Incomplete Stack を用いた並列一般化 LR パーザ PGLR について†

沼崎浩明†† 田中穂積††

筆者らは既に，並列一般化 LR 構文解析アルゴリズムの効率化を図るために，重複計算を避けると共に，プロセスの同期を低減する手法として，Incomplete Stack (以下 IS と呼ぶ) というデータ構造を提案している．これに対し，本研究ではプロセスの同期を完全に排除することを目的として，New Incomplete Stack (以下 NIS と呼ぶ) の形式を提案している．われわれは，この NIS を用いて並列一般化 LR 構文解析を行うパーザを PGLR (Parallel Generalized LR Parser) と呼んでいる．本論文では，PGLR の動作原理を明らかにするとともに，NIS を用いることが，解析速度の向上に大きく寄与することを実証している．PGLR は次の特徴を有する．(1)与えられた入力文を左から右へと解析する．(2)文の構文的な曖昧性を並列に解析する．(3)重複計算をすべて回避する．(4)NIS を用いて，非同期な並列構文解析を行う．(5)文法に与えた制約により，漸進的な曖昧性解消を行うことができる．効率性を実証するために，英語を例にとり，解析時間の計測を行った．実験には規則数 399 の DCG と単語数 653 の辞書を用いている．実験の結果，NIS は NIS を用いない場合と比較して，最大 3.1 倍の解析速度の向上をもたらすことを確認した．

1. はじめに

LR 法[2]は，LR パーズ表とスタックを用いて文を解析するアルゴリズムである．このアルゴリズムを一般の文脈自由文法に適用したものを，われわれは一般化 LR 法と呼んでいる．一般化 LR 法は文の構文的曖昧性を扱えるが，文の構文的曖昧性はパーザのアルゴリズムが横型探索に基づくものであれば，並列に解析できる．横型探索に基づく一般化 LR 法として，冨田法[3),4)]がよく知られている．冨田法は，複数のプロセスが同じ部分木を作る際に生ずる重複計算を，スタックの統合により解消している．この手法は，逐次処理において高い効率性を示すことが実証されているが，並列処理においてはいくつかの問題があることが指摘されている[5]．その理由は，スタックの統合がプロセスの同期を必要とすることによる．すなわち，本来，並列に実行できるプロセスを，スタックの統合のために，シフト動作で待ち合わせ，同期をとる必要があり，この待ち合わせが解析時間の増大を招くからである．

本研究では，この問題を解決するための手法として NIS (New Incomplete Stack) という新たなデータ構造を提案している．NIS とは，未定義部を含む枝を許す木構造化スタックのことである．このデータ構造は，非同期の並列構文解析を実現することを目的として，先に提案した Incomplete Stack の考え[1]を発展させたものである．ここでいう非同期の意味は，並列に走る一つ一つのスタックの操作の流れを，中断することなく実行できるということであり，プログラムとして記述されたプロセスが同期を必要としないという意味ではない．

われわれは，この NIS を用いた並列一般化 LR パーザを PGLR (Parallel Generalized LR Parser) と呼んでいる．本論文では，PGLR の動作原理を明らかにするとともに，NIS を用いることが，解析速度の向上に大きく寄与することを実証している．

われわれは，並列パーザ PGLR を並列論理型言語 GHC[6] を用いて記述しており，文法の記述形式として DCG を採用している．ただし，補強項のプログラムを GHC で記述する点において本来の DCG とは異なる．しかし，この点については本論文では議論しない．

本研究は，峯らの手法[7]および，安留らの手法[8]と関連している．峯らの手法は純粋な CFG の並列構文解析の効率化をプロセス数と解析時間の観点から徹底化している．また，安留らの手法は単一化文法の枠組に基づき，文の意味的曖昧性を並列処理によって解消する手法を提案している．

以下に本論文の構成を示す．

第 2 章では，PGLR の構成とアルゴリズムを示す．第 3 章では，PGLR による解析の具体例を示す．第 4 章では，PGLR の計算量について考察する．第 5 章で

† A Parallel Generalized LR Parser with Incomplete Stacks 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

は，NIS の有用性を示すための実験として，PGLR による文の解析時間を測定する．最後の第6章では，本論文の結論を述べる．

2. PGLR の構成とアルゴリズム

PGLR は文の曖昧性を並列に処理するパーザである．そのアルゴリズムは，従来の LR 法[2]の上に文の曖昧性の扱いと，重複計算の回避機構とを加えたものとして成り立ち，入力語の並びに応じて組み合わされる5種類のプロセスにより構成されている．以下に，各プロセスの処理内容を示す．

- Category プロセス：各入力語のカテゴリを先読みとする処理を統括するプロセス．文法中のすべてのカテゴリに対して一つずつ与える．
- Entry プロセス：LR パーズ表の各エントリの内容に応じたスタック操作を行うプロセス．すべてのエントリに対して一つずつ与える．エントリの種類に応じて，Shift, Reduce, Shift-Reduce Conflict, Reduce-Reduce Conflict, Goto, Accept, Error の7種類のプロセスがある．
- Reduce プロセス：Reduce Entry プロセスから呼び出され，スタックから指定された数の要素を取り出すプロセス．
- Condition プロセス：文法規則に記述されている制約条件(補強項) を評価し，条件に反する解析を終了させるプロセス．また，規則右辺のカテゴリの引数の情報から，規則左辺のカテゴリの引数の情報を生成する．
- Merge_Stack プロセス：NIS を作り，共通の先頭要素を持つスタックを統合するプロセス．このプロセスは，文の曖昧性により複数のスタックを受け取った時のみ NIS を生成し，曖昧性がない時は何もしない．

図1には，LR 法の各スタック操作に対応したプロセスの呼び出し関係を示す．また，付録に GHC のサブセット言語 KL1 で記述した各プロセスのアルゴリズムを示す．

2.1 PGLR の動作原理

構文解析は，Category プロセスの列を起動することにより行う．この列は入力語の並びに対応させ，一本のストリームで連結する．ストリームには左端から一つの初期スタック[0]を流す．各 Category プロセスはデータ駆動により動作し，入力ストリームからスタックを受け取ると Entry プロセスを起動してそれに必要な操作を施す．Entry プロセスは LR パーズ表のエントリが定めるスタック操作を行う．その操作の内容は，現在のカテゴリ名とスタックトップの状態番号により決まる．Category プロセスはその処理結果を出力ストリームに流す．

このように，ストリームの上を流れるスタックは，解析途中の情報を保持し，右端に到達したスタックは，解析結果の情報を持つ．

2.2 曖昧性の扱い

曖昧性を有する文を解析する時，少なくとも一回は Shift-Reduce Conflict，あるいは，Reduce-Reduce Conflict の Entry プロセスが呼び出される．これらのプロセスは，一つの入力スタックに複数の操作を施

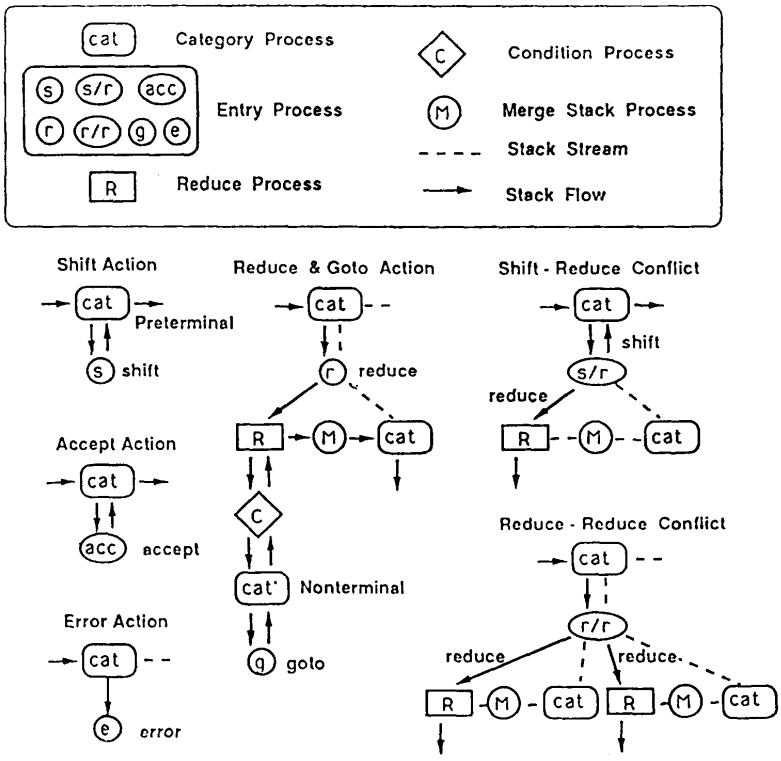

図1 PGLR のプロセス
Fig. 1 PGLR processes.

し，その結果として複数のスタックを出力する．以後，ストリームを流れるスタックは複数となる．

すべてのプロセスはデータ駆動により並列に動作するため，ストリームに流れるすべてのスタックは，並列に処理される．

2.3 制約の適用による曖昧性の削減

文法に与えた制約の適用はConditionプロセスが行う．Conditionプロセスはレデュース操作によりスタックから取り出された情報を，制約評価のプログラムに渡す．制約評価のプログラムはユーザ定義のGHCプログラムであり，評価に用いられる情報は，DCG文法のカテゴリの引数として受渡しされる任意の情報である．

Conditionプロセスは制約の評価に成功した時，規則左辺の非終端カテゴリに対応するCategoryプロセスを呼び出し，そのカテゴリの引数の情報を渡す．Categoryプロセスは Goto Entryプロセスを呼び出してその情報と新たな状態番号とをスタックに積む．

制約の評価に失敗した時，何も結果を返さずに処理を終了する．

2.4 NISを用いた重複計算の回避機構

LR法では，解析の途中，特定の場所を同じ先頭要素を持つスタックが通過する時，重複計算が生ずる．この重複計算は，スタックの先頭要素を統合することにより回避できる[3]．PGLRでは，Merge_Stackプロセスをすべてのカテゴリプロセスの間に配置し，そこを通過するスタックを統合する，スタックの統合は，スタックの流れを止めずに行うことができる．冨田法では，スタックを統合する際に各スタックのシフト動作で同期を取るが，PGLRでは，NISと呼ぶデータ構造を用いて同期を取らずに処理を進めることができる．

NISは，未定義部を含む枝を許す木構造化スタックのことをいう．NISの形式をBNF記法を用いて定義する．

```
〈NIS〉::=
    [0] |
    [〈State Number〉,
                〈Argument〉 | 〈Branch〉]
〈Branch〉::=
    〈NIS〉 |
    〈Unbounded〉 / 〈NIS〉 |
    〈Branch〉 / 〈NIS〉
```

ここで，〈State Number〉はLRパーズ表の状態番号，〈Argument〉はスタックに積まれたDCGのカテゴリが持つ引数のリスト，〈Unbounded〉は未定義部を表すGHCの変数である．

Merge_Stackプロセスは，入力ストリームから受け取ったスタックに未定義部を付加してNISを作り，これを出力ストリームに流す．例えば，

[7, P, 12, NP, 8, V, 4, NP, 0]

というスタックから作られるNISは，

```
[7, P| Unbounded
    /[12, NP, 8, V, 4, NP, 0]]
```

となる．これを図示すると，**図2**のようになる．Merge_Stackプロセスは，NISの未定義部を保持し，次のスタックを受け取った時，そのスタックの先頭要素が，既に出力しているNISの先頭要素と等しければ，そのスタックをNISの未定義部に束縛する．例えば，Merge_Stackプロセスは上のNISを出力した後に，

[7, P, 9, VP, 4, NP, 0]

というスタックを受け取ると，これをNISに統合するため，そのスタックから共通の先頭要素を取り除いた部分を，未定義部と束縛する．

```
Unbounded =
    Unbounded' / [9, VP, 4, NP, 0]
```

この時，新たな未定義部Unbounded'を付加する．この結果，先に出力されたNIS（ここでは，まだ次のプロセスにより処理されていないとする）は，二つの枝と一つの未定義部を持つ次のようなNISとなる．

```
[7, P | Unbounded'
      / [9, VP, 4, NP, 0]
      / [12, NP, 8, V, 4, NP, 0]]
```

Merge_Stackプロセスは，入力ストリームが閉じることにより終了する．その際，自分が出力したすべてのNISの未定義部を[ ]で具体化する．このようにして，スタックの流れを止めることなく重複計算を回避できる．

先に提案しているIS[1]とNISとの違いは，スタックの枝の部分にある．[State, Argument | Branch]というスタックから得られるISは[State, Argument | Unbounded]であり，一方，NISは[State,

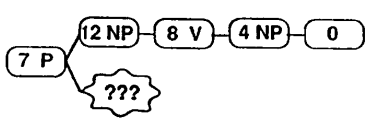

図2 NISの例
Fig. 2 Example of NIS.

Argument | Unbounded / Branch] である．先の手法では，IS の未定義部 Unbounded の具体化を，スタックの統合時に行うこととしていたため，レデュース操作が未定義部に及ぶと，本来レデュースできる Branch の要素を取り出せず．その処理を中断する必要があった．一方，本手法では NIS の枝の部分は最初から Branch を含んでおり，レデュース時の無駄な中断は生じない．

2.5 NIS の操作

NIS の操作は，基本的には通常の木構造化スタックと同じである．したがって，NIS を受けとったプロセスは，未定義部がいつ具体化されるかにかかわりなく，処理を進めることができる．ただし，レデュース操作の時，NIS の未定義部から要素を取り出そうとするプロセスは，その部分が具体化されるまで実行を待ち合わせる．例えば，前節の NIS の先頭要素 7, P を取り出した後のレデュース操作は，二つの枝 [12, NP, 8, V, 4, NP, 0], [9, VP, 4, NP, 0] と，一つの未定義部 Unbounded' とを独立に扱うが，未定義部のレデュース操作は，その具体化を待って実行する．ただし，この中断はスタックの処理の流れの中断ではないので，時間的効率には影響しない．

3. PGLR による解析の具体例

PGLR による構文解析の具体例を示す．図 4 (a)〜(m) は，**図 3** の文法と，それから得られる**表 1** の LR パーズ表を用い，"I saw men in the park." という文を解析する際の PGLR のプロセスの振舞いを示している．この文法は，カテゴリの引数として，解析木の情報を有し，各規則は補強項による制約を持たない．

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

	det	n	p	pron	v	$	np	pp	s	vp
0	sh 1	sh 5		sh 2			4		3	
1		sh 6								
2			re 5		re 5	re 5				
3						acc				
4			sh 7		sh 8			10		9
5			re 4		re 4	re 4				
6			re 3		re 3	re 3				
7	sh 1	sh 5		sh 2			11			
8	sh 1	sh 5		sh 2			12			
9			sh 7			re 1		13		
10			re 2		re 2	re 2				
11			sh 7/re 8		re 8	re 8		10		
12			sh 7/re 6			re 6		10		
13			re 7			re 7		10		

```
(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)           -->  [saw].
(11)  n(n)           -->  [men].
(12)  p(p)           -->  [in].
(13)  det(det)       -->  [the].
(14)  n(n)           -->  [park].
(15)  $([])          -->  ['.'].
```

図 3 曖昧性のある文法
Fig. 3 An ambiguous English grammar.

プロセスの左肩に記した数字はそのプロセスが動作する時刻を示している．時刻の単位は，ある時刻で実行可能なすべてのプロセスを 1 ステップ進める処理を 1 単位とし，これを cycle と呼んでいる．以下に，**図 4** に示す PGLR の動作について説明する．

図(a)：最初に起動するプロセスの列．

各プロセスはストリームを介して相互に連結している．構文解析は，初期スタック[0]をストリームに送ることにより開始する．これを受け取ったカテゴリプロセス pron は，sh 2 の操作を実行し，その結果として得た新たなスタックをストリームに送る．ストリームに送られたスタックは次のカテゴリプロセスの入力となる．

図(b)：カテゴリプロセス v における re 5, goto 4, sh 8 の操作．

図(c)：カテゴリプロセス n における sh 5 の操作．

図(d)：re 4 の操作の後，Conflict のあるエントリ sh 7/re 6 を呼び出す．Conflict の原因は，前置詞 "in" で始まる前置詞句に掛かり受けの曖昧性があることによる．

図(e)：Conflict の後のプロセスの並列実行．

時刻 26 において，二つのプロセス (Merge_Stack プロセスとレデュースプロセス) が並列に実行される．

図(f)：Merge_Stack プロセスによる NIS の生

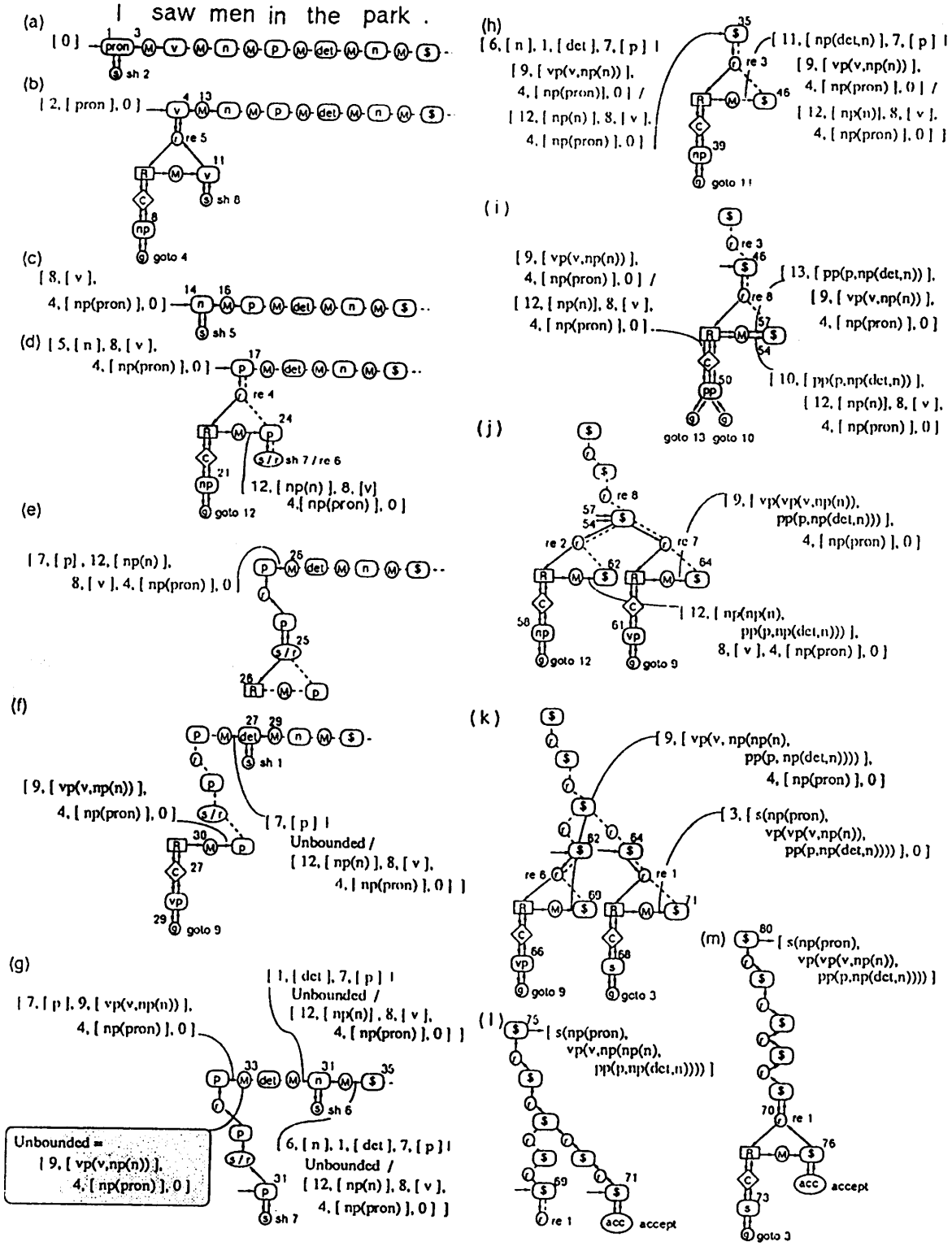

図 4 PGLR の動作
Fig. 4 Behavior of PGLR.

成.

図(g):スタックの統合.

後続するスタックは,Category プロセス p の下で先行するスタックと同じ sh 7 の操作を受け,Merge_Stack プロセスに送られる.Merge_Stack プロセスは,このスタックを先行するスタックの変数 Unbounded に束縛する.その結果,先行するスタックは木構造化スタックとなる(図(h)の入力スタック).

図(h):統合されたスタックのレデュース操作.

図(i):統合されたスタックの分離.

re 8 の操作によって,木構造化スタックの統合部分が取り除かれ,時刻 50 で動作する Category プロセス PP において,二つのスタックに分離する.

図(j)〜(m):二つのスタックに対する並列処理.

解析結果として,最終的に二つの解析木を得ている.

4. PGLR の計算量

PGLR の計算量は,実際には GHC の処理系に依存するが,ここでは,プロセッサ台数が十分に大きく,共有メモリ型の並列計算機の上に Shared-Structure の機構を備える理想的な処理系を仮定する.以下の文中の記号(A)〜(D)は,付録に示した PGLR のアルゴリズム中の対応する行を示している.

4.1 時間計算量

PGLR は複数のプロセス間で生ずる重複計算を避ける際に,スタック操作の流れを止めない非同期な並列構文解析を実現している(A).したがって,PGLR の解析に要する時間は,一つの解析木を構築する時間に相当する.一つの解析木を構築する手続きには,本来の LR 法の手続きに加えて,NIS の構築と制約評価の手続きが必要である(B).

Merge_Stack プロセスによる NIS の構築には,入力スタックと,既に出力している最大数 C 個(C は LR パーズ表の状態数)のスタックとの先頭要素の照合(C),スタックの統合(D),または新たな NIS の生成に要する,ユニフィケーションの操作(A)が必要である.しかし,これらの処理は入力文の長さに依存しない一定の時間内で終了する.制約評価の手続きは任意であるが,入力文の長さに依存しない一定の時間内に終わる手続きを与えたとすれば,PGLR の解析に要する時間のオーダは,入力文の長さ n にして,本来の LR 法と等しく $O(n)$ となる(文献 9) Theorem 5.13 参照).

一方,冨田法のように NIS を用いない場合は,各シフト操作で同期を必要とし,$O(n^2)$ の解析時間を要することが知られている[5].

4.2 空間計算量

PGLR で複数の解析木を個別に出力する.複数の解析木に共通する部分木は処理系の Shared-Structure の機構により共有されるが,冨田法で提案されているような Packed Forest を作らない(この点については,SGLR パーザ[10]と同一である).このため,空間計算量およびプロセス数は入力長に対して指数関数的に増大する.しかし,文法の適用に適切な制約を設けることにより,空間計算量は抑制できると考えている.PGLR が Packed Forest を作らない理由は,制約評価の際に統合したカテゴリを分離する操作が必要となり,この操作が時間的効率の低下を招くことを確認しているためである[10].

5. 実　　験

PGLR の性能を評価し,NIS の有用性を実証する実験として,シミュレーションにより,文の解析時間を評価した.実験は次の条件の下に行った.

- 処理系:Prolog で記述された Flat GHC の処理系
- 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
- 解析文:57 文

本実験で計測した文の解析時間の値は,プロセッサ数が十分に多く(PGLR が生成するプロセス数より多く),共有メモリによりプロセッサ間の通信コストのかからない理想的な並列計算機における値とみなせる.すなわち,この値は理想値であり,実際の並列計算機を用いて得た実測値の評価基準に用いることができる.ここで,時間の単位は各時刻で実行可能なすべてのプロセスを,1 ステップ(GHC ゴールの 1 reduction)進める処理を 1 単位とし,これを cycle と呼ぶ.測定値として,解析時間のほかに台数効果も求めている.台数効果とは逐次処理に要する時間を並列処理に要した時間で割った値である.ここでは,(ゴールの全 reduction 数)/(全 cycle 数)としている.これらの値を,NIS を用いた場合と用いない場合のそれぞれについて測定し,NIS の効果を調べた.その結果を図 5 のグラフに示す.このグラフは両者の解析時間を示している.文の長さに対する解析時間を表している.

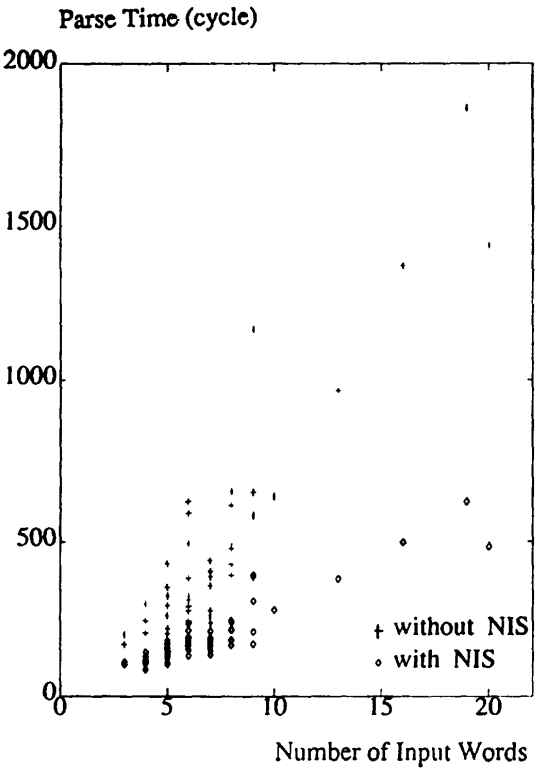

図 5 解析時間
Fig. 5 Parse time.

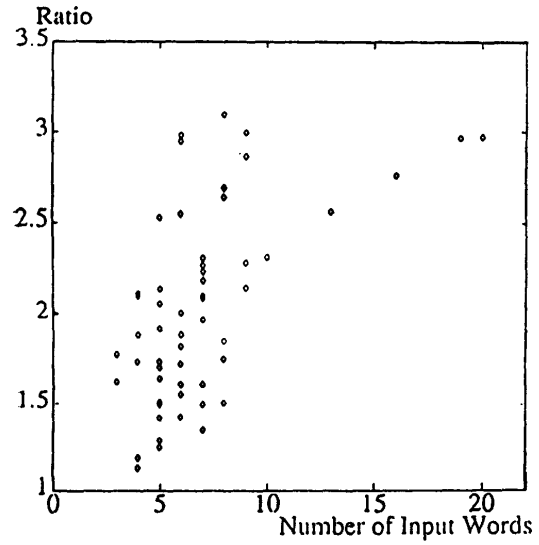

図 6 二つの解析時間の比率
Fig. 6 Ratio of two parse time.

NISを用いた場合は，用いない場合と比較して平均で2.2倍，最大で3.1倍の解析速度を得ている．図6に示すように，両者の速度の比は文の長さに応じて増大する傾向にある．この事実は，両者の入力長に対する解析時間のオーダが異なることを示している．

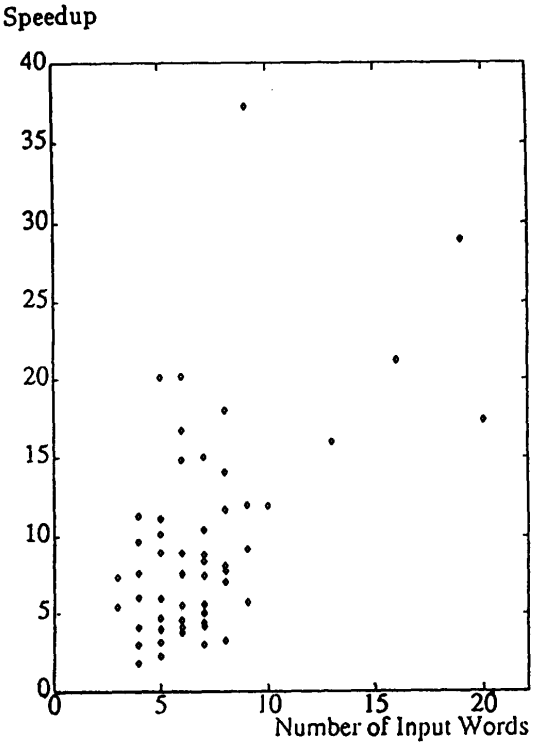

図 7 台数効果
Fig. 7 Speedup.

図7は，NIS を用いた場合の台数効果を示している．このグラフは，多数の曖昧性が生じる長い文ほど，高い台数効果を得ることができるという，期待どおりの結果を示している．台数効果の値は，NIS を用いた場合が最大で 37 倍，平均で 9.4 倍の高い値を得たのに対し，NIS を用いない場合は，最大で 12 倍，平均で 4.1 倍であった．

これらの実験結果は，NIS の有用性を示している．

6. おわりに

本論文では，NIS を用いた並列一般化 LR パーザ PGLR の動作原理を明らかにし，その有用性を実証した．NIS は，「効率化のため，データに不完全な部分を残したまま仕事を先に進める．」という考え方に基づいて考案したデータ構造である．この考え方は，従来の逐次アルゴリズムを並列アルゴリズムとして見直す際に，無駄な同期を廃する方法を考案するための一般性のある原則であると考えられる．他の構文解析のアルゴリズムを並列化する際にも，この考えを応用することにより，本質的でない同期は避けることができると思われる．

NIS は未定義部分を許すデータ構造として，チャート法の活性弧を連想させる．しかし，NIS は重複計算を避ける際の同期を廃するためのデータ構造であり，解析木を得るために必要不可欠なものではないが，チャート法の活性弧は，将来，構築される可能性のある部分木の先頭のカテゴリを待つデータ構造であり，解析に必要不可欠である．その意味において，それぞれの役割は異なっている．

PGLR は文法の記述をすべてユーザに任せており，文法の記述に不備がある場合，プログラムがデッドロックを起こすことがある．デッドロックを未然に防ぐことは困難であるが，デッドロックの危険性のある規則を予測することはある程度まで可能である．PGLR を実際に運用するには，ユーザにデッドロックの危険性のある規則を指摘し，注意を促す機構を設けるべきである．今後の課題は，本方式を実際の並列計算機上に実現するために，プロセス間の通信量の低減と，負荷の均等化を図るための，負荷分散方式を開発することである．

謝辞 本研究を進めるにあたり，日頃からご協力をいただいた田中研究室のみなさんに感謝いたします．また，Prolog 上の GHC の処理系を提供していただいた ICOT に感謝いたします．

参 考 文 献

1) Numazaki, H. and Tanaka, H.: A New Parallel Algorithm for Generalized LR Parsing, *Coling-90*, Vol. 2, pp. 305-310 (1990).
2) Knuth, D. E.: On the Translation of Languages from Left to Right, *Information and Control*, Vol. 8, No. 6, pp. 607-639 (1965).
3) Tomita, M.: An Efficient Augmented-Context-Free Parsing Algorithm, *Computational Linguistics*, Vol. 13, No. 1-2, pp. 31-46 (1987).
4) 田中穂積：自然言語解析の基礎，p. 98，産業図書，東京 (1989).
5) 峯 恒憲，谷口倫一郎，雨宮真人：文脈自由文法の並列構文解析，情報処理学会自然言語処理研究会，Vol. 73, No. 1, pp. 1-8 (1989).
6) Ueda, K.: Guarded Horn Clauses, *The Logic Programming Conference*, *Lecture Notes in Computer Science 221*, Wada, E. ed., pp. 168-179, Springer-Verlag, Berlin Heidelberg (1986).
7) Mine, T. et al.: An Efficient Parallel Parsing Algorithm for Context-Free Grammars, *Proc. of Pacific Rim International Conference on Artificial Intelligence*, pp. 239-244 (1990).
8) Yasutome, S. et al.: Parallel Semantic Disambiguation for Unification-Based Grammars, *Proc. of Pacific Rim International Conference on Artificial Intelligence*, pp. 239-244 (1990).
9) Aho, A. V. and Ulman, J. D.: *The Theory of Parsing, Translation, and Compiling Volume 1: Parsing*, p. 395, Prentice-Hall, Englewood Cliffs, New Jersey (1972).
10) 沼崎浩明，田中穂積：SGLR：逐次型一般化 LR パーザの Prolog による実現，情報処理学会論文誌，Vol. 32, No. 3, pp. 396-403 (1991).

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．自然言語処理の研究に従事．電子情報通信学会会員．日本学術振興会特別研究員．

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．

付録: PGLRのアルゴリズムのKL1による記述

使用するアトムと変数の意味

pt :	語彙カテゴリ名
nt :	非終端カテゴリ名
n :	状態番号
r :	規則に割り付けられた番号
p :	(規則rの右辺のカテゴリ数) - 1
N :	状態番号
A :	カテゴリの引数のリスト
P :	スタックから取り出す要素数
R :	規則に割り付けられた番号
S :	スタック
B :	スタックの枝
U :	スタックの未定義の枝
I :	入力ストリーム
O :	出力ストリーム
Rcs :	真偽値を示すフラグ

Categoryプロセス

```
pt([S],A,0):- [N|_]=S|
    pt_(N,S,A,0).
otherwise.
pt([S|I],A,0):- [N|_]=S|
    pt_(N,S,A,01),
    pt(I,A,02),
    merge(01,02,0).
nt(B/S,A,0):- [N|_]=S|
    nt_(N,S,A,01),
    nt(B,A,02),
    merge(01,02,0).
nt(S,A,0):- [N|_]=S|
    nt_(N,S,A,0).
```

第一引数は入力ストリーム，第二引数はカテゴリの引数のリスト，第三引数は出力ストリームである．

Entryプロセス

- shiftエントリ：状態n，カテゴリptのエントリが'sh n1'の時

```
pt_(n,S,A,0):- true|
    0 = [[n1,A|S]].
```

スタックの先頭にカテゴリの引数のリストAと新たな状態n1を積んだスタックをストリームに出力．

- 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,[_,A0|S],A,0):- true|
    re(S,p,r,A0,01),
    mg_st(01,00),
    pt(00,A,0).
```

Categoryプロセスptを再帰的に実行するのは，レデュース操作で先読み語ptを消費しないため．

- Conflictのあるエントリ：状態n，カテゴリptのエントリが'sh n1/re r1'の時

```
pt_(n,S,A,0):- S=[_,A0|S1]|
    0 = [[n1,A|S]|02],
    re(S1,p1,r1,A0,01),
    mg_st(01,00),
    pt(00,A,02).
```

二行目はシフト操作を行い，三行目はレデュース操作である．複数のスタックが出力される．また，エントリが're r1/re r2'の時

```
pt_(n,[_,A0|S],A,0):- true|
    re(S,p1,r1,A0,01),
    re(S,p2,r2,A0,02),
    merge(01,02,03),
    mg_st(03,00),
    pt(00,A,0).
```

二行目，三行目は，同じスタックに対する異なるレデュース操作．mergeはストリームを統合するプロセス．

- acceptエントリ：状態n，カテゴリ$のエントリがaccの時

```
$_(n,[_,A|_],_,0):- true|
    0 = [A].
```

変数Aは，解析木の先頭のカテゴリの引数．

- error エントリ

```
otherwise.
pt_(_,_,_,0):- true|
  0 = [].
```

error エントリは各述語'pt' の定義の末尾に置く．

- goto エントリ：状態n，カテゴリntのエントリが'goto n1' の時

```
nt_(n,S,A,0):-  true|
    0 = [[n1,A|S]].
```

Reduceプロセス

Reduceプロセスはスタックから P 個の要素を取り出し，規則 R に対する Condition プロセスを呼び出す．

```
re([],_,_,_,0):- true|
    0=[].
re(S,0,R,A,0):- true|
    cnd(R,S,A,0).
re([_,A1|S],1,R,A,0):- true|
    cnd(R,S,(A1,A),0).
re([_,A1|S],P,R,A,0):- P=\=1|
    P1 := P-1,
    re(S,P1,R,(A1,A),0).
re(B/[_,A1|S],1,R,A,0):- true|
    cnd(R,S,(A1,A),01),
    re(B,1,R,A,02),
    merge(01,02,0).
re(B/[_,A1|S],P,R,A,0):- P=\=1|
    P1 := P-1,
    re(S,P1,R,(A1,A),01),
    re(B,P,R,A,02),
    merge(01,02,0).
```

Conditionプロセス

Conditionプロセスは各規則ごとに記述する．例えば，規則 r が次のように補強項aug を持つ時，

```
nt(A) -->
     cat1(A1),
     cat2(A2),
   { aug(A1,A2,A) }.
```

Conditionプロセスを以下のように記述する．

```
cnd(r,S,([A1],[A2]),0):- true|
   aug(A1,A2,A,Res),                    (B)
  ( Res=:=1 -> nt(S,[A],0)
  ; Res=:=0 -> 0=[] ).
```

補強項の評価 (B) が成功した時，フラグ Res には 1 が返される．

Merge_Stack プロセス

```
mg_st( [], 0 ) :- true|
        0=[].
mg_st( [S], 0 ) :- true|
        0=[S].
mg_st( [ [N,A|B] |I], 0 ):- true|
        0 =[ [N,A|U/B] |01],          (A)
        mk_br( N,A, I, U, I1 ),
        mg_st( I1, 01 ).
mk_br( _,_, [], U, 0 ):- true|
        U = [],
        0 = [].
mk_br( N,A, [[N,A|B] |I], U, 0 ) :- true| (C)
        U = U1/B,                    (D)
        mk_br( N,A, I, U1, 0 ).
mk_br( N,A, [S|I], U, 0 ) :- wait(S)|
        0 = [ S|01 ],
        mk_br( N,A, I, U, 01 ).
```

mg_st は (A) で NIS を出力する．その第一引数は入力ストリーム，第二引数は出力ストリーム．mk_br は，NIS の未定義部を保持し，(C) で先頭要素を比較し，(D) でスタックの統合を行う．

(平成3年 3 月8日受付)
(平成3年11月5日採録)