自然言語処理 74－5
（1989. 9. 21）

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

沼崎 浩明　　田村 直良　　田中 穂積

東京工業大学工学部　　横浜国立大学工学部　　東京工業大学工学部

概要

本論文では，並列論理型言語 GHC[Ueda 85]に基づいて一般化LR構文解析アルゴリズムを実現するパーザ GLP(Generalized LR Parser)について述べる．LR構文解析法[Knuth 65]のアルゴリズムを，コンフリクトを扱えるように一般化し，自然言語に適用した研究として富田法[Tomita 86]が良く知られている．富田法は横型的な探索によってコンフリクトを扱っているため，並列処理との整合性がよい．本研究ではこの点に着目し，一般化LR構文解析アルゴリズムを並列論理型言語 GHC[Ueda 85]の枠組で記述する．筆者らは既に，富田法で導入されたグラフ構造化スタックを用いる並列LRパーザを示した[沼崎 89]．そこでは，副作用を用いてスタックを実現したため，効率のよいパーザが得られなかった．そこで本研究では，リストで表現された木構造化スタックを用いる効率のよい並列LRパーザ GLP を実現した．論理型言語のすぐれた記述力により，GLP のアルゴリズムの記述は容易である．GLP は Prolog 上にも容易に移植できる．GLP を Quintus Prolog 上で走らせ，規則数 550 程度の DCG 形式の実用的な英語の文法規則を用いて，英語の文の解析時間を，松本の SAX[松本 86]と比較した．実験の結果，GLP は平均して SAX の 2.5 倍速いことが分かった．

An Implementation of Generalized LR-Parsing Algorithm based on Parallel Logic Programming Language

Hiroaki NUMAZAKI†　　Naoyoshi TAMURA††　　Hozumi TANAKA†

† Department of Computer Science, Tokyo Institute of Technology
(2-12-1 Oookayama Meguro-Ku Tokyo 152 Japan)

†† Department of Electronic Information Engineering, Yokohama National University

Abstract

The generalized LR parsing algorithm, developed by Tomita[Tomita 86], which utilized the breadth first strategy to handle conflicts occured in a LR parsing table, can treat context free grammar. Considering that breadth first strategy has a good compatibility with parallel processing, and the simplicity for implementing the Tomita's algorithm in GHC and Prolog. We propose a new generalized LR parser(GLP) based on parallel logic programming language GHC[Ueda 85]. In this paper,first, we describe the implementation of GLP. Then to verify the ability of GLP, we compared the parsing time of GLP with that of SAX[Matsumoto 88] using about 550 English grammar rules on Quintus Prolog. The experiment revealed that our parser runs 2.5 times faster than SAX on an average.

1 まえがき

本論文では，並列論理型言語 GHC[Ueda 85]に基づいて一般化 LR 構文解析アルゴリズムを実現するパーザ GLP(Generalized LR Parser)について述べる．LR 構文解析法[Knuth 65]を文脈自由文法に適用すると，パーザを駆動するテーブルにコンフリクトを生じ，パーザの動作を一通りに決定できない場合が生ずる．よって文脈自由文法を扱うためには，そのアルゴリズムを一般化する必要がある．これを行なった研究として富田法[Tomita 86]が良く知られている．筆者らは既に，富田法で提案されているグラフ構造化スタックとスタックリストを用いた一般化 LR パーザを GHC の枠組で記述し，その効率性を検討した[沼崎 89]．グラフ構造化スタックを用いたパーザは再計算を行なわないので本来効率的であるが，我々のインプリメントではスタックの実現に副作用を用いていたため効率が悪かった．そこで本研究では，スタックのトップだけを統合した木構造化スタックを用いることにした．木構造化スタックはリストで表現できるため副作用を用いずに済む．これを用いて再計算を行なわない効率的な一般化 LR パーザ GLP を実現した．GLP の基本的アルゴリズムは富田法に従っており，コンフリクトによってスタックが複数生じた場合には，それを並列に処理するのが特徴である．

一般化 LR 法を用いて並列構文解析を行なう研究として，安留[安留 88]及び峯[峯 89]の報告がある．特に富田法のアルゴリズムに基づいて並列構文解析を行なうパーザを示した安留の研究は本研究に先行するものである．しかし，安留の研究は並列処理を行なうハードウェアの構成から出発しているため，文の曖昧さがプロセッサ数を越えた場合などの，プロセスの管理などに残された問題は多いと思われる．これに対し，本研究は並列論理型言語に基づいてパーザを構成するため，実現が極めて容易である．また，パーザの記述に若干変更を加えることにより prolog 上にインプリメントできることも特徴である．

次節では一般化 LR 構文解析法について述べ，第 3 節で木構造化スタックの実現方法を述べる．第 4 節では GLP のアルゴリズムを GHC による記述とともに説明する．我々は文法規則に DCG[Pereira 80]を採用しているが，アルゴリズムの説明には，理解を容易にするために文脈自由文法を用いる．第 5 節で DCG を扱う方法とその問題点について述べる．第 6 節では GLP を prolog 上に移植する方法について述べる．第 7 節では規則数 550 程度の DCG 形式の英語の文法規則を用いて文の解析時間を計った結果を示す．我々の使用している GHC の処理系 PDSS の制約から，この規則を載せることができなかったので，実験は Quintus Prolog 上で行ない，比較対象として松本の SAX[松本 86] を用いた．実験の結果，GLP は平均して SAX の 2.5 倍速いことが分かった．最後に今後本方式を自然言語処理に適用する局面での問題点について述べる．

2 LR 構文解析法について

LR 構文解析法の特徴は，与えられた文法規則からパーザの動作を決定するテーブルをあらかじめ計算する点にある．文の解析はこのテーブルに従って進められる．文法規則が LR 文法のとき，パーザの動作は決定的となる．しかし一般に，曖昧さのある文脈自由文法に対してはテーブルにコンフリクトを生じ，パーザの動作を一通りに決定できない．

図 1に曖昧さを持つ英語の文法規則を示し，これから得られるテーブルを図 2 に示す．テーブルの左側の部分を action 部と呼び，ある状態（テーブルの各行）に対して先読みの単語の品詞が決ると一つのエントリが定まる．'sh N'のエントリはパーザが先読みの単語をスタックの先頭にプッシュして状態 N に行くことを示し，'re N'のエントリは N 番目の規則を用いてスタック上の要素を還元することを示している．'acc'のエントリは文の受理を示し，空白はエラーを示している．テーブルの右側の部分は goto 部と呼ばれ，還元動作の後パーザがどの状態へ行けば良いかを示している．図 2 にはコンフリクトが 4 箇所生じている．状態 14 と 16 の行の p と relp の列である．このようなコンフリクトを shift-reduce コンフリクトと呼ぶ．パーザはこの点で次の動作を決定できない．

(1)	S	→	NP, VP.
(2)	S	→	S, PP.
(3)	NP	→	NP, RELC.
(4)	NP	→	NP, PP.
(5)	NP	→	det, noun.
(6)	NP	→	noun.
(7)	NP	→	pron.
(8)	VP	→	v, NP.
(9)	RELC	→	relp, VP.
(10)	PP	→	p, NP.

図 1: 曖昧性のある文法規則

富田法ではコンフリクトの生じた点で，スタックを分岐させてそれぞれの処理を進めていく横型的な方法を採用している，その際，還元動作を優先させシフト動作で複数の処理の同期をとる．シフト動作の後，分岐した複数のスタックのトップが同じになった場合，これを統合してグラフ構造化スタックを作る．この横型的な方法は並列処理にうまく適合する．本研究ではこの方式を用いて並列 LR 構文解析アルゴリズムを実現した，本研究の場合，スタックは見かけ上は木構造化されるが，実質的にはグラフ構造化されていることになる．このことについては第 4 章で述べる．

3 木構造化スタックの実現

ここでは，具体的な例を用いてスタックの木構造化について説明する．文の解析中にコンフリクトが生じると，スタックは複数に分かれる．今，図 2 のテーブルに従って，"I open the door with a key ."という文を"with"のところまで解析が進んでいるとしよう．この時のスタックは，

(top) [16,NP,10,V,5,NP,0] (bottom)

となっている(ここで NP や V には実際には木の情報が入っている)．"with"の品詞が"p"で，スタックトップの状態が 16 なので，テーブルを参照すると，'sh 7/re 8'というコンフリクトにぶつかる．そこで，スタックを二つ作りそれぞれの処理を別々に進めるが，シフト動作の方はもう一方のプロセスもシフト動作に到達するまで待ち合わせる．二つのプロセスがシフト動作を終えた時点のスタックは，

$[7, P, 4, S, 0]$

$[7, P, 16, NP, 10, V, 5, NP, 0]$

となっている．ここで，スタックトップが共通なので，それを一つに統合して木構造化を行なうと，

$$[7, P, [[4, S, 0], [16, NP, 10, V, 5, NP, 0]]]$$

となる．これ以後の"a key ."の処理は一つのプロセスで進めれば良く，再計算を避けることができる．"."まで解析が進むと還元動作が生じ，スタックの分岐点を越えて還元しようとする．このとき，再びスタックは二つに分かれる．

4 並列一般化 LR構文解析アルゴリズムの実現

本節で示す GLP のインプリメンテーションは，LR テーブルの各エントリを直接パージングプロセスの記述に置き換える点が特徴である．パージングプロセスは次の 4 つのプロセスで構成される．

- action プロセス
  action テーブルの動作を実現する．
  shift 動作に対しては，入力スタック上に木の情報と次の状態を積み，それを出力スタックとしてプロセスを終える．
  reduce 動作に対しては，reduce プロセスを呼びだし，続いて次の action プロセスを呼び出す．その際，還元されたスタックを渡す．
  shift-reduce コンフリクトがある場合には．reduce プロセス，action プロセスを呼び出した後，merge_stack プロセスを呼びだし，木構造化を行なったスタックを出力スタックとしてプロセスを終了する．
  accept に対しては，スタックから木の情報を取り出し，プロセスを終了する．
  error に対しては，出力スタックに[]を返しプロセスを終了する．

- reduce プロセス
  文法規則に従ってスタック上の要素を還元し，構文木を上へ伸ばして，goto プロセスに渡す．スタックをポップする際に木構造の分岐点に到達した場合，スタックを複数に分離し，それぞれのスタックについて還元動作を並列に実行する．

- goto プロセス
  goto テーブルのエントリに従って，次の状態と reduce プロセスで作られた木の情報を入力スタックに積み，それを出力スタックとしてプロセスを終了する．

- merge_stack プロセス
  第 3 節で示したように複数の入力スタックのトップが等しいとき，これを統合して一つの木構造化スタックを作る．このプロセスの実現方法はパーザ全体の並列度に影響する．前節で示したが，スタックが[16,NP,10,V,5,NP,0]のようなとき，'sh 7/re 8'のコンフリクトを処理すると，shift 動作の方は直ちに[7,P,16,NP,10,V,5,NP,0]を得て終了する．この時，もう一方の reduce 動作を行なっているスタックは何ステップかの reduce を経て shift に到達し，[7,P,4,S,0]を得る．二つのスタックを統合するために，shift 動作で複数のプロセスが同期をとると，$O(n^2)$の時間を要する[峯 89]．現在の merge_stack の実現はそのようになっている．しかし，GHC のストリーム通信の機能を生かせば，最初の'sh 7'が実行された時点で merge_stack の出力を

  [[7,P|X]|Y]

  というようにして，shift で同期を取らずに先に出力し，後でもう一方のスタックトップが 7 と決ったら

  X=[[[4,S,0],[16,NP,10,V,5,NP,0]]]
  Y=[]

  というようにすることによって並列度を上げることも可能である．このようなインプリメントの方法は今後の課題である．

パージングプロセスは，各入力語に対応する action プロセスを並列に呼び出すことによって実行を開始する．今，"I open the door."を解析しようとする時，各単語の辞書引きを行なって，次のような action プロセスを呼ぶ．

	det	noun	pron	v	p	relp	$	NP	PP	VP	RELC	S
0	sh1	sh2	sh3					5				4
1		sh6										
2				re6	re6	re6	re6					
3				re7	re7	re7	re7					
4					sh7		acc		8			
5				sh10	sh7	sh9			12	11	13	
6				re5	re5	re5	re5					
7	sh1	sh2	sh3					14				
8					re2		re2					
9				sh10						15		
10	sh1	sh2	sh3					16				
11					re1		re1					
12				re4	re4	re4	re4					
13				re3	re3	re3	re3					
14				re10	sh7/re10	sh9/re10	re10		12		13	
15				re9	re9	re9	re9					
16				re8	sh7/re8	sh9/re8	re8		12		13	

図 2: LR パージングテーブル

```
?- pron(0,[0,[]],[pron,'I'],Sate1,Stack1),
   v(State1, Stack1, [v,open], State2, Stack2),
   det(State2, Stack2, [det,the], State3, Stack3),
   noun(State3, Stack3, [noun,door], State4, Stack4),
   $(State4, Stack4, [$,'.'], _, Result).
```

ここで，$は文の終了を示す記号である．各プロセスの第1引数はスタックトップの状態，第2引数は入力スタック，第3引数は木の情報，第4引数は次の状態，第5引数は出力スタックである．第1引数と第4引数はスタックトップの値と等しいので冗長であるが，述語の検索に第1引数の hash を用いる処理系上で有利になるので与えた．

action プロセスはまず状態0から開始され，先読み語'pron'をシフトすると次のプロセス'v'が呼ばれる．全てのプロセスが終了すると，解析結果として構文木の情報が'Result'に返される．このように，各単語に対応するプロセスを呼び出す点は松本の SAX,PAX[Matsumoto 88]と同じである．SAX,PAX では各文法規則に割り付けられた識別子のリストをプロセス間で受渡ししているのに対し，GLP ではスタックを受渡しする．

次に GHC による各パージングプロセスの記述を示す．

4.1 action プロセスの記述

- shift エントリ

  図2のテーブルでスタックトップが0で，先読み語の品詞が noun の時，テーブルのエントリは'sh 2'なので，このエントリに対する action プロセスは以下のように記述される．

```
noun(0, Stack, T, NS, NStack) :- true |
    NS=2, NStack=[2,T| Stack].
```

  このプロセスが呼ばれると，入力スタックに木の情報 T と次の状態2を積んで出力スタックとし，プロセスを終了する．

- reduce エントリ

  図2で状態が2，先読み語の品詞が v の時，テーブルのエントリが're 6'なので，6番目の規則を用いてスタックを還元する．

```
v(2, [_,T1| Stack], T, NS, NStack) :- true |
    reduce(1, 6, Stack, [T1], NS1, NStack1),
    v(NS1, NStack1, T, NS, NStack).
```

  ここでは，reduce プロセスを呼んでスタックを還元した後，再び v で始まる action プロセスを呼び出す．これは，還元動作においては先読み語が消費されないからである．GHC の AND 並列性により，プロセス reduce とプロセス v が並列に呼び出されるが，プロセス reduce において，NS1 が具体化されるまで，プロセス v の実行は中断される．プロセス reduce の第1引数は還元すべきスタックの要素数を示し，還元に用いる文法規則の右辺の文法カテゴリの数と一致する．第2引数は還元に用いる規則の番号を示し，第3引数はスタックからポップされた木の情報のリストを与える．

- shift-reduce コンフリクトのあるエントリ

  図2で状態 14 で先読み語の品詞が p の時，テーブルのエントリが'sh 7/re 10'なので，同じスタックを二つ作り shift 7 と reduce 10 とを別々に行なう必要がある．

```
p(14, [14, T1| Stack], T, NS, NStack):- true |
    reduce(2, 10, Stack, [T1], S1, Stack1),
    p(S1, Stack1, T, _, Stack2),
    merge_stack([7,T,14,T1|Stack],Stack2,NS,NStack).
```

  プロセス reduce と p の呼び出しは reduce エントリと同じであり，p(S1,...), の実行は何回かの還元動作を経て，シフト動作が呼ばれた時点で終了する．これによって得られた'Stack2'と入力スタックに shift 7 を施したスタック[7,T,14,T1|Stack]とを merge_stack プロセスに渡す．これによって得られる'NStack'はリスト表現上は木構造化スタックであるが，内部的にはグラフ構造化スタックとなっている．これは以下の理由による．

  スタックを二つに分ける時，すなわち，プロセス reduce の第3引数と merge_stack の第1引数に'Stack'を渡す時，この二つのスタックは内部的には同じメモリを参照している．'Stack2'は，'Stack'の先頭のいくつかの要素を更新したものなので，スタックの底に近い部分はやはり同じメモリを参照する．従って merge_stack で得られる'NStack'はその先頭と底を共有するグラフ構造化スタックとなっている．

- reduce-reduce コンフリクトのあるエントリ

  図2のテーブルにはないが，仮に先読み語が v，状態が2の時，'re 5/re 6/re 7'というエントリがあったとしよう．action プロセスの記述は

```
v(2, [2, T1| Stack], T, NS, NStack):- true |
    reduce(2, 5, Stack, [T1], _, Stack1),
    reduce(1, 6, Stack, [T1], _, Stack2),
    reduce(1, 7, Stack, [T1], _, Stack3),
    merge([Stack1],[Stack2],[Stack3],StackList),
    v([list], StackList, T, NS, NStack).
```

  となる．それぞれの還元動作を行なった後，組み込み述語 merge によって，スタックを一本のリストにまとめる．次の action プロセス v にはスタックのリストを渡すことになる．第1引数はそのことを示している．スタックのリストの扱いについては後で述べる．

- accept エントリ

  図2で状態が4，先読み語が文の終了を示す$の時，

```
$(4, [_,Tree|_], _, _, Result):- true |
    Result=Tree.
```

  と記述する．これによって木の情報を返し，プロセスを終了する．

- error エントリ

  エントリが空白の時は，各先読み語ごとにまとめて次のような一つのプロセスの記述になる．例えば det に対しては，det に対する action プロセスの記述の後に以下を加える．

```
otherwise.
det(S,Stack,_,NS,NStack):- integer(S) |
        NS=[],NStack=[].
```

  'otherwise.'によって，det(S,...)の呼び出しが，その上に記述されているプロセス det の全ての定義にマッチしなかった場合にこの定義が呼び出され，状態とスタックに[]を返してプロセスを終了する．

- スタックのリストの扱い

文の解析の途中でコンフリクトを通過すると，スタックは複数になる．action プロセスに複数のスタックのリストが与えられた時，これを次のように処理する．

```
otherwise.
det([],[],_,_,NS,NStack):- true |
    NStack=[],NS=[].
det(_,_,[[]],_,_,NS,NStack):- true |
    NStack=[],NS=[].
det(_,_,[[]|Stack],T,NS,NStack):- true |
    det([list],Stack,T,NS,NStack).
det(_,_,[[S | Stack]],T,NS,NStack):- true |
    det(S,[S | Stack],T,NS,NStack).
det(_,_,[[S | Stack] | Rst],T,NS,NStack):- true |
    det(S,[S | Stack],T,_,NStack1),
    det([list],Rst,T,_,NStack2),
    merge_stack(NStack1,NStack2,NStack),
    NStack=[NS |_].
```

ただし，最後の定義中の merge_stack の呼び出しは，プロセス名が$のときは，

```
    merge([NStack1],[NStack2],NStack),
```

とする．この定義を error エントリの後に加える．最初から三つ目までのプロセスの定義は，解析の途中でエラーとなったプロセスを処理するためのものである．後の二つの定義は複数のスタックを一つずつ取り出して action プロセスを呼び出すためのものである．複数のスタックの処理は並列に実行される．

4.2 reduce プロセスの記述

reduce プロセスは指定された文法規則を用いて，与えられたスタックの要素を先頭から必要な数だけポップし，木の情報を取り出してより大きな木を作る．次に goto プロセスを呼んでポップされたスタックと作られた木を渡す．スタックが木構造化されている場合，還元動作がスタックの分岐点に及ぶと，その点からスタックを複数に分けてそれぞれのスタックについて並列に還元動作を実行する．

これを具体的に説明する．例えば，

```
(top) [14,NP,7,P,[[4,S,0,[]],
                  [16,NP,10,v,5,NP,0,[]]]]
```

のような木構造化スタックの先頭の 2 要素をポップすると，スタックは二つに分離し，

$$[4, S, 0, []]$$
$$[16, NP, 10, v, 5, NP, 0, []]$$

となる．

reduce プロセスは，次に示すプロセス'reduce'とプロセス're'によって実現される．プロセス'reduce'はスタックをポップし，木の情報を取り出してプロセス're'を呼ぶ．

```
reduce(_, _, [[]], _, NewSt):- true|
    NewSt=[].
otherwise.
reduce(1, N, [S, T1| St], T, NewSt):- integer(S)|
    re(N, [S, T1| St], T, NewSt).
otherwise.
reduce(1, N, [[St| Rst]], T, NewSt):- true|
    reduce(1, N, St, T, NewSt1),
    reduce(1, N, [Rst], T, NewSt2),
    NewSt=[NewSt1| NewSt2].
otherwise.
reduce(M, N, [S, T1| St], T, NewSt):- integer(S)|
    M1 := M-1,
    reduce(M1, N, St, [T1| T], NewSt).
otherwise.
reduce(M, N, [[St| Rst]], T, NewSt):- true|
    reduce(M, N, St, T, NewSt1),
    reduce(M, N, [Rst], T, NewSt2),
    NewSt=[NewSt1| NewSt2].
```

プロセス're'は文法規則に一対一に対応しており，'reduce'で取り出された木の情報と文法規則に従ってより大きな木を作り，goto プロセスを呼び出す．図 1の規則に対応するプロセス're'の記述は以下のようになる．

```
re(1,[S |Stack],T,S,NStack):- true |
    sentence(S,[S |Stack],[sentence |T],S,NStack).
re(2,[S |Stack],T,S,NStack):- true |
    sentence(S,[S |Stack],[sentence |T],S,NStack).
re(3,[S |Stack],T,S,NStack):- true |
    np(S,[S |Stack],[np |T],S,NStack).
re(4,[S |Stack],T,S,NStack):- true |
    ・・・・・
otherwise.
re(_,_,S,NStack):- true |
    S=[], NStack=[].
```

プロセス're'の body 部のプロセスの呼びだしは goto プロセスの呼びだしである．goto プロセスは LR テーブルの goto 部のエントリの実行を行なうもので，プロセス名はスタックの還元に用いる文法規則の左辺の非終端記号名である．プロセス're'は，スタックをポップして得られた木を上に伸ばして，goto プロセスに渡している．'otherwise.'の後の記述は，DCG を扱う際に必要となる．

4.3 goto プロセスの記述

goto プロセスは，LR テーブルの goto 部のエントリに一対一に対応し，その記述は shift エントリの記述と同じである．例えば図 2 で非終端記号名が NP，状態番号が 0 の時エントリは 5 なので，

```
np(0, Stack, T, NS, NStack):- true |
    NS=5, NStack=[5,T | Stack].
```

と記述する．ここでは，スタックに還元動作によって生成された木と次の状態番号を積み，action プロセスを呼んでいる．

4.4 merge_stack プロセスの記述

merge_stack プロセスのアルゴリズムの記述については，今後変更する可能性があるので付録に示すにとどめる．

5 DCG の扱いと問題点

我々の扱っている DCG 規則は各文法記号に対して 2 つの引数を持ち，さらに→の右側の任意の場所に補強項と呼ばれるプロセスの呼び出しを加えることができる．例えば，図 1の最初の文法規則の DCG による記述の一つは，

(1) s(A_A , A_S) →
np(NP_A , NP_S),
{ プロセス呼びだし },
vp(VP_A , VP_S).

のようになる．各文法記号の第一引数は構文的な情報を運び，第二引数は意味的な情報を運ぶ．補強項ではこの規則の適用の可否を調べ，規則の左辺に与える構文情報と意味情報を生成する．

GLPにおける引数の扱いは，スタックに積む木の情報Tを，

Arg = [第一引数, 第二引数, T]

に改め，補強項は還元動作の際に実行する．これを行なうために，プロセス're'を，

```
re(1,[I|St],[[NP_A,NP_S,NP_T],[VP_A,VP_S,VP_T]],
    S,NStack):-
    (プロセス呼びだし A)|
    (プロセス呼びだし B),
    s(I,[I|St],[S_A,S_S,[s,NP_T,VP_T]],S,NStack).
```

とする．ここで，プロセス呼びだしAはその成功／失敗によって，この規則の適用の可否を調べる補強項のプロセスであり，Bは新たな構文情報と意味情報を生成するためのプロセスである．このように，補強項を二種類に分けるためには，文法とは別に補強項の識別を行なうための情報をあらかじめ与えておく必要がある．また，GHCではbody部にあるプロセスBには失敗が許されないので注意が必要となる．

なお，我々の使用している処理系は'flat GHC'と呼ばれ，ガード部にプロセスAを置くことができないので，DCGの扱いは次に述べるGLPのProlog版で実現しているのみである．

GLPにおけるDCGの扱いの問題点として，環境のコピーの問題と，補強項が複数の解を持つ場合があげられる．環境のコピーの問題については，[沼崎 89] あるいは[松本 86]を参照されたい．補強項が複数の解を持つ場合には，一回の計算で全解を得られるように補強項を記述しておく必要がある．この点については，今後検討すべき課題として残されている．

6 Prolog上への移植

GLPは上に示したGHCによる記述に対し，以下の点を変更することにより，prolog上に実現できる．

- 'true |'を'!,'に置き換える
- ボディ部の変数の束縛をヘッドに移す．例えば，
  ```
  noun(0, Stack, T, NS, NStack) :- true |
      NS=2, NStack=[2,T| Stack].
  ```
  を
  ```
  noun(0, Stack, T, 2, [2,T| Stack]) :- !.
  ```
  と変更する．
- reduce-reduce コンフリクトのスタックの merge を差分リストによって行なう．
  ```
  v(2, [2, T1| Stack], T, NS, NStack):- true |
      reduce(2, 5, Stack, [T1], _, StackList, RSt1),
      reduce(1, 6, Stack, [T1], _, Rst1, RSt2),
      reduce(1, 7, Stack, [T1], _, RSt2, []),
      v([list], StackList, T, NS, NStack).
  ```
  これに伴い7引数のプロセスreduceの定義を与えておく．

7 他の自然言語処理システムとの比較

我々はGLPをQuintus Prolog(Ver. 2.2)上でコンパイルして走らせ，規則数550程度のDCG形式の実用的な英語の文法規則を用いて，英語の文の解析時間を，松本のSAX[松本 86]と比較した．英語の文としては次を用いた．これは奥西の報告[Okunishi 88]から引用した．実験にはSun-3(260)を使用した．

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.

実験は各文を5回ずつ解析し，解析時間の平均をとった．

文	GLP	SAX	木
1	37(ms)	140(ms)	1
2	20	120	1
3	157	454	4
4	83	323	1
5	260	566	4
6	147	403	1
7	626	1643	5
8	583	990	4
9	283	1209	8
計	2196	5848	

実験の結果GLPはSAXの平均2.5倍程度速いことが明らかになった．Quintus Prolog上のシステムの静的なメモリ消費量はGLPが4M byte，SAXが800K byteであった．また，曖昧さの多い文の解析時間を調べるため，次のような実験を行なった．

NP,v,NP
NP,v,NP,PP
NP,v,NP,PP,PP
NP,v,NP,PP,PP,PP
・・・

のように，入力文のPPの数を0から9まで単純に増やした時

の処理時間の推移を比較した．図3にその結果を示す．

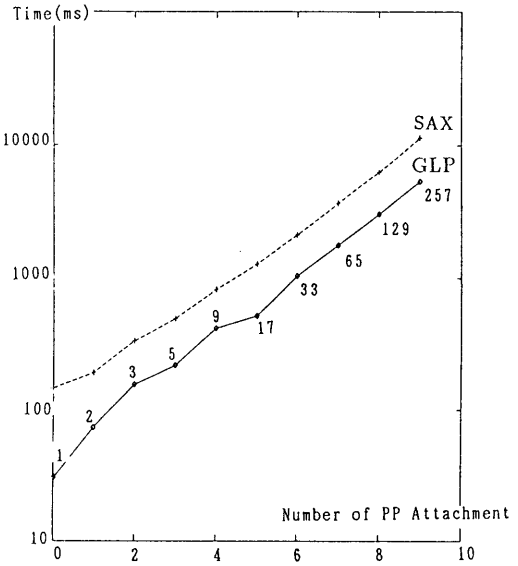

図 3:解析時間の推移

グラフの縦軸の目盛は対数をとっているので，一目盛で10倍の時間がかかっていることになる．グラフ中の数字は生成された木の数を示している．このように曖昧さの多い文を解析してもGLPの優位性はかわらなかった．

GLPの方が速い理由を検討してみよう．SAXとGLPの決定的な違いは規則の適用時期にある．これを具体的に説明するすると以下のようになる．今，図1の文法に対し，

"I open the window."

という文を"I"まで解析したとしよう．このとき，NPというカテゴリが作られるが，SAXの場合この時点で，

(1) S →NP (id1) VP.
(4) NP→NP (id4) PP.

の二つの規則を適用し，id1とid4をリストに加える．しかし，id4の方は次の"open"を解析する時点で排除される．その際，id4を排除すべきかどうかを判定するのにわずかではあるが時間を要する．文法規則が大きくなり，(4)と同じような規則が多数存在すると，不要なidを多く抱えることもある．

一方GLPの方は，"open the window"の解析を終え，VPができて初めて(1)の規則を適用する．従って，(4)の規則は適用されない．この点が異なる．

GLPには規則の数が多くなるとテーブルのエントリが膨大になる(規則数550で2万個程度)という欠点があるが，実験で使用したQuintus Prologの場合，述語名と第一引数でダブルhashを行ない，テーブルの参照を高速に行なえるため，この欠点が表面化しなかったようである．一方SAXは，プロセスの第一引数がリストなのでダブルhashの利点を生かしていない．このことも実験結果に反映されたようである．

8 結論と今後の課題

本研究では曖昧さのある自然言語を並列に解析するパーザGLPを提案し，これを並列論理型言語の枠組において記述した．

既存の自然言語処理方式との比較により，GLPの有用性が確認された．

今後の課題としてGLPをマルチPSIなどの並列マシン上に実現し，並列性を調べること，また，英語の関係詞節にみられる左外置の処理[今野 86]を本方式に組み込むこと，さらに，意味処理において複数の解析結果が得られる場合を扱うようにする点があげられる．

謝辞

本研究を進めるにあたり，GHCの処理系PDSSを提供して下さったICOT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

[今野 86] 今野 聡，田中穂積:左外置を考慮したボトムアップ構文解析，コンピュータソフトウェア，Vol.3, No.2, pp.115-125 (1986)

[中田 81] 中田育男:コンパイラ，産業図書 (1981)

[沼崎 89] 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)

[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)

[松本 86] 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)

[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)

[安留 88] 安留誠吾，青江順一: 自然言語の曖昧構文解析に対する並列処理，情報処理学会ソフトウェア基礎論研究会研究報告，27-12, pp.109-118 (1988)

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

[Aho 85] Aho,A.V.,Senthi,R.and Ulman,J.D.: *Compilers Principles,Techniques,and Tools,*Addison-Wesley (1985)

[Knuth 65] Knuth,D.E.: *On the translation of languages from left to right,*Information and Control 8:6,pp.607-639

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

[Nilsson 86] Nilsson,U.: *AID:An Alternative Implementation of DCGs,* New Generation Computing, 4, pp.383-399 (1986)

[Okunishi 88] Okunishi,T.,Sugimura,R.,Matsumoto,Y.: *Comparison of Logic Programming Based Natural Language Parsing Systems*,Natural Language Understanding and Logic Programming,II V.Dahl and P.Saint-Dizer(Editors) Elsevier Science Publishers B.V.(North-Holland), pp.1-14 (1988)

[Pereira 80] Pereira,F.and Warren,D.: Definite Clause Grammar for Language Analysis-A Survey of the Formalism and a Comparison with Augmented Transition Networks, *Artif. Intell*, Vol.13, No.3, pp.231-278 (1980)

[Tomita 86] Tomita,M.:*Efficient Parsing for Natural Language*, Kluwer Academic Publishers (1986)

[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 : merge_stack プロセスの記述

```
  merge_stack([],[],NS,NStack):- true |
     NS=[],NStack=[].
merge_stack([],[I|MStack],NS,NStack):- true |
     NS=I,NStack=[I|MStack].
merge_stack([I|Stack],[],NS,NStack):- true |
     NS=I,NStack=[I|Stack].
merge_stack([S |St],MStack,NS,NStack):- integer(S) |
     merge_stack1([S |St],MStack,NStack),
     NStack=[NS|_].
otherwise.
merge_stack([St |Rst],MStack,NS,NStack):- true |
     merge_stack(St,MStack,_,NStack1),
     merge_stack(Rst,NStack1,NS,NStack).

  merge_stack1([],St,NStack):- true |
     NStack=St.
merge_stack1(St,[],NStack):- true |
     NStack=St.
merge_stack1([S,T |St],[S,T |St1],NStack):-
     integer(S)|
     merge_stack3(St,St1,St2),
     NStack=[S,T,St2].
otherwise.
merge_stack1(St,[S1 |St1],NStack):- integer(S1) |
     NStack=[St,[S1 |St1]].
merge_stack1([S,T |St],[[S,T |St1] |MStack],
     NStack):- true |
     merge_stack3(St,St1,St2),
     merge_stack2([S,T,St2],MStack,NStack).
otherwise.
merge_stack1(St,[St1 |MStack],NStack):- true |
     merge_stack1(St,MStack,NStack1),
     merge_stack2(St1,NStack1,NStack).

  merge_stack2(St,[],NStack):- true |
     NStack=St.
merge_stack2(St,[S |Rst],NStack):- integer(S) |
     NStack=[St,[S |Rst]].
otherwise.
merge_stack2(St,MStack,NStack):- true |
     NStack=[St |MStack].

  merge_stack3([S |Rst],St2,NStack):- integer(S) |
     merge_stack4([S |Rst],St2,NStack).
merge_stack3(St1,[S |Rst],NStack):- integer(S) |
     merge_stack4([S |Rst],St1,NStack).
otherwise.
merge_stack3([[]],St2,NStack):- true |
     NStack=St2.
merge_stack3([[St1 |Rst]],St2,NStack):- true |
     merge_stack3(St1,St2,NStack1),
     merge_stack3([Rst],NStack1,NStack).

  merge_stack4(St,[S |Rst],NStack):- integer(S) |
     NStack=[St,[S |Rst]].
otherwise.
merge_stack4(St,[St1],NStack):- true |
     NStack=[St |St1].
otherwise.
merge_stack4(St,St1,NStack):- true |
     NStack=[St |St1].
```