論理型言語に基づく効率的な並列一般化 LR 構文解析

An Efficient Parallel Generalized LR Parsing based on Logic Programming

沼崎 浩明 Hiroaki NUMAZAKI
田中 穂積 Hozumi TANAKA

東京工業大学 工学部
Tokyo Institute of Technology
〒152 目黒区大岡山 2-12-1 TEL. 03-726-1111 (内線 4175)
E-mail : numazaki@cs.titech.ac.jp

概要

本研究では，富田法 [Tomita 85] として知られる一般化 LR 構文解析法を並列処理に適用する際の問題点を，並列論理型言語 GHC[Ueda 85] のストリーム通信を用いて解決する方法を示す．富田法では，複数のスタックを統合する際にパージングプロセスをシフト動作で同期させるため，入力数 n に対して $O(n^2)$ の解析時間を要する [峯 89]．これに対し，本研究で提案する方法は，パージングプロセスを非同期に実行できる．これはスタックの統合を GHC のストリーム通信によって実現することにより可能となる．これによって，本方式の解析時間は O(n) となる．

1 まえがき

我々は，並列論理型言語を用いて自然言語の並列構文解析を効率良く実現する方法を検討してきた．逐次計算機上で効率の良い構文解析を実現した富田法は，LR 法の持つ予測と先読みの効果によって無駄な解析を抑制し，スタックの統合によって再計算を避けることができる．富田法は横型探索を採用するため，並列処理との整合性が良い．我々はこの点に着目し，富田法を並列論理型言語で実現した [沼崎 89a]．そこでは，グラフ構造化スタックの効率的な実現法が課題として残された．[沼崎 89b] では，スタックを木構造とし，論理型言語に適したパージングプロセスの実現方法を示すと共に，逐次計算機上でその効率性を確認した．しかし，これを実際の並列計算機で実行するには，スタックを統合する際のプロセスの同期が問題となる．富田法は各入力語のシフト動作でプロセスを同期させるため，入力長 n に対して $O(n^2)$ の解析時間を要することが指摘されている [峯 89]．本研究ではこの問題を解決するため，各パージングプロセスの同期を必要としないスタック統合の方法を考案した．具体的に述べると，先にシフト動作を終えたプロセスは直ちに次の解析に進み，後から同じシフト動作を行なったプロセスはスタックの情報を先行するプロセスに送り解析を終了する．スタックの情報は GHC のストリーム通信によって受渡しされる．

この基本的な考え方は既に [沼崎 90] で示した．しかし，そこで示した方法ではパージングプロセスの中断が生ずる可能性があった．本研究ではこれを改善し，スタックの統合のためのプロセスの同期を一切必要としない方法を示す．これによって，解析時間は入力数 n に対して O(n) となる．さらに，複数のプロセスが文の異なった部分を同時に解析し，後にそれらの持つスタックが統合された場合，通常の LR 法よりも短い時間で解析を終了するプロセスが生ずる．これは，解析が重複するプロセスが存在する場合に生じ，自然言語の解析においては頻繁に起こる．

峯らは本研究と同様，LR 法に基づいた並列構文解析アルゴリズムを示した [峯 89]．本方式と峯らの方法との違いは，本研究が全ての重複計算を避けるのに対し，峯らの方法は同時刻に同じ状態に推移したプロセスのみを統合し，同じ状態へ推移しても，その時刻が異なるプロセス同士は統合しないという戦略を採用している点にある．

第 2 節では並列一般化 LR 法について説明し，第 3 節ではスタックの操作について説明する．第 4 節ではスタックの統合を行なうプロセスを GHC で記述する方法を示し，第 5 節では本方式の効率性について議論する．第 6 節では逐次計算機上での実験によって本方式の効率性を示す．最後の第 7 節では，全体のまとめと今後の課題とを述べる．また，付録には，方法式を実現するためのパージングプロセスの記述を示す．

(1)	S	→	NP, VP.
(2)	S	→	S, PP.
(3)	NP	→	NP, PP.
(4)	NP	→	det, n.
(5)	NP	→	pron.
(6)	VP	→	v, NP.
(7)	PP	→	p, NP.

図 1: 曖昧な英語の文法

	det	n	pron	v	p	$	NP	PP	VP	S
0	sh1		sh2				4			3
1		sh5								
2				re5	re5	re5				
3					sh6	acc		7		
4				sh8	sh6			10	9	
5				re4	re4	re4				
6	sh1		sh2				11			
7					re2	re2				
8	sh1		sh2				12			
9					re1	re1				
10				re3	re3	re3				
11				re7	sh6/re7	re7		10		
12					sh6/re6	re6		10		

図 2: LR パーズ表

2 並列一般化 LR 構文解析法の効率化

LR構文解析法は，文法から得られるLRパーズ表に基づいたスタックの操作によって文を解析するアルゴリズムである．図1に曖昧な英語の文法規則を示し，図2にその規則から計算されたLRパーズ表を示す．スタックの操作にはカテゴリと次の状態とをスタックに積むシフト動作と，必要な数の要素をスタックから取り出し，これに規則を適用して新たなカテゴリを生成するレデュース動作とがある．

解析は状態0から始まり，先読み語のカテゴリと状態番号によってテーブルを参照する．'sh N'のエントリはパーザがシフト動作を行い，状態Nに推移することを指定し，'re N'のエントリはスタックに積まれた要素に対し，第N番目の規則を適用して規則左辺のカテゴリを新たに生成することを示す．レデュースの後，パーザはスタックの先頭の状態番号と生成されたカテゴリによって，テーブルの右側の部分を参照する．参照されたエントリの番号は，パーザがそのカテゴリをシフトした後，次に推移すべき状態を示している．入力文が受理される場合，パーザは最後に'acc'のエントリに到達する．一方，入力文が受理されない場合，パーザは解析の途中で空白のエントリに到達して解析を終了する．

文法に曖昧性がある時，LRパーズ表のエントリには複数の動作の指定，すなわち，コンフリクトが生じる．パーザはコンフリクトに対して次の動作を一意に決定できない．図2にはコンフリクトのエントリが二つが存在する．このエントリはパーザの状態が11または12で，先読み語のカテゴリがpのときに参照される．

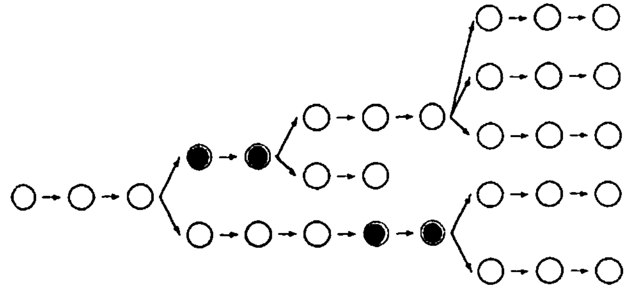


図 3: パージングプロセスの推移

一般化LR構文解析法はコンフリクト中の全ての動作を実行し，可能な構文的解釈を全て求めるアルゴリズムである．一般に，この解析過程は図3に示すような木構造となる．並列計算機を使用すれば，これら複数の解析を同時に進めることができるため，全ての解析が一つの木を計算する時間で終了する．しかし，図3に示すように，プロセスの数は，入力長に比例する解析ステップ数に対して指数関数的に増大する．従って，実際の並列計算機上に単純な一般化LRパーザを実現すると，解析に要するプロセスの数が物理的なプロセッサの数をはるかに上回る可能性がある．

この問題を解決するための方法として，次の二つが考えられる．

- 曖昧な文の構文解析では，複数のパージングプロセスが同じシフト動作を実行した後，解析過程の重複が生ずる．この部分の解析を一つのプロセスに統合すれば，プロセス数を削減できる．
- 規則の適用に制約を設け，その制約に反する規則を適用しようとした解析を刈り込めば，プロセス数を削減できる．

富田法はこの二つを実現したアルゴリズムであり，逐次計算機上でその効率性が確認されている．しかし，これを並列計算機上に実現するには重大な問題がある．一般に，重複する複数の解析過程には時間的なずれがある．例えば，図3中の黒い円で示した解析ステップのように，重複した解析は時間的にずれて生ずる．富田法ではこの時間的なずれを調整するために，各入力語のシフト動作でプロセスの同期を取る．つまり，先にシフト動作を終えたプロセスは，他の全てのプロセスがシフト動作を終えるまで実行を中断する．毎回のプロセスの中断時間は最悪の場合，入力数に比例した回数のレデュース動作を行なう時間となる．従って，富田法の並列実行に要する時間は，入力長nに対して$O(n^2)$となる[峯 89]．これは，プロセスの統合を行なわない場合に要する計算時間が，$O(n)$(一つの解析木を作る時間)で済むことを考慮すると，著しい効率の低下と言える．

本研究ではこの問題をGHCのストリーム通信を利用して解決した．例えば，解析にシフト・レデュースコンフ

リクトが生じた時，シフト動作を先に終えたプロセスは直ちに次の解析に進む．別のプロセスが何回かのレデュース動作の後に，同じシフト動作を行なった場合，そのプロセスの持つスタックを，先行するプロセスの持つスタックに統合する．これを簡単に説明すると以下のようになる．

コンフリクトによって複数のプロセスが生じた後，先にシフト動作を終えたプロセスの持つスタックに，後からスタックを結合する場所を確保するための変数を埋め込む（第3.2節参照）．先行するプロセスはこのスタックを用いて次の解析に進むことができる．一方，他のプロセスがある時点で同じシフト動作を実行した時，そのプロセスの持つスタックを，先行するプロセスの持つスタック中の変数に束縛する．本研究では，これを実現する merge_stack プロセスを GHC で記述した（第4節参照）．このプロセスを [沼崎 89b] で提案したパーザに組み込むと，本方式を実行するパーザが得られる．パーザの記述は付録で紹介する．

これによって，本方式は各パージングプロセスが非同期に文を解析できる．通信コストやプロセッサ数など，ハードウェアの制約がない理想的な場合，解析に要する時間は一つの解析木を生成する時間に等しく，O(n) となる [峯 89]．

3 スタックの操作

3.1 木構造化スタックの実現

富田法では，複数のスタックの共通要素を統合したグラフ構造化スタックを用いることにより，解析の重複を避けることができる．しかし，これを論理型言語で実現するには，副作用を用いることが必要となり，解析の効率を低下させる [沼崎 89a]．そこで我々はスタックの先頭の要素のみを統合した木構造化スタックを採用する．解析の重複を避けるためには，木構造化スタックで十分である．木構造化スタックは，論理型言語のリスト型データとして容易に表現できる．

例えば，図1の文法と図2のパーズ表を用いて次の入力文を解析することを考える．

文　　：I saw a man in the park .
カテゴリ：pron v det n p det n $

解析がカテゴリ p まで進むと，シフト・レデュースコンフリクトが起こり，パージングプロセスは二つに分かれる．それぞれのプロセスがカテゴリ p をシフトした後のスタックは以下のようになる．

(1) 先頭：[6,p,3,s,0] :底
(2) 先頭：[6,p,12,np,8,v,4,np,0] :底

これ以後の解析は，スタックから先頭の要素'6,p'をポップするまで同一となる．そこで，スタックの先頭の部分を次のように統合して木構造化スタックを作ると，それ以後は一つのプロセスで解析を進めることができ，重複計算を避けることができる．

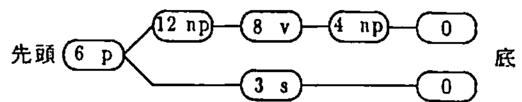


図 4: 木構造化スタック

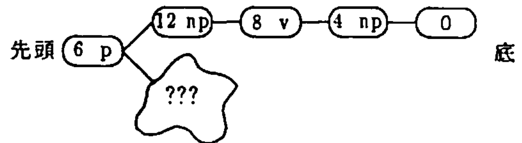


図 5: 未定義部分を持つ木構造化スタック

(3) [6,p, [12,np,8,v,4,np,0], [3,s,0]]

この木構造化スタックは図4のような構造を持つ．

3.2 ストリーム通信を利用したスタック操作

既に述べたように，このようなスタックを作るために複数のプロセスを同期させると並列性が低下する．ここでは，本研究で提案するストリーム通信を利用したスタックの操作について述べる．今，3.1節で示したスタック (2) がスタック (1) よりも先に生成された場合を考える．この時，パージングプロセスはこのスタックを，スタック統合プロセス（以後 merge_stack プロセスと呼ぶ）に渡し，次のようなスタックを得る．

(4) [6,p, [12,np,8,v,4,np,0] | Branch]

このスタックはスタック (2) の 12,np 以下の部分を木構造化スタックの一つの枝に置き換え，他の枝が後で結合できる場所を変数 Branch として確保している．このスタックは図5に示すようなイメージとなる．実際の自然言語の構文解析では，パージングプロセスはこのような未定義の部分を数多く含むスタックを用いて解析を進めていく．

これ以後の解析は以下のように進む．

- このスタックを受けとったプロセスは次の解析を直ちに継続する．
- もう一方のプロセスがスタック (1) を生成し，merge_stack プロセスにそのスタックを渡すと，merge_stack プロセスは変数 Branch を次のように具体化する．

Branch=[[3,s,0] | Branch1]

変数 Branch1 は，他のプロセスがスタックを merge_stack プロセスに送るたびに，再帰的に具体化される．

- 他の全てのプロセスが持つスタックが統合できない場合，merge_stackプロセスは，

  Branch=[ ]

  という具体化を行なう．

- 変数Branchが具体化される前に，レデュース動作がスタック(4)の'6,p'の部分を越えてスタックを還元する場合，そのレデュースを二つのプロセスが受け持つ．一方のプロセスはスタック(2)から得られた枝をレデュースし，もう一方のプロセスは変数Branchが具体化されるまで実行を中断する．ただし，これは文を左から右に解析するパージングプロセスの中断ではないので，全体の解析時間には影響しない．変数Branchが具体化されると，中断していたプロセスは直ちにその枝のレデュースを実行する．

- スタック(1)を用いて解析を行なうプロセスは，'6,p'のシフト以後の解析の重複部分を実行しなくてよいため，(なぜなら，重複部分の解析はもう一方のプロセスによって実行されるから．)そのプロセスは一つの木を作る時間よりも短い時間で終了する(第5節参照)．

以上が本研究で提案する解析法である．

4 スタック統合プロセスの実現

この節ではmerge_stackプロセスのアルゴリズムを，並列論理型言語GHCのサブセットであるKL1で記述した例を示す．このプロセスは各パージングプロセスから呼び出される．パージングプロセスの記述は付録に示した．

```
merge_stack([],Out):- true | Out=[].
merge_stack([[ N,C | Tail ] | In ],Out):-  %(1)
    true|
    Out=[[ N,C,Tail | Branch ] | Out1 ],   %(2)
    make_branch(N,C,In,Branch,Rest),       %(3)
    merge_stack(Rest,Out1).

make_branch(_,_,[],Branch,Out):- true |
    Branch=[],Out=[].
make_branch(N,C,[[ N,C|Tail ]|In ],Branch,
    Rest):-
    true|
    Branch=[ Tail | Branch1 ],             %(4)
    make_branch(N,C,In,Branch1,Rest).
otherwise.
make_branch(N,C,[ Stack | In ],Branch,Rest):-
    true|
    Rest=[ Stack | Rest1 ],
    make_branch(N,C,In,Branch,Rest1).
```

merge_stackの第一引数は複数のスタックを受けとるストリームであり，第二引数は統合したスタックを出力するストリームである．各ストリームは次のような形式とする．

```
[ STACK1, STACK2, ..., STACKn | Stream ]
```

このプロセスはパージングプロセスからスタックを一つ受け取るたびにアクションを起こし，木構造化したスタックを出力する．その際，木構造化スタックの枝の部分は遅れて具体化される．このプロセスの実行は，入力ストリームが閉じると終了する．

プログラム中の(1)の行では，ストリーム中の一つのスタックから先頭の要素N,Cを取り出す．(2)では，そのスタックを木構造化して直ちに出力ストリームに送る．この時，スタックの先頭以外の部分Tailは木構造化スタックの一つの枝になる．また，他の枝がつながる場所を変数Branchとしてスタックに埋め込む．(3)のmake_branchは，もう一方のストリームに同じ先頭要素N,Cを持つスタックが流れてくるたびに，Branchを具体化していく((4)の行)．

5 並列性の比較

この節では，本方式による並列構文解析の過程が，他の方式とどのように異なるのかを具体例で示し，時間的効率性の違いを明らかにする．そこで，図1の文法と，図2のパーズ表を用いて入力文：

  I saw a man in the park.

を解析することを考える．その解析過程を，プロセスを統合しないLR法，富田法，そして本方式についてそれぞれ図7の(a),(b),(c)に示す．この図はスタックにカテゴリを積んだ直後のプロセスの状態を示したものである．図の各円の中の番号は各プロセスの状態を示し，矢印の上のシンボルは状態推移の際にスタックに積まれたカテゴリを示している．この解析過程を簡単に説明する．

- プロセスは状態0から始まり，先読み語pronによってパーズ表を参照すると，'sh 2'のエントリを得る．これに従って，pronをスタックにシフトし，プロセスは状態2へ推移する．この時スタックは[2,pron,0]となる．

- その状態2と次の先読み語vでパーズ表を参照すると，'re 5'が得られる．そこで，5番目の規則を用いてpronをnpに還元する．このとき，スタックの先頭から状態2とカテゴリpronが取り除かれ，スタックは[0]に戻る．スタックトップの状態0と得られたカテゴリnpから，エントリ'4'が参照される．そこで，カテゴリnpと4とをスタックに積み，プロセスは状態4に推移する．これによって，スタックは[4,np,0]となる．

- 上のようなシフト・レデュース動作を繰り返し，プロセスが状態 12，先読み語 p に到達すると，'sh 6/re 6' というコンフリクトに出会う．各アルゴリズムはここでプロセスを二つに分岐し，それぞれのエントリに対する解析を行なう．

- 図 7 (a) に示した単純な LR 法では，それ以後の解析は完全に独立に進む．そこでは，p をシフトして状態 6 へ推移した後の 3 ステップ (矢印 3 つ分) が全く同一であることが分かる．このような解析の重複を許すと，入力文によっては，プロセス数が爆発してしまう．

- 図 7 (b) に示した富田法では，解析の重複を避けるために，先読み語 p を先にシフトしたプロセスは，もう一方のプロセスが同じ語のシフト動作を終えるまで解析を中断する．このため，全てのプロセスが終了する時刻は，(a)の場合よりも遅い．最悪のケースとして，全てのシフト動作でプロセスが中断が生ずる場合は $O(n^2)$ の解析時間が必要となる [峯 89]．これは，(a) と比べて著しい効率の低下と言える．

- 図 7 (c) に示したように，本方式では解析の重複を避けるためにプロセスを統合する．しかし，そのためにプロセスの実行を中断する必要はない．図 7(c) の上側のプロセスは，カテゴリ p をシフトした後，第 3.2節で示したスタックを用いて，表 7に示すように解析を進める．

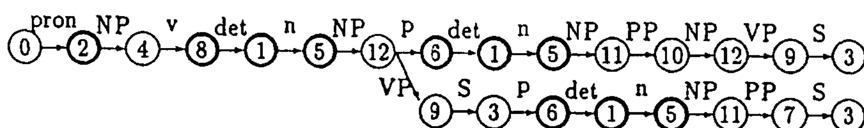
- 先行するプロセスは，スタックに Branch という変数を持ったまま解析を進めていく．図 7(c) から分かるように，その解析がカテゴリ n をシフトした時，もう一方の解析が，カテゴリ p をシフトし，スタック [6,p,3,s,0] を生成する．merge_stack プロセスはこのスタックを受けとり，先のシフトと同じシフトが実行されたことを確認すると，このスタックの先頭要素'6,p' を除いた部分を変数 Branch に束縛する (第 4 節のプログラム中の (4) の行)．

- その値はストリーム通信によって先行するプロセスに送られる．これによって，表 7の最後に示した解析ステップでは，既に変数が具体化されている．

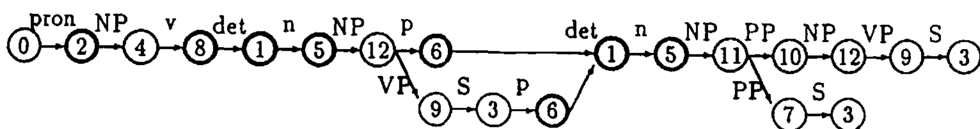
- こうして，本方式ではプロセスを統合しないアルゴリズムと同様，O(n) の時間で解析を終了する．さらに，図 7(a),(b),(c) を比較して分かるように，他のアルゴリズムよりも短い時間で終了するプロセスが存在する (第 3.2節の最後に述べた通り)．尚，この例に対しては，峯らのアルゴリズムは (a) と同様に解析が進む．これは，重複部分が時間的にずれて生じているためである．峯らのアルゴリズムではこのようなプロセスは統合しない．

これまでの議論では，プロセス間の通信のコストと，merge_stack プロセスのオーバーヘッドを無視してきた．通信コストは計算機によって異なり，メモリを共有しない疎結合型の計算機では，それが無視できない大きさになると予想される．一方，merge_stack プロセスはパージングプロセスと並列実行できるため，オーバーヘッドは比較的小さいと思われる．実際，逐次計算機上での実験では，merge_stack プロセスの導入により，全解析ステップ数 (reduction 数) は減少することが確認されてい

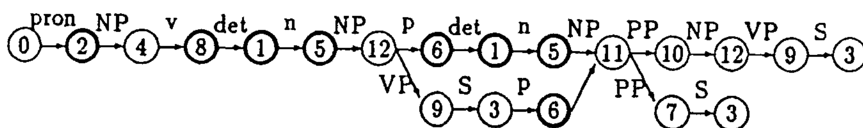
Shift Action　　Reduce & Goto Action



(a): プロセスを統合しない単純な並列 LR 構文解析



(b): 富田法による並列構文解析



(c): 本方式

図 6: 構文解析過程

る．その理由は，このプロセスが，スタックの先頭の要素しか参照しないため，スタックの深さに依存しない処理時間で実行できることにあると思われる．

6 実験

本方式の効率性を示すために，Prolog 上の Flat-GHC(ガード部に組み込み述語のみ記述できる GHC のサブセット言語）処理系を用いて英語の文を解析する実験を行なった．この処理系は，GHC のゴールの実行に要した reduction 数，suspention 数，cycle 数その他の情報を表示する．cycle 数は理想的な並列計算機上での解析時間に相当する．実験に用いた文法は規則数 123 の CFG であり，以下の英文を用いて解析に要する cycle 数を測定した．その結果を表 2に示す．

1. I open the window.
2. Diagram is an augmented grammar.
3. The structural relations are holding among constituents.
4. It is not tied to a particular domain of applications.
5. Diagram analyzes all of the basic kinds of phrases and sentences.
6. This paper presents an explanatory overview of a large and complex grammar that is used in a sentence.
7. The annotations provide important information for other parts of the system that interpret the expression in the context of a dialogue.
8. For every expression it analyzes, diagram provides an annotated description of the structural relations holding among its constituents.
9. Procedures can also assign scores to an analysis, rating some applications of a rule as probable or as unlikely.

この実験では，本方式に対し富田法に基づく方式は，最大で 4.8 倍，平均で 3.0 倍の cycle 数を必要としている．

文番号	単語数	木の数	本方式	富田法
1	4	2	93(cycle)	185(cycle)
2	5	1	125	115
3	7	15	175	383
4	10	14	219	506
5	11	30	261	732
6	18	56	416	1137
7	21	192	663	3191
8	18	200	1296	3898
9	20	186	444	1025

表 2: 構文解析に要する cycle 数の比較

これは予想以上に大きな差である．一回の同期でプロセスが待つ時間は短くても，富田法は全ての入力語に対して同期を行なう必要があるため，プロセスの中断時間が累積してこのような差をもたらすと考えられる．尚，2番の結果が逆転しているのは，文に曖昧性がないため，本方式のスタック操作のオーバーヘッドが顕著に現れたものと思われる．

7 おわりに

本研究では，一般化 LR 構文解析法を並列処理に適用する際の問題点を，並列論理型言語 GHC のストリーム通信を用いて解決する方法を示した．この方法は富田法の効率性を生かし，その問題点を解決した構文解析法である．解析に要する時間は O(n) であり，複数の解析の重複を統合することによって，通常の LR 法よりも短い時間で終了するプロセスが生ずる．

一方，本方式で生成されるプロセスの数は，入力長に対して指数オーダとなる．これは，本方式が意味解析やその他の制約によってプロセス数を削減することを目指しているために，解析木を個別に扱う必要があるためである．構文解析と意味解析を融合するには，解析中に得られる各部分木に対して異なる意味解釈を与える必要があり，富田法で提案するような Packed Node や，峯らのようなパージングプロセスと解析木を分離する方式を採用することは困難である．この点について富田は規則の制約条件を判定する際に，Packed Node 中の属性情報を参照する必要が生じた場合は，結合した Node を Unpack しなければならないと述べている [Tomita 87]．しかし，Pack,

State	Symbol	Action	Stack
6	det	sh 1	[1,det,6,p, [12,np,8,v,4,np,0] | Branch]
1	n	sh 5	[5,n,1,det,6,p,[12,np,8,v,4,np,0] | Branch]
5	$	re 4	[6,p,[12,np,8,v,4,np,0] | Branch]
6	np	goto 11	[11,np,6,p,[12,np,8,v,4,np,0],[3,s,0]]

表 1: 先行するプロセスの解析過程

Unpackの操作が解析効率にどの程度影響するのかは明らかにされていない.

本方式をMulti-PSIのような疎結合型の並列計算機上に実現すると，プロセス間の通信コストが無視できなくなる．今後の課題として，本方式を実際の並列計算機上で実現し，その有用性を実証する必要がある．

謝辞

本研究を進めるにあたり，日頃から暖かいご支援をいただいた田中研究室のみなさんに感謝致します．また，本研究に対し貴重な御意見をいただいたICOTのKL1-タスクグループの方々に感謝致します．

参考文献

[沼崎 89a] 沼崎浩明，田村直良，田中穂積:並列論理型言語を用いた自然言語処理のためのLR構文解析アルゴリズムの実現，PROCEEDINGS OF THE LOGIC PROGRAMMING CONFERENCE '89, 11.2, PP.183-192 (1989)

[沼崎 89b] 沼崎浩明，田村直良，田中穂積:並列論理型言語による一般化LR構文解析アルゴリズムの実現，自然言語処理 74-5, PP.33-40 (1989)

[沼崎 90] 沼崎浩明，田中穂積:LR法に基づく新しい並列構文解析アルゴリズム，情報処理学会第40回全国大会, 4F-3, pp.456-457 (1990)

[淵 87] 淵一博監修，古川康一，溝口文雄共編:並列型論理言語GHCとその応用，共立出版 (1987)

[峯 89] 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研究報告，73-1, pp.1-8 (1989)

[Aho 72] Aho,A.V.and Ulman,J.D.: *The Theory of Parsing,Translation,and Compiling,* Prentice-Hall,Englewood Cliffs,New Jersey (1972)

[Matsumoto 88] Matsumoto,Y.:*Natural Language Parsing System based on Logic Programming,* Doctoral Thesis, Kyoto University (1988)

[Tomita 85] Tomita,M.:*Efficient Parsing for Natural Language,* Kluwer Academic Publishers (1985)

[Tomita 87] Tomita,M.: *An Efficien Augmented-Context-Free Parsing Algorithm,* Computational Linguistics, Vol.13, Numbers 1-2, pp.31-46 (1987)

[Ueda 85] Ueda,K:*Guarded Horn Clauses,* Proc. The Logic Programming Conference, Lecture Notes in Computer Science, 221 (1985)

付録

以下では，本方式を実現する並列LRパーザのKL1による記述を示す．パーザの実現の特徴はLRパーズ表の各エントリをパージングプロセスの記述に置き換え，そこでスタックを操作する点にある．

- 親プロセスの記述
  全ての語彙カテゴリについて下に示すような親プロセスを記述する．このプロセスはストリームにスタックを受けとり，各スタックについてパーズ表のエントリを実行するプロセスを呼び出す．例えば，カテゴリpの親プロセスは，

```
p( [], _, Out ):- true |
   Out = [].
p( [[N|Stack]|Stream], Cat, Out ):-
   true |
   p_( N, [N|Stack], Cat, Out1 ),
   p( Stream, Cat, Out2),
   merge( { Out1, Out2 }, Out ).
```

  と記述する．ここで，p_はエントリを実行するプロセスであり，各エントリに対して以下のように記述される．

- シフトエントリの記述例
  状態1，先読み語がnのエントリ'sh 5'は，

```
n_( 1, Stack, Cat, Out ):- true |
   Out = [ [ 5,Cat | Stack ] ].
```

  と記述する．

- レデュースエントリ
  状態2，先読み語が'v'のエントリ're 5'は，

```
v_( 2, [2,T1|Stack], Cat, Out ):- true |
   cond( 5, Stack, [T], Result ),
   v(Result, Cat, Out ).
```

  と記述する．condプロセスは規則の適用条件に応じて解析の継続あるいは停止を決めるプロセスである．また，プロセスvは親プロセスの呼び出しである．その親プロセスは，Result中の全てのスタックに対して，プロセスv_を呼び出す．

- シフト・レデュースコンフリクトのあるエントリ
  状態11，先読語がpのエントリ'sh 6/re 7'は，

```
p_( 11, [11,T|Stack], Cat, Out ):- true |
    Out=[ [ 6,Cat,11,T | Stack ] | Out1 ],
    reduce( 7, 1, Stack, [T], Result ),
    p( Result, Cat, Out1 ).
```

と記述する．reduceプロセスは第二引数に与えられた数だけスタックの要素をポップし，それをcondプロセスに渡す．

- レデュース・レデューコンフリクトのあるエントリ
  例えば，先読みのカテゴリcat，状態2，のエントリが're 5/re 6/re 7'であったと仮定したとき，これを，

```
cat_( 2, [2,T|Stack], Cat, Out ):- true |
    cond( 5, Stack, [T], Out1),
    reduce( 6, 1, Stack, [T], Out2),
    reduce( 7, 1, Stack, [T], Out3),
    merge( {Out1,Out2,Out3}, Result),
    cat( Result, Cat, Out ).
```

と記述する．

- acceptエントリ
  状態3，先読みが$(文の終了)のエントリ'acc'は，

```
$_( 3, [3,Info,0], _, Out ):- true |
    Out = Info.
```

と記述する．

- 空白(error)エントリ
  解析の失敗を示す空白のエントリは，例えば，カテゴリpronについては，

```
otherwise.
pron_(_, _, _, Out):- true |
    Out = [].
```

と記述し，これをpron_の記述の末尾に置く．

- パーズ表右側のgoto部のエントリは，シフトエントリと同じである．
- reduceプロセス

```
reduce(_,_,[],_,NewSt):- true|
    Out=[].
reduce(N,0,Stack,Cat,Out):- true|
    cond(N,Stack,Cat,Out).
reduce(N,Pop,[S,Cat1|Stack],Cat,Out):-
    integer(S)|
    Pop1 := Pop-1,
    reduce(N,Pop1,Stack,[Cat1|Cat],Out).
reduce(N,Pop,[[_,Cat1|Stack]|Rest],Cat,
    Out):- true|
    Pop1 := Pop-1,
    reduce(N,Pop1,Stack,[Cat1|Cat],Out1),
    reduce(N,Pop,Rest,Cat,Out2),
    merge({Out1,Out2},Out).
```

引数は左から，規則番号，ポップする要素数，入力スタック，ポップしたカテゴリ情報のリスト，出力スタックのストリームである．

- condプロセス
  condプロセスはDCG文法の補強項を評価する．例えば，ある文法のi番目の規則が次のように記述されているとき，

```
(i) sent(Subj,[sent,Np_T,Vp_T]) -->
      np(Np,Np_T),
      vp(Vp,Vp_T),
      { subj_verb_check(Np,Vp) }.
```

この規則に対応するcondプロセスは以下のように記述できる．

```
cond( i,[Top|Stack],[[Np,Np_T],[Vp,Vp_T]],
    Out ):- true |
    subj_verb_check(Np,Vp,Result),
    ( Result=true ->
        ( integer(Top) ->
            sent_( [Top|Stack],
            [Subj,[sent,Np_T,Vp_T]],Out )
        ; otherwise
        ; true ->
            sent( [Top|Stack],
            [Subj,[sent,Np_T,Vp_T]],Out1 ),
            merge_stack(Out1,Out) )
    ; otherwise
    ; true ->
        Out = [] ).
```

- パーザの起動
  パーザは入力文の各語を形態素解析して得たカテゴリとそれに付随する引数のリストを受け取り解析を開始する．今，入力文の各単語のカテゴリがpron，v，nで，付随する情報がPron，V，Nの時，次のような親プロセスの列を呼ぶことによってパーザを起動する．

```
?- pron( [ [0] ], Pron, St1a ),
     merge_stack( St1a,St1 ),
   v( St1, V, St2a ),
     merge_stack( St2a,St2 ),
   n( St2, N, St3a ),
     merge_stack( St3a,St3 ),
   $( St3, [], Info ).
```

最初のゴールpronに一つのスタック[0]を与える．第二引数に与える情報は，[Arg1,Arg2,...]のようなカテゴリに付随した引数のリストとする．
多品詞語の扱いは容易で，例えば二つ目の単語の品詞が'v'と'n'ならば，三行目の親プロセスの呼び出しを

```
v( St1, V, St2b ), n( St1, N', St2c ),
merge( {St2b,St2c},St2a )
```

とすればよい．