# 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

# An Implementation of LR-Parsing Algorithm for Natural Language Processing Based on Parallel Logic Programming Language

沼崎 浩明
Hiroaki NUMAZAKI
東京工業大学工学部
Tokyo Institute of Technology

田村 直良
Naoyoshi TAMURA
横浜国立大学工学部
Yokohama National University

田中 穂積
Hozumi TANAKA
東京工業大学工学部
Tokyo Institute of Technology

**概要**

LR構文解析法[Aho 72]を自然言語に適用するには，コンフリクトを扱えるように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．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．GHCのすぐれた記述力により，アルゴリズムの記述は容易であった．簡単な実験によって，松本のPAX[Matsumoto 88]との比較を行ない，本方式の有用性を確認した．

## 1 まえがき

LR構文解析法はLR文法と呼ばれる文脈自由文法のサブセットに対して決定的な解析を可能にする．一般に自然言語の文法はLRでないため，LR法を自然言語に適用すると，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ある．そのため，LR法を自然言語処理に用いるには，コンフリクトを扱えるようにアルゴリズムを一般化する必要がある．一般化には二つの方法があり，一つは[Nilsson 86]のように曖昧さの生じた点をバックトラックポイントとして，パーザの動作を非決定的に行なう方法と，もう一つは富田法のようにスタックを複数持ち横型的な戦略で処理を進めていく方法である．本研究では後者の方法が並列処理との整合性に優れている点に着目し，これを用いて並列処理を実現するアルゴリズムを提案する．アルゴリズムは並列論理型言語GHC[Ueda 85]によって記述する．我々は自然言語の文法をDCG[Pereira 80]によって記述し，この文法に基づいてLALRパージングテーブルを生成する．テーブルの各エントリをGHCの節として記述し，文法規則もGHCの節に変換する．

富田法では，複数のスタックを結合してグラフ構造化することにより，文の解析の重複を避ける機構を設けている．本研究ではこのグラフ構造化スタックを我々のアルゴリズムに組み込み，処理の効率化を図る．富田法を並列処理に適用した研究として安留ら[安留 88]の報告があるが，本研究では安留らの方法とは異なったアプローチをとっている．

本論文ではまずLR構文解析法について述べ，次に，これを並列処理に適用するアルゴリズムをGHCによる記述とともに説明する．さらに，本方式にグラフ構造化スタックを組み込む方法について述べ，安留の方法との違いについて言及する．

アルゴリズムの説明には，その理解を容易にするために文脈自由文法を用い，その後でDCGを扱う方法とその問題点について述べる．そして，本方式の処理速度を簡単な実験を通して松本のPAX[Matsumoto 88]と比較し，その有用性を確認する．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構文解析法は最も効率的な構文解析アルゴリズムの一つとして知られている．その特徴は与えられた文法規則をコンパイルして，パーザがどのように動けばよいかを示すテーブルをあらかじめ生成する点にある．構文解析はこのテーブルに基づいて進められる．

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と呼ばれる動作を決定できないエントリが生ずる．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求まる．'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．これをどう扱うかによってパーザのアルゴリズムが異なってくる．これには前節で述べたようにコンフリクトの生じた点をバックトラックポイントとしてパーザに非決定的な動作をさせる縦型的な方法と，その点で複数の処理を同時に進めていく横型的

(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

な方法とがある．富田法は横型であり，複数のプロセスに優先度を設け，これを逐次的に処理している．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．

## 3 並列 LR 構文解析アルゴリズムの実現

本研究で実現した並列 LR 構文解析アルゴリズムの特徴は，パーザの動作にコンフリクトが生じた時，パージングプロセスを分岐して，以後複数のプロセスで構文解析を進めていく点にある．これは GHC の特徴である AND 並列性を利用して記述される．プロセスが複数に分かれたとき，スタックをどう扱うかが問題となるが，ここでは最も単純な方法として，分岐した各プロセスにそれぞれ一つずつスタックを割り当てる．

従来の LR パーザはテーブルとそれを駆動するパーザを別々に記述するのが普通であるが，本方式では，テーブルの各エントリをパージングプロセスの記述に置き換えることによって実行速度の向上を図る．

パージングプロセスは次の三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，スタック上に入力を取り込み，状態遷移を行なって．次の action プロセスを呼び出す．reduce 動作に対しては，reduce プロセスを呼びだし，スタックを渡す．コンフリクトがある場合には，同時に複数のプロセスを呼びだし，それぞれにスタックを渡す．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，直ちに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，goto プロセスを呼び出す．

- goto プロセス
  状態遷移を行なって次の action プロセスを呼び出す．

以下に GHC によるパージングプロセスの記述を示す．

### 3.1 action プロセスの記述

action プロセスは入力文の辞書引の結果として得られる単語の品詞の列を入力として受け取り，実行を開始する．

テーブルのエントリが shift の時，例えば，図 2 ではプロセスの状態が 0 で，先読み語の品詞が noun の時，テーブルのエントリは'sh 1'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．

```
i0(noun, Stack, [_,NextCat|Dict], Info) :- true |
    i1(NextCat, [[1,noun]|Stack], [NextCat|Dict], Info).
```

述語の第一引数が先読み語，第二引数がスタック，第三引数が入力語の品詞のリスト，第四引数が解析木のリストである．ここでスタック上に入力語の品詞と次の状態を積み，入力語のリストを一つ消費して次の action プロセスを呼び出している．

エントリが reduce の時，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，

```
i2(v, Stack, Dict, Info) :- true |
        re6(v, Stack, Dict, Info).
```

のように記述される．ここでは入力は消費されず，また，スタックはそのまま reduce プロセスに渡される．

shift-reduce コンフリクトがある場合，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'sh7/re10'なの

| | 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

で,

```
i14(p, Stack, [p,NextCat|Dict], Info) :- true |
        i7(NextCat, [[7,p]|Stack], [NextCat|Dict], Info1),
        re10(p, Stack, [p,NextCat|Dict], Info2),
        merge(Info1, Info2, Info).
```

となり,GHCのAND並列性によって,actionプロセスとreduceプロセスが並列に呼び出されて実行される.この例では呼び出されるreduceプロセスの数が一つであるが,文法規則によっては同時にreduce-reduceコンフリクトを生ずることもある.その場合は,下のようにreduceプロセスの呼び出しが複数になる.

reduce-reduceコンフリクトの場合は,コンフリクトの数だけreduceプロセスが呼び出される.今,状態Sで先読みがcatの時,エントリが're R1/re R2/ .../re RN'の場合,

```
iS(cat, Stack, Dict, Info) :- true |
        reR1(cat, Stack, Dict, Info1),
        reR2(cat, Stack, Dict, Info2),
                . . .
        reRN(cat, Stack, Dict, InfoN),
        merge(Info1, Info2, ..., InfoN, Info).
```

となる.

エントリがacceptの時,図2では状態が4,先読み語が文の終了を示す$の時,

```
i4($, [[_,Tree]|_], _, Info) :- true |
        Info=[Tree].
```

と記述する.これによって第四引数に木の情報が返され,途中でコンフリクトが生じていた場合にはmergeプロセスによって複数の木の情報が一つのリストに統合される.

エントリが空白(エラー)の時は,各状態番号ごとにまとめて次のような一つの述語によって記述する.例えば状態0に対しては,上に示したようなエントリの記述の後に以下を加える.

```
otherwise.
i0(_, _, _, Info) :- true |
        Info=[].
```

我々の使用しているGHCの処理系であるPDSSの構文otherwise.により,i0(cat, ...)の呼び出しがその上に記述されている述語i0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,第四引数に[]を返してプロセスを終了する.

### 3.2 reduceプロセスの記述

reduceプロセスの記述は文法規則に一対一に対応する.例えば,(1) S→NP,VP. は,

```
re1(NextCat, [A1,A2,A3|Stack], Dict, Info) :- true|
    A1=[_, T1],
    A2=[_, T2],
    A3=[State, _],
    s(State, NextCat, [s,T2,T1], [A3|Stack], Dict, Info).
```

のようなGHC節に変換される.ここで,述語名're1'の1は文法規則の番号である.第二引数でスタックの先頭から三つの要素を取り出している.ボディ部で,先頭の二つの要素から部分木の情報を取り出し,三つ目の要素からは状態番号を取り出す.述語sの呼び出しはgotoプロセスの呼びだしである.ここで,述語名sは文法規則の左辺の非終端記号名である.その第三引数にこの規則で生成される木の情報を渡し,第四引数でポップした後のスタックを渡す.ここでは,スタックの先頭から二つの要素がポップされているが,それは文法規則の右辺の要素の数と等しい.そして,第一引数にはポップしたスタックトップの状態番号を渡す.

### 3.3 gotoプロセスの記述

上に示したようにgotoプロセスはテーブルを参照するための非終端記号名を述語名に,状態番号を第一引数にして記述される.例えば図2で非終端記号名がS,状態番号が0の時エントリは4なので,

```
s(0, NextCat, Tree, Stack, Dict, Info) :- true |
        i4(NextCat, [[4,Tree]|Stack], Dict, Info).
```

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.

パージングプロセスは以上のように記述される.第7節で,簡単な実験により,本方式と松本のPAX[Matsumoto 88]との処理速度の比較を行なう.本方式を実用化する際に問題となるのは,文法規則の数が多くなるとテーブルのエントリの数が増大し,パージングプロセスの記述に膨大な空間を必要とする点である.実験によると,この方法で550の文法規則をテーブルに変換し,パージングプロセスを記述すると約3Mバイトの空間を必要とした.これを減らすにために,パージングプロセスの記述とテーブルの記述とを分離すれば,約1Mバイトまで減らすことができる.しかしこの場合,前者と比較して実行速度は低下する.

## 4 グラフ構造化スタックの導入

前節までで実現した一般化LR構文解析アルゴリズムは,プロセスが分岐する時スタックのコピーを作り,それぞれをパージングプロセスに渡して処理を進めていく.しかし,[Tomita 86]に指摘されているように各プロセスが持ち歩くスタックには共通の部分が多く存在しており,これを別々に持ち歩くのでは空間的な効率が悪い.また分岐したプロセスが途中で同じ状態になったとき,それ以後の入力のシフト動作は全く同じように行なわれる.この非効率性を改善するために富田はスタックの同一の部分を結合したグラフ構造化スタックを提案した.

富田法を並列処理に適用した研究として安留ら[安留 88]の報告がある.安留のアルゴリズムの特徴はグラフ構造化スタックを分割して階層化された複数のプロセスに割り当てている点にある.shift動作はルートのプロセスのみで行ないコンフリクトが生じた時にサブプロセスが生成される.この時,ルートプロセスはスタックの長さを記憶しておく.サブプロセスは還元動作のみを行ない,一旦生成されたサブプロセスは統合されない.そして,サブプロセスが生成された後でルートプロセスが取り込んだ入力に還元動作が生じ,その還元動作がスタックを記憶された長さよりも短くする時,還元すべきスタックの要素をサブプロセスに伝搬し,ルートプロセスとサブプロセスで同じ還元動作を行なう.このアルゴリズムは複数のプロセッサが別々のメモリを管理しながら並行に動作するハードウェアを想定して作られており,テーブルも各プロセッサが個々に保持している.この方法の問題点を指摘するとすれば,実用的な文法規則で生成されるテーブルは大きい

ので（エントリの数が数万のオーダ）これを各プロセッサが一つずつ持つのはコストがかかるという点と，解析する文の曖昧さがプロセッサの数を越えた場合どう処理するのかという点があげられる．

これに対し，本研究ではアルゴリズムを並列論理型言語の枠組で記述するので，ハードウェアの構成は問わない．本研究の特徴をまとめると，以下のようになる．

- グラフ構造化スタックをパージングプロセスとは別に，一つの独立したプロセスとして実現する．これに辞書プロセスを加えた三つのプロセス（図3参照）により解析が行なわれる．
- コンフリクトが生じると，パージングプロセスが複数に分岐する．これをGHCのAND並列性を利用して記述することは既に示した．
- パージングプロセスはグラフスタックへのポインタを持ち歩き，テーブルに応じてshift,reduceのなど要求をグラフスタックプロセスに送る．
- 分岐したパージングプロセスは入力語を同時にシフトするように同期をとる．この同期は辞書プロセスが管理する．これはGHCのストリームによるプロセス間の通信とメッセージの待ち合わせ機能を利用して実現される．
- 複数のパージングプロセスが入力語をshiftする際，同じ状態に推移する場合は，プロセスを一つに統合する．これと同時にスタックも統合され，グラフ構造化される．
- グラフスタックプロセスは還元動作の際，スタックの要素をポップせずにパージングプロセスが持っているポインタをスタックの根の方に必要な数だけ戻すだけである．

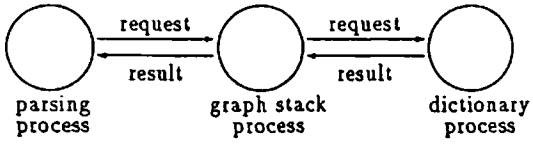


図3 三つのプロセス

以下では，まずグラフ構造化スタックについて説明し，それを実現するプロセスについて述べる．次にパージングプロセス，辞書プロセスの順で述べる．

### 4.1 グラフ構造化スタックについて

ここでは，[Tomita 86]で導入されたグラフ構造化スタックについて説明する．グラフ構造化スタックは次に示すような情報を持つ節点の集合として表される．

節点番号 N ：木の情報 T ：状態番号 I ：接続リスト L

グラフ構造化スタックの最初の節点は，

節点番号 0 ：木の情報 [] ：状態番号 0 ：接続リスト []

である．一つの節点を

N : (T, I, L)

という形式で表すとすると，図4に示されるような構造を持つグラフ構造化スタックの各節点は次のようになる．

0 : ([], 0, [])　　1 : (T1, I1, [0])
2 : (T2, I2, [1])　　3 : (T3, I3, [1])
4 : (T4, I4, [3])　　5 : (T5, I5, [2])
6 : (T6, I6, [4, 5])　　7 : (T7, I7, [6])

本方式では還元動作の際スタックの要素をポップしないのでスタックの節点の数は，文の解析が進むにつれて単調に増加する．

次に，このようなグラフ構造がどのようにして得られるかを簡単に示す．詳細は，付録のグラフ構造化スタックを用いた構文解析過程の説明を参照されたい．まず，スタックの分岐について説明する．スタックの分岐はパーザの還元動作の際に生ずる．図2のテーブルに対し，det, noun, v, ...のような品詞を持つ入力語の並びがあるとき，解析の途中の段階でスタックは図5のようになる．(この図及び以下のスタックの図で，det，noun，NPなどの品詞の部分には実際は木の情報が入る．)図5に対してNP → det，noun.の文法規則を用いて還元動作を行なう．この時，パージングプロセスが保持するグラフスタックへのポインタ（節点番号）は2から0に移る．そして，節点0の状態が0，還元動作によって生成された非終端記号がNPなので，gotoテーブルを参照して状態5へ移る．このとき，スタックは図6のように分岐する．これによって，上の枝はそれ以上伸びない不活性な枝となり，下の枝がこれから伸びる活性な枝となる．しかし，還元動作を行なう時点でシフト動作も許されるようなシフト-レデュースコンフリクトが生じていた場合には，どちらの枝も活性となる．

グラフスタックプロセスはどの枝が活性であり，どの枝が不活性であるかを知らない．枝が活性かどうかはいずれかのパージングプロセスがその枝の終端の節点番号を持っているかどうかによって決まる．

図5:分岐前のスタック

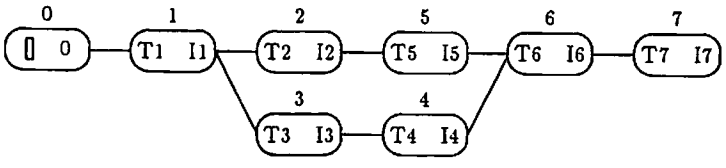


図4:グラフ構造化スタック

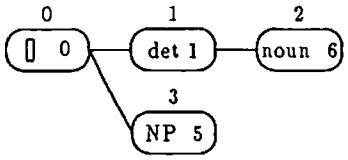



図 6:分岐後のスタック

グラフスタックの結合は，複数のパージングプロセスのシフト動作の際に，同じ状態に移るプロセスが存在した場合に生ずる．例えば，NP, v, NP, p, ...のような文を pのシフトの直前まで解析を進めた段階での活性なスタックは図 7 のようになり二つに枝分かれしている．各枝に対して一つのパージングプロセスが存在している．pをシフトしようとする段階でパージングプロセスはどちらも shift 7を実行しようとする．このとき，グラフスタックプロセスは一つの節点 10 : (p, 7, [7, 9])を生成し，一つのパージングプロセスの一つを終了させる．スタックが結合した後の様子を図 8 に示す．

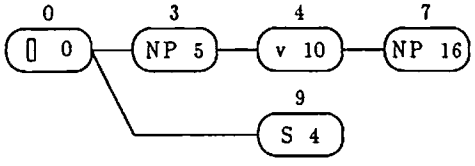



図 7:結合前のスタック

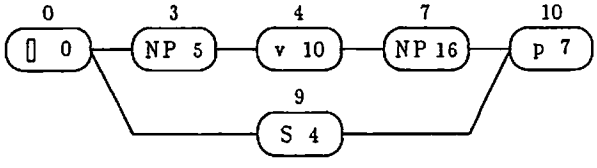



図 8:結合後のスタック

### 4.2 グラフスタックプロセスの実現

グラフスタックプロセスはパージングプロセスからの次の 6 つの要求によって駆動される．

- shift シフト動作に対して出される要求．
- pop 還元動作に対して出される要求．
- goto 還元動作後の状態推移に対して出される要求．
- accept 文の受理に対して出される要求．
- create パージングプロセスの分岐に対して出される要求．
- terminate パージングプロセスの停止に対して出される要求．

この要求は次のような変数を持ち，パージングプロセスとグラフスタックプロセスとの情報の交換，及びパージングプロセス制御が行なわれる．

shift(N, Top, NextTop↑, NextCat↑)
pop(Top, N, Poplist↑)
goto(N, Tree, Top, NextTop↑)
accept(Top, Info↑)

ここで↑の付いている変数はグラフスタックプロセスがパージングプロセスに値を返す変数である．また，グラフスタックプロセスは辞書プロセスとも次の情報を交換する．

request, NextCat↑, create, terminate

この request により辞書プロセスに次の入力単語の辞書引きを要求し，その結果を NextCat↑によって受け取る．上に示したグラフスタックプロセスへの 6 つの要求に対するプロセスの動作は次の通りである．

- shift(N, Top, NextTop↑, NextCat↑)
  N : シフト先の状態番号
  Top : スタックトップの節点番号
  NextTop↑ : 新しいスタックトップの節点番号
  NextCat↑ : 次の先読み単語の品詞

  まず，辞書プロセスに辞書引きの要求 request を通信する．スタック及びパージングプロセスの統合を次のように行なう．

  あるパージングプロセスがシフト動作によって状態 N へ推移する要求をグラフスタックプロセスに出している時，グラフスタックプロセスはそれより先に状態 N へ推移したパージングプロセスがあるかどうかを調べる．ある場合には，すなわち，状態番号 N を持つ節点 M:(Tree, N, L) がすでに存在する場合には，シフト要求を出したプロセスの持つスタックトップの番号 Top を，その節点の接続リスト L に加え，[Top|L]とすることによってスタックを統合する．さらに，パージングプロセスに NextCat↑ = [] を返す．これによってパージングプロセスは停止する（後述）．この機構により，ある入力語をシフトして同じ状態に推移しようとするパージングプロセスは一つに統合される．

  パージングプロセスが統合されない場合には，新しい節点番号 NextTop↑ を生成し，スタックに節点 NextTop:(Tree, N, [Top])を加える．ここで，現在の先読み語を Word，その品詞を Cat としたとき，Tree = [Cat,Word]である．そして，新しい節点番号 NextTop↑と，辞書プロセスが次に辞書引を行なった時に値を束縛する変数 NextCat↑とをパージングプロセスに返す．この変数が未束縛の間はパージングプロセスは中断される．この機構により，全てのパージングプロセスのシフト動作を同期させる．

- pop(Top, N, Poplist↑)
  Top : スタックトップの節点番号
  N : 還元動作に必要な節点の数
  Poplist↑ : スタックのトップから必要な数だけコピーされた節点のリスト．
  スタックの節点の接続情報にしたがって N 個だけスタックを後向きにたどり，得られた全てのパス上の節点のコピーを作り Poplist に蓄える．例えば，スタックが図 9 のような状態の時，pop(9,3,Poplist)を実行すると，

  Poplist↑=[[C',B',A'],[C',E',D'],[C',E',F']]

  となる．ここで各変数 A',B',C', ...は，

  A' = [A,IA,TA]

のようにその各節点の番号,状態,部分木の情報を持つ.

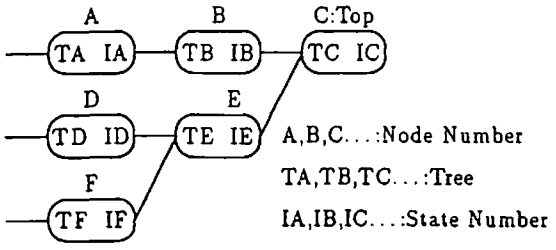



図 9:pop の処理

- goto(N,Tree,Top,NextTop↑)
  N : 還元動作の後のパージングプロセスの行き先の状態番号
  Tree : 還元動作によって得られた木の情報
  Top : スタックトップの節点番号
  NextTop↑ : 次のスタックトップの節点番号
  新しい節点番号 NextTop↑を生成し,スタックに節点 NextTop : (Tree , N , [Top])を加え, その NextTop↑を返す.
  図 10 にその様子を示す.

- accept(Top , Info↑)
  Top : 文が受理された時のスタックトップの節点番号
  Info↑ : 最終的に得られた構文木の情報
  文が受理された時点で生成されている構文木の情報を返す.

- create
  このメッセージはそのまま辞書プロセスに伝えられる.

- terminate
  このメッセージはそのまま辞書プロセスに伝えられる.

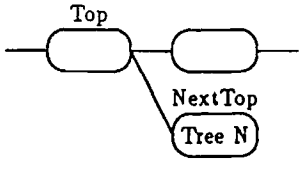



図 10:goto の処理

### 4.3 パージングプロセスの実現

第 3節に示した GHC によるパージングプロセスの記述を以下のように変更する.

#### 4.3.1 action プロセスの実現

シフト動作の処理

```
i0(noun , Top , Stream , Info) :- true |
        Stream=[shift(1,Top,NextTop,NextCat) | Rst],
        i1(NextCat , NextTop , Rst , Info).
```

ここで, action プロセスはストリーム通信によって辞書プロセスから先読み語 NextCat を受け取る. これを受け取る前に次の action プロセス i1 が呼び出されると, 述語 i1 の定義とのユニフィケーションは未束縛変数 NextCat を具体化しようとするので, GHC の原則に従ってそのプロセスは中断される. ここで, 複数のパージングプロセスが存在する場合, 辞書プロセスは全てのパージングプロセスが shift 要求を出すまで辞書引きを待ち合わせる. これにより, 先に shift 要求を出したパージングプロセスは中断される.
また, グラフスタックプロセスが NextCat に[]を返し, action プロセスに停止を要求すると, 術語 i1([], NextTop , Rst , Info) の呼び出しが, 下に示すエラー処理の記述とマッチしてプロセスは停止する.
還元動作の処理

```
i2(v , Top , Stream , Info) :- true |
        Stream=[pop(Top , 2 , Poplist) | Rst],
        reduce(6 , v , Poplist , Rst , Info).
```

ここでは, スタックトップとその一つ前の節点の情報が Poplist に返され, これが reduce プロセスに渡される.
コンフリクトの処理

```
i14(p , Top , Stream , Info) :- true |
        Stream=[create , pop(Top , 3 , Poplist),
              shift(7 , Top , Ntop , Ncat) | Rst],
        reduce(10 , p , Poplist , St1 , Inf1),
        i7(Ncat , Ntop , St2 , Inf2),
        merge(St1 , St2 , Rst),
        merge(Inf1 , Inf2 , Info).
```

ここでは, ストリームに create を送信することにより, プロセスが分岐したことを辞書プロセスに知らせる.
文の受理

```
i4($, Top , Stream , Info) :- true|
        Stream=[accept(Top , Info),
              terminate].
```

文が受理されるとプロセスはストリームに terminate を報告して終了する.
エラーの処理

```
otherwise.
i0(_ , _ Stream , Info) :- true |
        Stream=[terminate], Info=[].
```

これが呼び出されると, ストリームに terminate を報告してプロセスは終了する.

#### 4.3.2 reduce プロセスの実現

reduce プロセスは, 還元動作の際グラフスタックを必要な数だけ後向きにたどった時, スタックの結合点で複数のパスに枝別れすることがある. この場合, 複数の還元動作が要求されるため, reduce プロセスの定義を以下のように変更する.

```
reduce(N , NextCat , [List], Stream , Info) :- true |
        re(N , NextCat , List , Stream , Info).
reduce(N , NextCat , [List | Rest], Stream , Info) :- true |
        Stream=[create | Rst],
        re(N , NextCat , List , St1 , Inf1),
        reduce(N , NextCat , Rest , St2 , Inf2),
        merge(St1 , St2 , Rst),
        merge(Inf1 , Inf2 , Info).
```

還元動作が複数要求される場合は reduce の下の定義が呼び出され, ストリームに create を通信してプロセスが分岐することを知らせる.

ここで，述語 re は第 3節と同様文法規則を GHC 節に変換したものであるが，その第一引数が規則の番号を示し，以下のように変更される．例えば，(1) S→NP，VP． は，

```
re(1, NextCat, [A1, A2, A3], Stream, Info) :- true|
        A1=[_, _, T1],
        A2=[_, _, T2],
        A3=[Top, State, _],
        s(State, Top, NextCat, [s,T2,T1], Stream, Info).
```

述語 re の第三引数はグラフスタックプロセスに pop を要求した時に返される Poplist の一つの要素であり，スタックトップから後向きに必要な数だけたどる一つのパス上の節点の情報を持っている．変数 A1，A2，A3 にはポップされたスタックの節点の持つ情報が束縛されている．A1 には文法記号 VP を含む節点の情報が与えられ，A2 には文法記号 NP を含む節点の情報が与えられている．A3 にはそのさらに一つ前の節点の情報が与えられる．還元動作はこの A1，A2 をポップしたことに相当し，A3 が現在のスタックトップの節点の情報を持つことになる．ここで Top はその節点の番号であり，State は節点の状態番号である．

#### 4.3.3 goto プロセスの実現

goto プロセスも以下のように変更される．図 2 で非終端記号名が S，状態番号が 0 の時エントリに相当する goto プロセスの記述は，

```
s(0, Top, NextCat, Tree, Stream, Info) :- true |
        Stream=[goto(5, Tree, Top, NextTop) | Rst],
        i4(NextCat, NextTop, Rst, Info).
```

と記述される．これによりグラフスタックプロセスに goto を通信し，還元動作によって得られた木の情報 Tree をスタックに先頭に積むことを要求する．

パージングプロセスは以上のように変更される．

## 5 辞書プロセスの実現

辞書プロセスはグラフスタックプロセスからの以下の 3 つのメッセージによって駆動される．

- create パージングプロセスの生成の報告
- terminate パージングプロセスの停止の報告
- request 辞書引きの要求

辞書プロセスは現在起動されているパージングプロセスの数を認識し，全てのプロセスがシフト動作の際の辞書引きの要求を行なうまで，辞書引きを待ち合わせる．これによって，パージングプロセスの同期がとられ，各入力語のシフト動作が全てのプロセスで同時に行なわれる．

この機構はグラフスタックプロセスのアルゴリズムをできるだけ簡略化するために導入された．並列性の向上を考えればプロセスを同時にシフトするよりも，スタックが伸びている間はプロセスを先に進めた方がよい．しかしこうすると，どこまで還元した場合にプロセスの同期をとる必要が生ずるかということを認識するアルゴリズムと，スタックを統合する時，統合するポイントをサーチするアルゴリズムにコストがかかる．そのため，現在のインプリメントではこれを行なわず，単純にシフト動作の際にプロセスの同期をとっていが，この点について検討の余地が残されている．

## 6 DCGの扱いと問題点

本研究で扱う DCG 規則は次のように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述語呼び出しを加えることができる．例えば，図 1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),
        VP(VP_A, VP_S),
        { 述語呼びだし }.
```

のようになる．まず，引数の扱いであるがグラフスタックの各節点 N : (T，I，L)の木の情報 T を，

T' = [第一引数，第二引数，T]

というリストに置き換えることにより，スタックの各節点に引数の情報を与えることができる．補強項を評価するタイミングとしては，文の解析を終えたあとで行なう方法も考えられるが，構文解析の曖昧さの組合せ的な爆発を抑制するために構文解析と意味解析とは融合することが好ましい[Mellish 85][奥村 89]．そこで，本研究ではスタックの還元動作の際に補強項を評価することにする．上のような DCG 規則の場合これを次のように GHC 節に変換する．

```
re(1, NextCat, [A1, A2, A3], Stream, Info) :- true |
        A1=[_, _, [VP_A, VP_S, VP_T]],
        A2=[_, _, [NP_A, NP_S, NP_T]],
        A3=[Top, State, _],
        (補強項の述語呼びだし),
        s(State, Top, NextCat,
          [S_A,S_S,[s,NP_T,VP_T]], Stream, Info).
```

これにより，節点が保持している文法規則の右辺の引数 NP_A，NP_S，VP_A，VP_S の値を取り出して補強項を実行し，左辺の引数 S_A，S_S の値を計算する．そして，goto プロセスにその値を渡し，新しい節点にその値を保持させる．

ここで問題となるのは，補強項の述語呼び出しが失敗する場合と，複数の解を持つ場合である．GHC では，body 部で呼び出される述語に失敗が許されないので，呼び出される述語は必ず成功するように記述しなければならない．そこで，得られる全ての解をリストにして返すように，補強項で呼び出される述語を記述する．そして，goto プロセスを呼び出す述語 goto を作り，reduce の定義と同様に再帰的に解の数だけ繰り返し goto を呼び出すようにする．解がない場合には goto プロセスを停止させる．これによって補強項の述語呼び出しが失敗する場合と，複数の解を持つ場合に対処できる．

また，構文解析システム SAX[松本 86]でも指摘されているように，本方式にも環境のコピーの問題がある．これは，解析する文に曖昧さがある場合に生ずる．例えば，図 8 の後，解析が NP まで進み，図 11 のようになったとする．

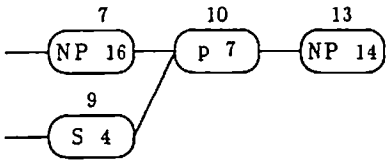



図 11:曖昧さのある文の解析

この後，pと NP を図 1の(10)の規則を用いて PP に還元すると，図 12 のようになり，

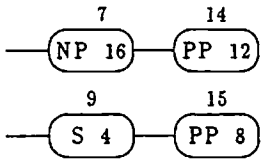



図 12:スタックの分離

スタックは二つに分離するが，その際，二つの PP に付随する引数の情報は，同じ環境を運んでいる．このとき引数に未束縛の変数が含まれていると，次に規則(2)と(4)を用いて PP が別々に還元される時，どちらか一方で変数の束縛が行なわれると，それがもう一方に影響を及ぼす．これにより解析が正しく行なわれないことになる．本来，二つの還元動作は別々の環境で行うべきものなのである．これを解決するには，スタックが運ぶデータに変数を許さないようにするか，あるいはスタックが分離する際に変数のコピーを作り，それぞれの環境に渡すようにする必要がある．

## 7 他の自然言語処理方式との比較

ここでは第 3節で実現した並列 LR 構文解析法のアルゴリズム，及び第 4節で実現したグラフ構造化スタックを組み込んだアルゴリズムと，他の自然言語処理方式との実行効率の比較を行なう．他の方式としては，松本の PAX[Matsumoto 88]を用いる．PAX は本方式と同様 GHC で記述される．GHC の処理系は ICOT で開発された PDSS1.6 版を使用した．
比較の方法は，図 1の文法規則を用い，入力文を

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

のようにして，PP の数を単純に増やした時の処理時間の推移を比較した．図 13 にその結果を示す．また，図 14 に付加した PP の数に対して生成される解析木の数のグラフを示した．

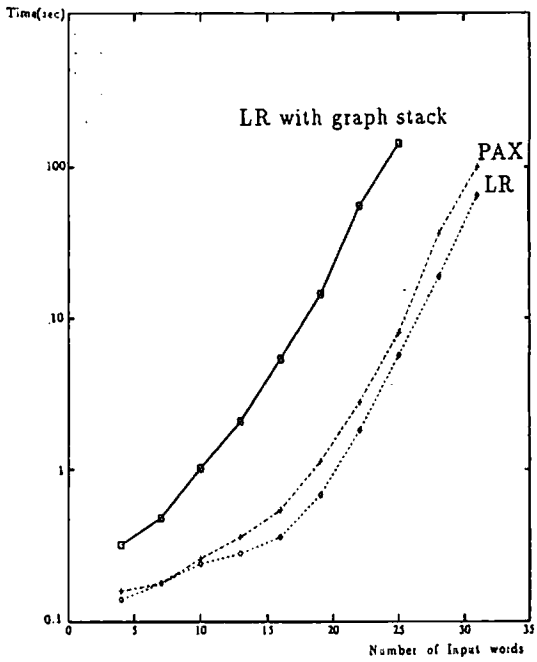



図 13:解析時間の推移

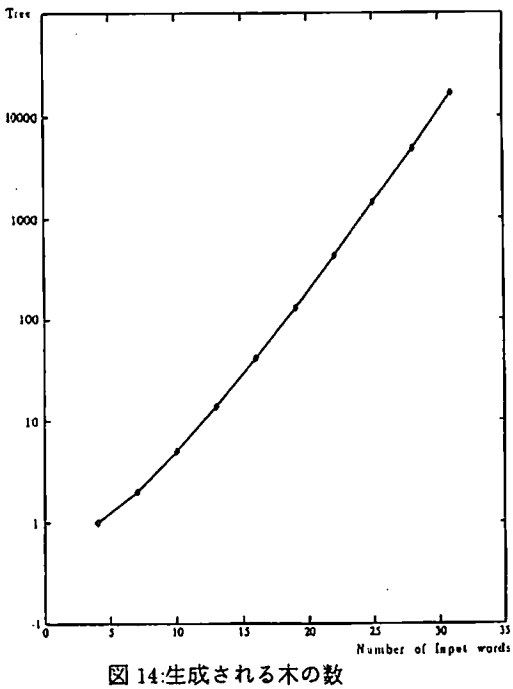



図 14:生成される木の数

ここでは，PP の数が 0から 9までの場合を調べている．グラフの y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．グラフ構造化スタックを用いない方式は解析時間が PAX と同形の推移をたどっており，実行速度は平均して PAX の 1.4 倍速かった．ただし，PDSS はガード部にユーザ定義の述語を書くことができないため，実験で比較した PAX のインプリメントには，トップダウンフィルターの記述に効率を落す要因があったことを指摘しておく．これに対し，グラフ構造化スタックを組み込んだ方式では，組み込まない方式及び PAX と比較して，一桁以上遅いことが分かった．これはアルゴリズムの記述量が大きいことと，グラフ

スタックの節点の保持に副作用を用いていることが原因であると思われる.

## 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扱う並列 LR 構文解析アルゴリズムを提案し，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，各プロセスがスタックを別々に保持するインプリメントの有用性が確認された.

本来効率的とされるグラフ構造化スタックを用いた LR 構文解析アルゴリズムが，インプリメントの方法によっては効率の低下をまねくことが示された．並列論理型言語による効率的なグラフ構造化スタックの実現法は今後の課題とさせていただく.

さらに今後の課題として，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，および，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 提供して下さった ICOT および GHC について助言をいただいた ICOT の近山 隆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

[上原 83] 上原邦昭, 豊田順一: 先読みと予測機能を持つ述語論理型構文解析プログラム:PAMPS, 情報処理学会論文誌, Vol.24, No.4, pp.496-504 (1983)

[奥村 89] 奥村 学:自然言語解析における意味的曖昧性を増進的に解消する計算モデル, 情報処学会自然言語処理研究会研究報告, 71-1 (1989)

[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)

[徳永 88] 徳永健伸, 岩山 真, 上脇 正, 田中穂積: 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)

[中田 81] 中田育男:コンパイラ, 産業図書 (1981)

[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)

[安留 88] 安留誠晋, 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)

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

[Mellish 85] Mellish,C.S.:*Computer Interpretation of Natural Language Descriptions,* Ellis Horwood Limited (1985)

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

[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)

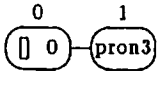
[Ueda 85] Ueda,K:*Guarded Horn Clauses,* Proc. The Logic Programming Conference, Lecture Notes in Computer Science, 221 (1985)

付録:グラフ構造化スタックを用いた構文解析過程

入力文
i open the door with a key .
入力文の文法カテゴリ
pron v det noun p det noun $

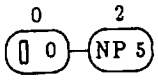
以下，図 1の文法と図 2 のテーブルに従って解析を進める．action テーブルの参照を a(State,Category)=sh N,or re N，goto テーブルの参照を g(State,Category)=N と示すことにする.


a(0,pron)=sh 3

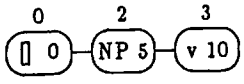



a(3,v)=re 7

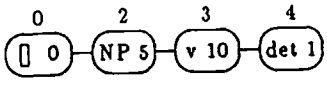

g(0,NP)=5



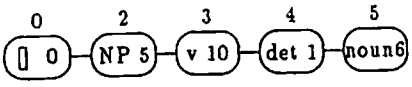

a(5,v)=sh 10



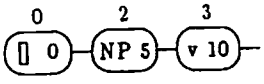

a(10,det)=sh 1



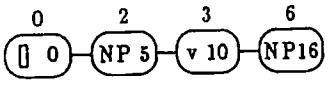

a(1,noun)=sh 6




a(6,p)=re 5



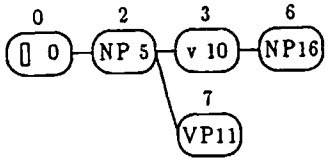

g(10,NP)=16




a(16,p)=sh 7/re 8

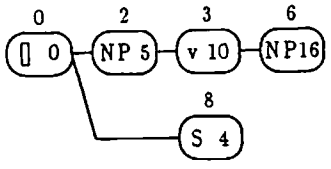
g(5,VP)=11

ここで，文の解析に曖昧さを生ずる．ここでシフト動作は中断され，還元動作が行なわれる．



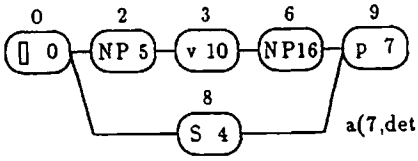

a(11,p)=re 1

g(0,S)=4

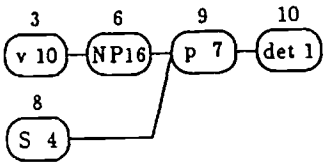



a(4,p)=sh 7

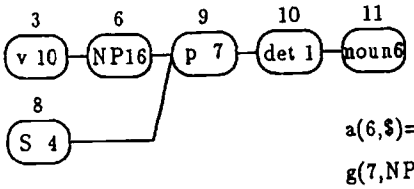
先ほど中断したシフト動作が再開され，二つのシフト動作が同じ状態 7 へ推移するのでスタックを統合する．




a(7,det)=sh 1

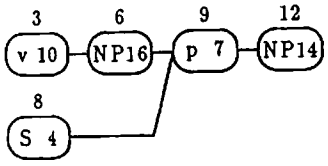



a(1,noun)=sh 6



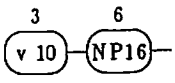

a(6,$)=re 5

g(7,NP)=14




a(14,$)=re 10

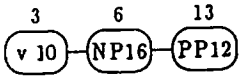
ここで，スタックは再び二つに分離し以後並列に動作する．

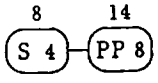




g(16,PP)=12

g(4,PP)=8



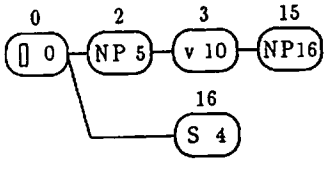




a(12,$)=re 4

g(10,NP)=16

a(8,$)=re 2

g(0,S)=4

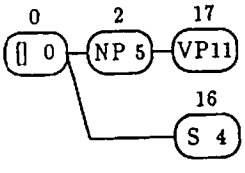



a(16,$)=re 8

g(5,VP)=11

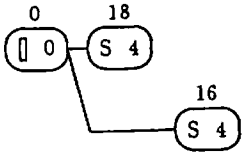
a(4,$)=acc

ここで一方のプロセスが終了する．




a(11,$)=re 1

g(0,S)=4




a(4,$)=acc

もう一方のプロセスも終了して文の解析を終る．結局二つの解析結果が得られたことになる．