統合共有森を Prolog の副作用を利用して実現した場合のオーバヘッドを評価した．上の105 文の解析で得られる統合共有森のすべての節点を，assert するのに要した時間は，19,448 (msec) であった．これは SGLR の解析時間の 2.4 倍に相当する．このように，副作用を用いた場合，統合共有森の構築は解析の効率化に寄与しない.

7. SGLR の課題

SGLR は補強項の実行に関して，次のような課題が残されている.

- 補強項の全解を探索しない.
  SGLR の実行はバックトラックを伴わないため，補強項の解は一つしか得られない.
- カテゴリ間の情報の流れがボトムアップに限定される.
  SGLR では，SAX と同様環境のコピーの問題がある[7]．これを回避するためには，DCG の引数による情報の流れをボトムアップの方向に限定する必要がある.
- 補強項の評価のタイミングが遅れる．補強項が規則中のどの場所に記述されていても，条件節で一つにまとめて評価される.

8. おわりに

本研究では，DCG 文法に対する一般化 LR パーザ SGLR を論理型言語 Prolog に実現し，その有用性を示した．SGLR は同じ DCG のパーザとして効率性に優れていることで知られる SAX と比較して，解析の時間的効率において SAX を凌ぐ性能を有することが明らかとなった．SGLR の構文解析の方式はボトムアップ横型であり，これは並列処理との整合性が良い．比較的簡単な変更により，SGLR の記述を並列論理型言語 GHC の記述に置き換えることができ，並列一般化 LR パーザを実現できる．ただし，並列化に際しては，プロセスの同期やプロセッサ間の通信のオーバヘッドその他の問題を検討する必要があると思われる.

LFG のような単一化文法に対しても，これを DCG 形式で表現することにより，SGLR が効率のよいパーザを提供することが期待できる．ただし，高い効率を得るためには，素性情報とその操作の効率的な実現方法をよく検討する必要がある.

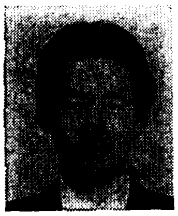
また，LangLAB や YAP に見られる外置の扱いのように，自然言語の諸現象をより的確に記述するための文法記述形式とその解析方式の開発も今後の課題である.

謝辞 本研究を進めるにあたり，SAX の処理系を快く提供してくださった松本裕治氏に感謝いたします．また，本研究に対する貴重な御意見をいただいた田中研究室の皆さんに感謝いたします.

参考文献

1) Tomita, M.: *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*, Kluwer Academic Publishers, Boston, Massachusetts (1985).
2) Knuth, D. E.: On the Translation of Languages from Left to Right, *Inf. Control*, Vol. 8, No. 6, pp. 607-639 (1965).
3) Johnson, M.: The Computational Complexity of Tomita's Algorithm, *Proc. of Parsing Technologies*, pp. 203-208 (1989).
4) 田中穂積：自然言語解析の基礎，産業図書，東京 (1989).
5) Tomita, M.: An Efficient Augmented-Context-Free Parsing Algorithm, *Computational Linguistics*, Vol. 13, Nos. 1-2, pp. 31-46 (1987).
6) Matsumoto, Y. et al.: Bup—A Bottom-up Parser Embedded in Prolog, *New Generation Computing*, Vol. 1, No. 2, pp. 145-158 (1983).
7) 松本裕治，杉村領一：論理型言語に基づく構文解析システム SAX，コンピュータソフトウェア，Vol. 3, No. 4, pp. 4-11 (1986).
8) 徳永健伸ほか：自然言語解析システム LangLAB，情報処理学会論文誌，Vol. 29, No. 7, pp. 703-711 (1988).
9) Nilsson, U.: AID: An Alternative Implementation of DCGs, *New Generation Computing*, Vol. 4, No. 4, pp. 383-399 (1986).
10) Kindermann, J.: An Extension of LR-Parsing for Lexical Functional Grammar, *Natural Language Parsing and Linguistic Theories*, Reyle, U. and Rohrer, C. (eds.), pp. 131-148, D. Reidel Publishing Company, Dordrecht, The Netherlands (1987).
11) 森 達也：論理型言語による構文解析法 YAP について，情報処理学会論文誌，Vol. 29, No. 5, pp. 480-487 (1988).
12) 高倉 伸：Prolog による英語のボトムアップ構文解析，東京工業大学工学部卒業論文 (1984).

(平成2年 5 月 18 日受付)
(平成2年 12月 18 日採録)



沼崎 浩明（正会員）

1962年生．1986年東京工業大学工学部情報工学科卒業．1988年同大学院修士課程修了．同年(株)三菱総合研究所入社．1989年東京工業大学大学院博士課程入学．自然言語処理，並列処理に興味を持つ．電子情報通信学会会員．

田中 穂積（正会員）

昭和39年東京工業大学理工学部制御工学科卒業．昭和41年同大学修士課程修了．同年電気試験所（現，電子技術総合研究所）入所．昭和58年東京工業大学工学部情報工学科助教授．昭和61年同大学教授となり現在に至る．工学博士．人工知能，自然言語処理の研究に従事．電子情報通信学会，認知科学会，日本ソフトウェア科学会，人工知能学会，計量国語学会各会員．