一般の文脈自由文法に対する効率的な並列構文解析

峯，恒憲
九州大学システム情報科学研究院知能システム学部門

谷口，倫一郎
九州大学システム情報科学研究院知能システム学部門

雨宮，真人
九州大学システム情報科学研究院知能システム学部門

出版情報：情報処理学会論文誌. 32 (10), pp.1225-1237, 1991-01-15. 情報処理学会
バージョン：

Vol. 32 No. 10 情報処理学会論文誌 Oct. 1991

一般の文脈自由文法に対する効率的な並列構文解析†

峯 恒憲†† 谷口倫一郎†† 雨宮真人††

自然言語を解析するためには，構文情報や意味情報，文脈情報など各種の膨大な量の知識情報を用いて処理することが必要である．しかし，これらの多量な知識情報を計算機で処理するためには，莫大な解析時間が必要となるため，並列処理などの手法を導入して，解析時間の短縮を計る必要がある．一般に文脈自由言語の構文解析には長さが n の入力に対して $O(n^3)$ の手数を必要とする．したがって実時間自然言語理解システムを実現するためには，まず文脈自由言語を $O(n)$ 時間で解析できる並列構文解析アルゴリズムを開発する必要がある．その時，使用するプロセッサ数は実現の可能性上 $O(n^2)$ 以下であることが望ましい．これまでに並列構文解析のアルゴリズムがいくつか提案されてきたが，これらのアルゴリズムには，解析時間やプロセッサ数が多くなりすぎるという問題がある．本論文では，一般の文脈自由文法を対象とする並列構文解析アルゴリズムを提案する．このアルゴリズムの性能は解析時間 $O(n)$・プロセッサ数 $O(n^2)$ でその積が $O(n^3)$ である．本アルゴリズムでは，一般の文脈自由文法から LR 法に類似した手法を用いて作成した LR 状態遷移図を解析制御表として使用し，すべての可能性を並列に試しながら入力に非同期に解析を行う．

1. はじめに

自然言語を解析するためには，構文情報や意味情報，文脈情報など各種の膨大な量の知識情報を用いて処理することが必要である．しかし，これらの多量な知識情報を計算機で処理するには，膨大な解析時間が必要となるため，並列処理の手法を導入して解析時間の短縮を計る必要がある．

一般に文脈自由言語の逐次型構文解析には，長さが n の入力に対してほぼ $O(n^3)$ の手数を必要とする*．したがって，実時間自然言語理解システムを実現するためには**，まず文脈自由言語を $O(n)$ 時間で解析する並列構文解析アルゴリズムを開発することが必要である．その時，使用するプロセッサの数は実現可能性の点から見て，$O(n^2)$ 以下であることが望ましい．

これまでにいくつかの並列構文解析アルゴリズムが提案されてきた[1)~5)]．しかし，これらのアルゴリズムには，解析時間が $O(n^2)$ 以上必要となる[2)]か，あるいは解析時間は $O(\log n)$ と優れているがプロセッサ数が $O(n^6)$ と多くなりすぎる[3)]，というように解析時間かプロセッサ数かのどちらかが多くなり過ぎるという問題がある．

本稿では，一般の文脈自由文法を対象として，解析時間 $O(n)$・プロセッサ数 $O(n^2)$ という性能をもつ並列構文解析アルゴリズムを提案する．このアルゴリズムでは，一般の文脈自由文法から構成した LR 状態遷移図[8),9)]を解析制御表として使用し，すべての可能な解析パスを並列に試しながら入力に対して非同期な解析を行う．

以下，2 章では，われわれが提案する LR 状態遷移図について，その特徴と構成アルゴリズムを示す．3 章では，まず，この LR 状態遷移図を使用して 1 個のプロセッサが 1 個の構文木を作成するアルゴリズム（基本アルゴリズムと呼ぶ）を述べ，基本アルゴリズムが長さ n の入力を $O(n)$ 時間で解析できることを示す．次に，基本アルゴリズムを改良して，プロセッサ数を $O(n^2)$ に抑えるアルゴリズム（最適化アルゴリズムと呼ぶ）を示す．4 章では，最適化アルゴリズムに対する理論的な性能評価を行い，5 章で，他の手法と比較する．

2. LR 状態遷移図

2.1 shift-reduce テーブルと LR 状態遷移図

LR 構文解析法[7),10)]は，LR shift-reduce パージングテーブル（以下，shift-reduce テーブルと略記）と呼ばれる解析制御表を使用して，shift, reduce, goto の 3 動作により，入力記号列を左から右に見ながら最右導出の逆順に解析を行う PDA (Push Down Automaton) である．以後，LR 構文解析法や shift-reduce テーブルの作成法についての知識を仮定して述べていく．これらについては，文献 7) などを参照されたい．

shift-reduce テーブルは，文法規則を使った還元

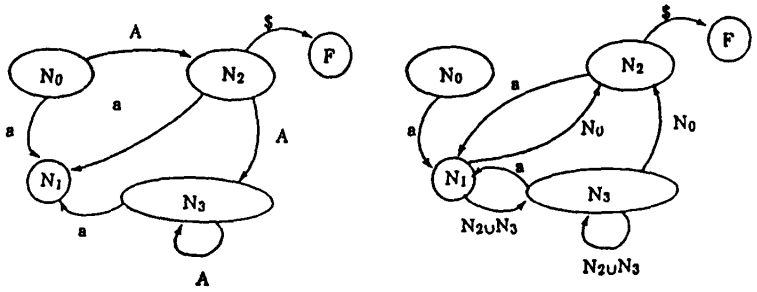
† An Efficient Parallel Parsing Algorithm for General Context-free Grammars by TSUNENORI MINE, RIN-ICHIRO TANIGUCHI and MAKOTO AMAMIYA (Department of Information Systems, Graduate School of Engineering Sciences, Kyushu University).

†† 九州大学総合理工学研究科情報システム学専攻

* Valiant のアルゴリズムは $O(n^{2.81})$ の手数ですむ[6)]が，複雑過ぎて実現するのに問題がある[7)]．

** ここでいう実時間とは，応答時間が $O(n)$ 以下の時間を指す．

(reduce) 動作と reduce 動作後の状態の遷移 (goto) 動作とを別動作に分けているため，reduce 動作を行う状態から次に遷移する状態が明示的ではなく，アルゴリズムの reduce 動作の記述が複雑になる．このためわれわれは，reduce 動作と goto 動作を1動作として実行できるように shift-reduce テーブルを改良した LR 状態遷移図を提案している[8),9)]．この LR 状態遷移図では，reduce 動作の遷移情報として，非終端記号の代わりに状態番号を使用し，reduce 動作後の遷移先を直接アークで指示するようにしている．この改良により，解析アルゴリズムの reduce 動作の記述が簡明となり，記述量も減る．しかも，オートマトンとしても自然な形をもつ．



(a) 一般の LR 状態遷移図
(a) Conventional LR transition diagram.

(b) われわれの LR 状態遷移図
(b) Our LR transition diagram.

図 1 LR 状態遷移図の比較
Fig. 1 Comparison between conventional and proposed LR transition diagrams.

図1に，われわれの LR 状態遷移図と shift-reduce テーブルから直接作成した一般の LR 状態遷移図とを示す．一般の LR 状態遷移図では，reduce 動作の遷移先が非終端記号によって明示されているため，例えば，状態 N_0 から shift 動作により遷移した状態 N_1 で，文法規則 $A \rightarrow a$ を使う reduce 動作を行う場合，shift-reduce テーブルでは，状態 N_1 から状態 N_0 への戻り (reduce 動作)* と，A で示された状態 N_2 への遷移 (goto 動作) とが，別々の動作として行われる．一方，われわれの状態遷移図では，reduce 動作後の遷移先を状態番号 N_i $(i=0..3)$ によって明示しているため，状態 N_1 から N_2 への遷移を直接行うことができる．

以下，単に LR 状態遷移図と呼ぶ場合は，われわれが提案する LR 状態遷移図を指すものとする．

2.2 基本定義

LR 状態遷移図の記述に必要な，用語の定義および記法を示す．

[定義 1] 文脈自由文法：文脈自由文法Gを $G=<V, T, P, S>$ とする．ここで V と T はそれぞれ非終端記号の有限集合，終端記号の有限集合で，$S\,(\in V)$ は開始記号である．また，P は $A \rightarrow \alpha$ という形の文法規則の有限集合で，$A \in V$, $\alpha=(V \cup T)^*$ である．ただし * は0個以上の連接を表す記号とする．

[記法 1] 終端記号を英小文字 $a, b, c\cdots$，非終端記号を英大文字 $A, B, C\cdots$ で表す．ただし X, Y は $X \in (T \cup V)$, $Y \in (T \cup V)$ とする．ギリシャ文字 $\alpha, \beta, \gamma\cdots$ は $(V \cup T)^*$, $|\alpha|$ は α の長さを表す．また，空記号列を ε, 空を *nil* と記す．長さ n の入力記号列 W を $W=$"$w_1 w_2 \cdots w_{n-1} w_n$" と表す．$\alpha \Rightarrow \beta$ は，0回以上の文法規則の適用によって，α から β が最右導出されることを表す．

[定義 2] マーカ付規則：文法規則 $A \rightarrow \alpha\beta$ の右辺にマーカ "・" を加えたものをマーカ付規則と呼び，$[A \rightarrow \alpha \cdot \beta]$ と表す．

[定義 3] クロージャ：I を文法 $G=<V, T, P, S>$ に対するマーカ付規則の集合とするとき，I のクロージャ *closure*(I) は，以下の規則によって I から生成したマーカ付規則の集合である[7)]．

1. Iに含まれるマーカ付規則は，すべて *closure*(I) に含まれる．
2. $[A \rightarrow \alpha \cdot B\beta]$ が *closure*(I) に含まれ，$B \rightarrow \gamma$ という文法規則があるとき，マーカ付規則 $[B \rightarrow \cdot \gamma]$ が *closure*(I) に含まれていなければ，*closure*(I) に加える．

[定義 4] LR 状態遷移図：文法Gの LR 状態遷移図とは，次の節で述べるアルゴリズムによって作成した文法Gのマーカ付規則の集合族からなるノードの集合と，shift/reduce 動作の指示およびその動作後の遷移先状態を指示するアークをもつグラフである．

上で定義した LR 状態遷移図の各ノードを，状態ノード (あるいは単に状態) と呼ぶ．この状態ノードはクロージャと一対一に対応する．また，アーク上に記された shift/reduce 動作の指示を遷移情報と呼ぶ．

[記法 2] クロージャ C_i と対応する状態ノードに番号 N_i を付け，$N_i = label(C_i)$ あるいは $C_i = label^{-1}$

* スタックの pop up によって，状態番号 N_0 を格納するリストを得る．

(N_i) と表す.

[記法 3] 状態 N_i から状態 N_j への遷移情報の集合を $Trans(N_i, N_j)$ と表す. shift 動作の遷移情報として終端記号を用い, reduce 動作の遷移情報として状態番号を用いる.

2.3 LR 状態遷移図構成アルゴリズム

LR 状態遷移図構成アルゴリズムは, 状態遷移図のノードの構成アルゴリズムと, アーク上に記された遷移情報の構成アルゴリズムからなる. アルゴリズムの記述には Pascal 風の構文を用いる.

与えられた文法 $G=<V, T, P, S>$ に規則 $S'\rightarrow S\$$ を加え, マーカ付規則 $[S'\rightarrow\cdot S\$]$ に対するクロージャを初期状態ノードとして, **図 2** のアルゴリズムに従って状態ノードを作成する. $label(closure(\{[S'\rightarrow S\$\cdot]\}))$ を最終状態 *final* とする. 図 2 の状態ノード構成アルゴリズムは, 文献 7) でいう LR(0) 項の構成アルゴリズムと等しい. また, **図 3** の遷移情報構成アルゴリズムでは, 8〜12 行目が shift 動作の遷移情報とその情報による遷移先状態を決定し, 13〜21 行目が reduce 動作の遷移情報とその遷移先状態を決定している. reduce 動作で使用される遷移情報や遷移先状態の決定は, まず N_i に含まれる $[A\rightarrow\alpha\cdot X\beta]$ という形のマーカ付規則の集合から, 各規則のマーカを一記号分右に移動させたマーカ付規則の集合 $Rwm[X]$ を求め, ついで $label(closure(Rwm[X]))$ なる状態 N_j と, $[X\rightarrow\gamma\cdot]\in N_k$ なる状態 N_k から, $N_i\in Trans(N_k, N_j)$ という関係を求めることによって行っている.

2.3.1 LR 状態遷移図の構成例

文法 G1$\{A'\rightarrow A\$, A\rightarrow AA|a\}$ を例にとり説明する.

1. 状態ノードの構成
   図 2 の状態ノード構成アルゴリズムから, 次のような 5 つの状態ノードが構成される.
   C_0: $\{[A'\rightarrow\cdot A\$], [A\rightarrow\cdot AA], [A\rightarrow\cdot a]\}$
   C_1: $\{[A\rightarrow a\cdot]\}$
   C_2: $\{[A'\rightarrow A\cdot\$], [A\rightarrow A\cdot A], [A\rightarrow\cdot AA], [A\rightarrow\cdot a]\}$
   C_3: $\{[A\rightarrow AA\cdot], [A\rightarrow A\cdot A], [A\rightarrow\cdot AA], [A\rightarrow\cdot a]\}$
   C_4: $\{[A'\rightarrow A\$\cdot]\}$
   $N_i=label(C_i)$ $(0\leq i\leq 4)$, $(final=N_4)$
2. 遷移情報の構成
   図 3 の遷移情報構成アルゴリズムから, 遷移情報はつぎのように構成される.
   ● $i=0$ で, 5 行目の手続き state_maker より, $Rwm[A]=\{[A'\rightarrow A\cdot\$], [A\rightarrow A\cdot A]\}$, $Rwm[a]=\{[A\rightarrow a\cdot]\}$, $Sym=\{a, A\}$ を得る. 10, 11 行目から, $label(closure(Rwm[a]))=N_1$ より $N_j=N_1$ となり, $Trans(N_0, N_1)=\{a\}$ を得る. 15〜20 行目から, $label(closure(Rwm[A]))=N_2$ で, $\{[A\rightarrow a\cdot]\}\in label^{-1}(N_1)$ よ

```
1: procedure LR_transition_state_node;
2: begin
3:    C0 :=closure({[S' → ·S$]});
4:    C := {C0}..a;N := {}..b;
5:    j := 1; i := 0 ;
6:    for each X ∈ S do ..c
7:       Rwm[X] := {}
8:    repeat
9:       Sym := {};
10:      state_maker(Ci, Sym, Rwm);
11:      make_node(Ci, Sym, Rwm, j);
12:      i:=i+1;
13:   until  i ≥ j
14:   for each Ci ∈ C do
15:      begin
16:         Ni := label(Ci);
17:         N := N ∪ {Ni}
18:      end
19:end
```

- a C はクロージャの集合.
- b N は状態ノードの集合.
- c S は文法記号の集合.

```
1: procedure state_maker(C,var Sym,var Rwm);
2: begin
3:    for each rwm ∈ C do..a
4:       begin
5:          X := rsym_of_marker(rwm).. b;
6:          if X ≠ ε then
7:             Rwm[X] := Rwm[X] ∪ {rshift_marker(rwm)} ..c;
8:             if X ∉ Sym then
9:                Sym := Sym ∪ {X};
10:      end
11:end
```

- a *rwm* はマーカ付規則.
- b *rsym_of_marker(rwm)* はマーカ付規則 *rwm* のマーカのすぐ右隣の記号を返す関数.
  例) $rsym\_of\_marker([A\rightarrow\alpha\cdot X\beta])=X$
- c *rshift_marker(rwm)* は *rwm* のマーカを 1 記号分右に移動させたマーカ付規則を返す関数.
  例) $rshift\_marker([A\rightarrow\alpha\cdot X\beta])=[A\rightarrow\alpha X\cdot\beta]$

```
1: procedure make_node(C,Sym,var Rwm,var j);
2:    for each X ∈ Sym do
3:       begin
4:          Cj := closure(Rwm[X]);
5:          if Cj ∉ C then
6:             begin
7:                C := C ∪ {Cj};
8:                j := j + 1
9:             end
10:         Rwm[X] := {}
11:      end
12:end
```

図 2 状態ノード作成アルゴリズム
Fig. 2 Construction of closures.

```
1: procedure State_Transition_Information;
2: begin
3:   for each N_i ∈ N do
4:     begin
5:       state_maker(label^-1(N_i), Sym, Rwm);
6:       for each X ∈ Sym do
7:         begin
8:           if X ∈ T then
9:             begin
10:              N_j := label(closure(Rwm[X]));
11:              Trans(N_i, N_j) := {X}
12:            end
13:          else /* X ∈ V */
14:            begin
15:              N_j := label(closure(Rwm[X]));
16:              for each rule ∈ reduceRWM[X] do ..a
17:                for each N_k ∈ N do
18:                  if rule ∈ label^-1(N_k) then
19                     if N_i ∉ Trans(N_k, N_j) then
20:                      Trans(N_k, N_j):= Trans(N_k, N_j) ∪ {N_i}
21:            end
22:        end
23:    end
24:end
```

[a] *reduce RWM*[X] は，左辺記号が X で，マーカの右の記号が ε のマーカ付規則 [$X \rightarrow \alpha\cdot$] の集合．

図 3 遷移情報作成アルゴリズム
Fig. 3 Construction of state transition information.

り，$Trans(N_1, N_2) = \{N_0\}$ を得る．また，$[A \rightarrow AA\cdot] \in label^{-1}(N_3)$ より，$Trans(N_3, N_2) = \{N_0\}$ を得る．

- $i=1$ で，state_maker より $Sym = \{\}$ となるので何も行わない．
- $i=2$ で，state_maker より $Sym = \{\$, A, a\}$，$Rwm[\$] = \{[A' \rightarrow A\$\cdot]\}$，$Rwm[a] = \{[A \rightarrow a\cdot]\}$，$Rwm[A] = \{[A \rightarrow AA\cdot], [A \rightarrow A\cdot A]\}$ を得る．$i=0$ の時と同様に，$label(closure(Rwm[a])) = N_1$ より $Trans(N_2, N_1) = \{a\}$，$label(closure(Rwm[\$])) = final$ から $Trans(N_2, final) = \{\$\}$ を得る．また，$label(closure(Rwm[A])) = N_3$ で，$[A \rightarrow a\cdot] \in label^{-1}(N_1)$ かつ $[A \rightarrow AA\cdot] \in label^{-1}(N_3)$ なので，$Trans(N_1, N_3) = \{N_2\}$，$Trans(N_3, N_3) = \{N_2\}$ を得る．
- $i=3$ でも同様にして，$Trans(N_3, N_1) = \{a\}$，$Trans(N_1, N_3) = \{N_2, N_3\}$，$Trans(N_3, N_3) = \{N_2, N_3\}$ を得る．

以上から，図1（b）に示された LR 状態遷移図を得る．

3. 並列 LR 構文解析アルゴリズム

3.1 基本アルゴリズム

3.1.1 基本アルゴリズムの概要

本章では，まず LR 構文解析法[7),10)]を並列化し，一般の文脈自由文法を扱うことができるようにしたアルゴリズムを示す．これを基本アルゴリズムと呼ぶ．基本アルゴリズムが想定するマシンは，多数のプロセッサを持つ MIMD タイプの P-RAM (Parallel-Random Access Machine)[11)] である．

基本アルゴリズムでは，プロセッサは，解析を行うための情報 inf：$<N, i, p>$（N は2章で述べた LR 状態遷移図の状態番号，i は入力記号へのポインタで，p はスタックへのポインタ）を持ち，状態 N で指定される実行可能な shift/reduce 動作のすべてを並列に実行しながら解析を行う．この時，実行できた動作の数だけ新たにプロセッサを起動して，それらに解析状態を保持するスタック内容を継承させる．スタック内容を一定時間内で継承させるために，複数のプロセッサがもつスタックエントリを DAG (Directed Acyclic Graph) で表現し，スタック内容の共通部分を共有化する．このとき，プロセッサがもつスタックへのポインタ p は，DAG ノードへのポインタを表す．プロセッサは，新たに起動したすべてのプロセッサに inf を渡して解析を継続させた後，解放される．起動された各プロセッサは，それぞれ並列に動作しながら，各々異なる1個の構文木を作成する．

3.1.2 基本アルゴリズム

アルゴリズムを示すために，まず，DAG ノードの定義を行う．

[定義 5] DAG ノード：DAG ノードを $<N, X, l, r, p>$ とする．ここで N は状態番号，X は $(V \cup T)$ の要素，l, r はそれぞれ X から導出される入力記号列の左端位置，右端位置で $X \Rightarrow^{*} w_{l+1} \cdots w_r$ であり，p は他の DAG ノードへのポインタである．

[記法 4] DAG ノード dn の各成分の値を，$dn.N$, $dn.X$, $dn.l$, $dn.r$, $dn.p$ と表す．また，dn へのポインタを $\uparrow dn$ と表し，ポインタ p が指す DAG ノードの内容を $\downarrow p$ と表す．すなわち $p = \uparrow dn$ のとき $\downarrow p = dn$ である．

次に，inf：$<N_i, r, \uparrow dn>$（dn：$<N, X, l, r, p>$）をもつプロセッサ P が行う shift/reduce 動作を定義する．

[定義 6] shift/reduce 動作：

[shift 動作：] もし，$w_{r+1} \in Trans(N_i, N_j)$ なる状態 N_j が存在するならば，P は，DAG ノード dn'：$<N_i, w_{r+1}, r, r+1, \uparrow dn>$ の生成と，新たなプロセッサ P' の起動を行い，P' に inf'：$<N_j, r+1, \uparrow dn'>$ をもたせる．

[reduce 動作:] 状態 N_i で reduce 動作に使用する文法規則を $A \rightarrow X_1X_2\cdots X_m$ とする．プロセッサ P は DAG ノードのポインタを $m-1$ 回たどり，そのとき得たエントリが DAG ノード $<N_k, X_1, l', r', p'>$ を指すとき，もし $N_k \in Trans(N_i, N_j)$ なる状態 N_j が存在するならば，P は DAG ノード dn' : $<N_k, A, l', r, p'>$ の生成と，新たなプロセッサ P' の起動を行い，P' に inf' : $<N_j, r, \uparrow dn'>$ をもたせる．

以上から，基本アルゴリズムを以下に示す．

1. 開始：プロセッサ P_0 が inf : $<N_0, 0, \uparrow dn_0>$ (dn_0 : $<nil, nil, 0, 0, nil>$) をもつ．
2. shift/reduce 動作：inf : $<N_i, r, p>$ をもつプロセッサ P は，状態 N_i で実行可能となるすべての shift/reduce 動作を並列に実行した後，解放される．
3. 終了：shift 動作によって終了記号 '$' を読むプロセッサがある時，入力は受理 (acceptance) され，解析を行うプロセッサが1個もない時，入力は拒絶 (rejection) される．

3.1.3 解析例

ここで，文法 G1 から構成した図1(b)の LR 状態遷移図を使用して，入力記号列 "*aaa*$" を解析する例を図4に示す．図では，shift と reduce の各動作を1単位時間として，各時間ごとの DAG の様子を示している．簡単のため，DAG ノードを指すポインタを矢印によって表し，DAG ノードを $<N, X, l, r>$ とする．DAG ノード dn の右に付された $P_i(N_i)[<=P_k]$ は，P_k によって起動された P_i が，inf : $<N_i, dn.r, \uparrow dn>$ をもつことを表している．

時間	DAG
0	$<nil, nil, 0, 0>: P_0(N_0)$
1	$<N_0, a, 0, 1>: P_1(N_1)[<=P_0]$
2	$<N_0, a, 0, 1>$ $<N_0, A, 0, 1>: P_2(N_2)[<=P_1]$
3	$<N_0, a, 0, 1>$ $<N_0, A, 0, 1> \leftarrow <N_2, a, 1, 2>: P_3(N_1)[<=P_2]$
4	$<N_0, a, 0, 1>$ $<N_0, A, 0, 1> \leftarrow <N_2, a, 1, 2>$ $\nwarrow <N_2, A, 1, 2>: P_4(N_3)[<=P_3]$
5	$<N_0, a, 0, 1>$ $<N_0, A, 0, 1> \leftarrow <N_2, a, 1, 2>$ $\nwarrow <N_2, A, 1, 2> \leftarrow <N_3, a, 2, 3>: P_5(N_1)[<=P_4]$ $<N_0, A, 0, 2>: P_6(N_2)[<=P_4]$
6	$<N_0, a, 0, 1>$ $<N_0, A, 0, 1> \leftarrow <N_2, a, 1, 2>$ $\nwarrow <N_2, A, 1, 2> \leftarrow <N_3, a, 2, 3>$ $\nwarrow <N_3, A, 2, 3>: P_7(N_3)[<=P_5]$ $<N_0, A, 0, 2> \leftarrow <N_2, a, 2, 3>: P_8(N_1)[<=P_6]$
7	$<N_0, a, 0, 1>$ $<N_0, A, 0, 1> \leftarrow <N_2, a, 1, 2>$ $\nwarrow <N_2, A, 1, 2> \leftarrow <N_3, a, 2, 3>$ $\nwarrow <N_3, A, 2, 3>$ $\nwarrow <N_2, A, 1, 3> P_9(N_3)[<=P_7]$ $<N_0, A, 0, 2> \leftarrow <N_2, a, 2, 3>$ $\nwarrow <N_2, A, 2, 3> P_{10}(N_3)[<=P_8]$
8	$<N_0, a, 0, 1>$ $<N_0, A, 0, 1> \leftarrow <N_2, a, 1, 2>$ $\nwarrow <N_2, A, 1, 2> \leftarrow <N_3, a, 2, 3>$ $\nwarrow <N_3, A, 2, 3>$ $\nwarrow <N_2, A, 1, 3>$ $<N_0, A, 0, 2> \leftarrow <N_2, a, 2, 3>$ $\nwarrow <N_2, A, 2, 3>$ $<N_0, A, 0, 3> P_{11}(N_2)[<=P_9]$ $<N_0, A, 0, 3> P_{12}(N_2)[<=P_{10}]$

図4 基本アルゴリズムの解析例
Fig. 4 Parsing of an input string "*a a a* $" by using the basic algorithm.

3.1.4 基本アルゴリズムの性能と評価

基本アルゴリズムは，その性質上，すべての可能な動作を並列に実行する非決定性の PDA と等価である(証明略)ので，一般の文脈自由言語を受理する．この基本アルゴリズムでは，プロセッサは，それぞれ他のプロセッサの解析結果に関与しないため，長さ n の入力記号列を解析する時間は，単純に1個の構文木を作成する時間に等しく，高々 $O(n)$ である[8]．またプロセッサは，ある状態で実行可能な動作がある場合，その動作の数だけ起動されるので，プロセッサ数は曖昧さに比例し[12]，$O(c^n)$ である．

3.2 最適化アルゴリズム

3.2.1 基本アルゴリズムの改良

3.1 節で述べた基本アルゴリズムは，長さ n の入力記号列を $O(n)$ 時間で解析できるが，使用するプロセッサの数が曖昧さに比例[12]して $O(c^n)$ と多くなり過ぎることが問題である．プロセッサ数を削減するために，同じ DAG ノードをもつプロセッサが複数個存在しないように制御し，同一の頂点をもつ部分木を1個に縮約しながら，1個に縮退された構文木を作成する．また，各プロセッサが行った動作結果をテーブルに記録して，基本アルゴリズムで生じる重複動作を省く．このような改良を基本アルゴリズムに加えたアルゴリズムを最適化アルゴリズムと呼ぶ．最適化アルゴリズムを実行するための解析モデルとして，DAG アレイ解析モデルを導入する．DAG アレイ解析モデルは，各プロセッサが非同期に動作するプロセッサ群と，3次元配列構成の同時読み出し可能な共有メモリ(これを DAG メモリと呼ぶ)をもつ P-RAM ベースの解析モデルである．DAG メモリの各セルには，ポインタ成分を除いた4つ組の DAG ノード $<N, X, l,$

$r>$ が対応し*，各セルへは DAG ノード dn をインデックスとしてアクセスできる．そのセルを DAG[dn] と表す．各セルには1個のプロセッサが割り当てられている．DAG アレイ解析モデルの構成を**図5**に示す．

まず同一の DAG ノードをもつプロセッサを複数個存在させないように，DAG メモリを使用して，同じ DAG ノードが既に生成されているか否かのチェックを行う．このため DAG メモリを，初期設定時に各 DAG ノード dn に対して DAG[dn]=*off* としておき，dn に割り当てられているプロセッサを起動するときに DAG[dn]=*on* とする．つぎに重複動作を省くため，reduce 動作を，使用する文法規則の右辺の長さ回に分割し，各1記号ごとの reduce 動作（これを1記号 reduce 動作と呼ぶ）を，その reduce 動作時にトレースされる DAG ノードに割り当てられたプロセッサに行わせる．このとき1記号 reduce 動作の連続性を保持するため，1記号 reduce 動作の結果は，shift 動作で生成した DAG ノードにテーブルを割り付けて，それに記録する．トレースされる DAG ノード $<N, X, i, j>$ ($X \Rightarrow w_{i+1} \cdots w_j$) に割り当てられたプロセッサが行う1記号 reduce 動作が右辺の最左記号に対するものでない場合，つまりその1記号 reduce 動作の後に1記号 reduce 動作が続く場合，その動作結果を $<N, w_{i+1}, i, i+1>$ に割り付けられたテーブルに記録する．shift 動作で $<N, w_{i+1}, i, i+1>$ を生成するすべてのプロセッサは，その動作結果を利用して，つぎの記号に対する1記号 reduce 動作を行う．この1記号 reduce 動作の中で，特に最右の記号に対する1記号 reduce 動作を reduce 1 動作，他の1記号 reduce 動作を reduce 2 動作と呼び，1記号 reduce 動作で生成される動作結果を reduce 情報と呼ぶ．また，reduce 情報を格納するためのテーブルを reduce テーブルと呼ぶ．reduce テーブルは，reduce 情報の登録／未登録をチェックするためのテーブル（参照テーブル）と reduce 情報を登録するためのテーブル（登録テーブル）からなる．reduce テーブルの構成を**表1**に示す．

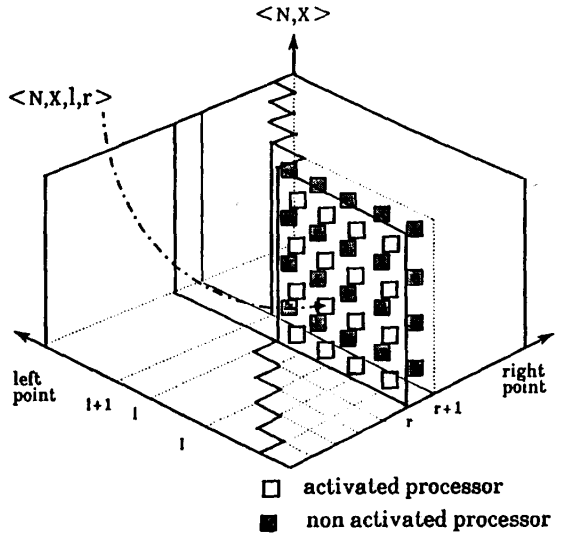


図5 DAG アレイ解析モデルの全体構成
Fig. 5 DAG array parsing model.

* DAG ノードの $<N, X>$ 成分は，状態遷移図中の特定の状態を表し，N と X とは独立ではない．このため，DAG アレイ解析モデルでは，$<N, X>$ を1つの次元として扱う．

DAG アレイ解析モデルでは，各動作によって生成された DAG ノード dn が DAG[dn]=*off* のとき，DAG[dn]=*on* とし，dn に割り当てられているプロセッサ P を起動する．起動された P は，shift/reduce 1 動作を行う．プロセッサは shift 動作で生成した DAG ノード dn' に対して，DAG[dn']=*off* のとき，dn' に reduce テーブル rt を割り付ける．shift 動作終了後，プロセッサは rt に蓄えられる reduce 情報を使って，reduce 2 動作を行う．reduce 2 動作では，DAG ノードの X 成分と，reduce 情報のマーカ

表1 DAG ノード dn に割り当てられた reduce テーブル
Table 1 Reducing table assigned to a DAG node dn.

参照テーブル $Rt_{dn}.F[ri]$

	ri	*boole*
$F[ri]$	$<[A \to \alpha \cdot \beta], i, N_j>$	*on*
	$<[A \to \alpha \cdot \beta], i+1, N_j>$	*off*
	$<[A \to \alpha \cdot \beta], i+2, N_j>$	*off*
	…	*off*
	$<[A \to \alpha \cdot \beta], m, N_j>$	*off*
	…	*off*

登録テーブル Et_{dn}

reduce 情報
$<[A \to \alpha \cdot \beta], i, N_j>$
$<[A \to \alpha \cdot \gamma], i, N_k>$
…
$<[A \to \alpha \cdot \delta], n, N_k>$
nil
…
nil

付規則成分内のマーカの左隣の記号との照合を行い，一致した reduce 情報を選択する．reduce 1/reduce 2 動作では，最左記号に対する1記号 reduce 動作を行うときに DAG ノードを生成し，それ以外の記号に対する1記号 reduce 動作を行うとき，reduce 情報を reduce テーブルに格納する．

reduce 2 動作を行うプロセッサは，登録テーブルに登録されている reduce 情報を上から順に読みとり，もし，読みとるべき reduce 情報がないときは，登録されるのを待って，reduce 2 動作を続ける．このように，登録テーブル内の reduce 情報は，各 reduce テーブルごとに並行して，パイプライン的に処理される．

3.2.2 最適化アルゴリズム

簡単のために，以下では並列認識アルゴリズムを示す．まず，アルゴリズムの説明のために必要な定義，および記法を記す．

[定義 7] reduce 情報：reduce 情報を $<[A\rightarrow\alpha\cdot\beta], i, N>$ と表す．ここで，$[A\rightarrow\alpha\cdot\beta]$ は，$[A\rightarrow\alpha\beta\cdot]$ から β までの還元がすんだことを表すマーカ付規則で，i は reduce 動作を開始したプロセッサがもつ入力記号へのポインタ，N は reduce 動作が開始された状態を表す．

[記法 5] reduce 情報 ri のマーカ付規則，入力記号へのポインタ成分，および状態番号成分を，それぞれ $ri.rwm$, $ri.i$, $ri.N$ と表す．

[定義 8] reduce テーブル：reduce テーブルは参照テーブルと登録テーブルからなり，両テーブルとも同時読み出しを許す．DAG ノード dn に割り当てた参照テーブル，登録テーブルをそれぞれ Rt_{dn}, Et_{dn} と表す．Rt_{dn} は reduce 情報 ri が格納されているかどうかを調べるためのフィールド $F[ri]$ をもつ配列で，Et_{dn} は reduce 情報を格納するリストである．

[定義 9] inf：DAG ノード dn に割り当てられたプロセッサ P がもつ inf を $<N, i, p_{dn}, p1, p2>$ とする．ここで N は状態番号，i は入力記号へのポインタ，p_{dn} は P が割り当てられている DAG ノードへのポインタである．$p1$ は reduce 情報を書き込む先の reduce テーブルが割り当てられている DAG ノードへのポインタで，$p2$ は次に読みとるべき reduce 情報の格納場所を指すポインタである．

[記法 6] P がもつ inf を $P.inf$ と表し，inf の各成分 N, i, p_{dn}, $p1$, $p2$ の値を $inf.N$, $inf.i$, $inf.p_{dn}$, $inf.p1$, $inf.p2$ と表す．

[定義 10] 1記号 reduce 動作：inf：$<\%, \%, \uparrow dn, p1, \%>$* ($dn$：$<N_k, X, l, r>$) をもつプロセッサ P が reduce 情報 ri：$<[A\rightarrow\alpha X\cdot\beta], m, N_i>$ に対して，1記号分マーカを左に移動させた ri'：$<[A\rightarrow\alpha\cdot X\beta], m, N_i>$ を生成するとき，もし $\alpha\neq\varepsilon$ かつ $Rt_{i,p1}.F[ri']=off$ ならば，$Et_{i,p1}$ に ri' を追加する．

もし $\alpha=\varepsilon$ かつ $DAG[dn']=off$ (dn'：$<N_k, A, l, m>$) で，$N_k\in Trans(N_i, N_j)$ なる N_j が存在するならば，dn' に割り当てられたプロセッサ P' を起動し，P' に inf'：$<N_j, m, \uparrow dn', p1, nil>$ をもたせる．

また，並列実行の記述のため，以下の構文を用いる．

```
1:program main; /* メイン プログラム */
2:begin
3: dn0 :=< nil, nil, 0, 0 >;
4: P := allocate_Processor(); /* プロセッサの割り当て */
5: P.inf :=< N0, 0, ↑dn0, nil, nil >;
6: parallel_SR(P)
7:end
```

```
1:procedure parallel_SR(P); /* shift/reduce 動作の並列実行 */
2:parbegin
3:    shift_action(P);
4:    reduce1_action(P)
5:parend
```

図 6 メインプログラム
Fig. 6 Main program.

```
1: procedure shift_action(P);/* shift 動作 */
2: begin
3:    Ni := P.inf.N; ip := P.inf.i; p := P.inf.pdn; p1 := P.inf.p1;
4:    for each Nj ∈ N parallel do .. a
5:       if Trans(Ni, Nj) ∈ STI(Ni) then .. b
6:          begin
7:             if wip+1 = $ and wip+1 ∈ Trans(Ni, Nj) then /* Nj = final */
8:                return acceptance
9:             else if wip+1 ∈ Trans(Ni, Nj) then
10:                begin
11:                   dn :=< Ni, wip+1, ip, ip + 1 >;
12:                   if DAG[dn]=off then
13:                      begin
14:                         DAG[dn] := on;
15:                         newP := allocate_Processor();
16:                         newP.inf :=< Nj, ip + 1, ↑dn, ↑dn, nil >;
17:                         parallel_SR(newP)
18:                      end
19:                   p2 := topEtdn; .. c
20:                   P.inf :=< nil, nil, p, p1, p2 >;
21:                   reduce2_action(P)
22:                end
23:          end
24:end
```

a N は状態番号の集合．
b $STI(N_i)$ は状態 N_i で使用する遷移情報の集合．
c $top_{Et_{dn}}$ は Et_{dn} の先頭を指すポインタ．

図 7 shift 動作
Fig. 7 Shift action.

* %はダミー記号

- for each $p \in P$ parallel do begin…end

  P の各要素 p に対して begin…end で括られた式を並列に実行する.
- parbegin…parend

  parbegin, parend で括られた … の中のすべての式を並列に実行する.

アルゴリズムを図 6, 7, 8, 9 に示す.

```
1: procedure reduce1_action(P);/* reduce1 動作 */
2: begin
3:    Ni := P.inf.N; ip := P.inf.i; p := P.inf.pdn; p1 := P.inf.p1;
4:    for each rule ∈ RULE[Ni] parallel do .. a
5:       begin
6:          A := lsym(rule); .. b
7:          α := rstr(rule);.. c
8:          γ := lizard(α); .. d
9:          X := last(α); .. e
10:         ri :=< [A → γ · X], ip, Ni >;
11:         reduceaction(ri, p, p1)
12:      end
13:end
```

a $RULE[N_i]$ は，状態 N_i で reduce 動作に使用される文法規則の集合.

b $lsym(rule)$ は文法規則（または，マーカ付規則）$rule$ の左辺記号を返す関数.

c $rstr(rule)$ は，文法規則 $rule$ の右辺記号列を返す関数.

d $lizard(\alpha)$ は α から α の最右端記号を除いた記号列を返す関数.

e $last(\alpha)$ は α の最右端記号を返す関数.

```
1: procedure reduceaction(ri, p, p1);
2: begin
3:    Ni := ri.N; rwm := ri.rwm; ip := ri.i;
4:    A := lsym(rwm);
5:    β := lstr_of_marker(rwm); .. a
6:    if β = nil then
7:       for each Nj ∈ N parallel do
8:          begin
9:             if Trans(Ni, Nj) ∈ STI(Ni) and (↓ p).N ∈ Trans(Ni, Nj)
10:            then
11:               begin
12:                  dn:=< (↓ p).N, A, (↓ p).l, ip >;
13:                  if DAG[dn] = off then
14:                     begin
15:                        DAG[dn] := on;
16:                        newP := allocate_processor();
17:                        newP.inf :=< Nj, ip, ↑ dn, p1, nil >;
18:                        parallel_SR(newP)
19:                     end
20:               end
21:         end
22:   else /* β ≠ nil */
23:      if Rt↓p1.F[ri]=off then
24:         begin
25:            Rt↓p1.F[ri] := on;
26:            append(Et↓p1, ri) .. b
27:         end
28:end
```

a $lstr\_of\_marker(rwm)$ は rwm の右辺で，マーカの左にある記号列を返す関数.
例) $lstr\_of\_marker([A \rightarrow \alpha \cdot \beta]) = \alpha$

b $append(Et_{\downarrow p1}, ri)$ は，登録テーブル $Et_{\downarrow p1}$ に ri を追加する関数.

図 8 reduce 1 動作
Fig. 8 Reduce 1 action.

3.2.3 解析例

文法 G1 から構成した図 1 (b) の LR 状態遷移図と入力記号列 W="aaaa$" を使って，最適化アルゴリズムによる解析例を図 10 に示す. 図は，shift, reduce 1, reduce 2 の各動作を 1 単位時間として，DAG ノードとプロセッサおよび reduce テーブルの状態を表している. 各動作を行っているプロセッサには，# 記号をつけている. 登録テーブル Et_i に格納されている reduce 情報の右に記されたプロセッサ P_j は，1 時刻前に，その reduce 情報を格納した（あるいは格納しようとした）プロセッサを表している. また，図中，時刻 12 で，$<N_2, A, 1, 4>$ の右の (P_4) は，既に生成されている $<N_2, A, 1, 4>$ を P_4 が再び生成したことを表している.

3.2.4 構文木の作成

構文木を作成するためには，DAG ノード間，および reduce 情報と DAG ノード間をリンクしなければならない. そこで，DAG ノード dn と reduce 情報 ri へのポインタを要素としてもつリスト ($[\uparrow dn | \uparrow ri]$) を格納するためのフィールド（これを PLF (Pointer List Field) と呼ぶ）を，各 DAG ノードと reduce 情報に用意する.

例えば，dn_1 : $<0, a, 0, 1>$ に割り当てられたプロセッサが reduce 1 動作で，dn_2 : $<0, A, 0, 1>$ を生成するとき，dn_2 の PLF に $[\uparrow dn_1 | nil]$ を格納する. また，dn_4 : $\langle 2, A, 1, 2 \rangle$ に割り当てられたプロセッサ

```
1: procedure reduce2_action(P); /* reduce2 動作 */
2: begin
3:    p := P.inf.pdn; p1 := P.inf.p1; p2 := P.inf.p2;
4:    repeat
5:       begin
6:          if ↓ p2 ≠ nil then
7:             begin
8:                Ni := (↓ p2).N;
9:                rwm := (↓ p2).rwm;
10:               ip := (↓ p2).i;
11:               A := lsym(rwm);
12:               α := lstr_of_marker(rwm);
13:               if ↓ p.X = last(α) then
14:                  begin
15:                     ri := [lshift_marker(rwm), ip, Ni]; .. a
16:                     reduceaction(ri, p, p1)
17:                  end
18:               p2 := p2 + 1 /* 次の reduce 情報を受けとる. */
19:               P.inf :=< nil, nil, ↑ p, p1, p2 >;
20:               reduce2_action(P)
21:            end
22:      end
23:   forever
24:end
```

a $lshift\_marker(rwm)$ は，rwm のマーカを 1 記号分左に移動させたマーカ付規則を返す関数.

図 9 reduce 2 動作
Fig. 9 Reduce 2 action.

が reduce 1 動作で ri: $<[A \rightarrow A \cdot A], 2, 3>$ を生成するとき，ri の PLF に $[\uparrow dn_4 | nil]$ を格納する．ri を使って，dn_2 に割り当てられたプロセッサが reduce 2 動作によって dn_5: $<0, A, 0, 2>$ を生成するとき，dn_5 の PLF に $[\uparrow dn_2 | \uparrow ri]$ を格納する（図 11 参照）．

このように DAG ノードや reduce 情報の PLF にリストの格納を行うとき，$O(n)$ 個の書き込み競合が生じる可能性がある．そのため，DAG ノードに割り当てたプロセッサとは別に，リストの格納を行うためのプロセッサを起動する必要がある．

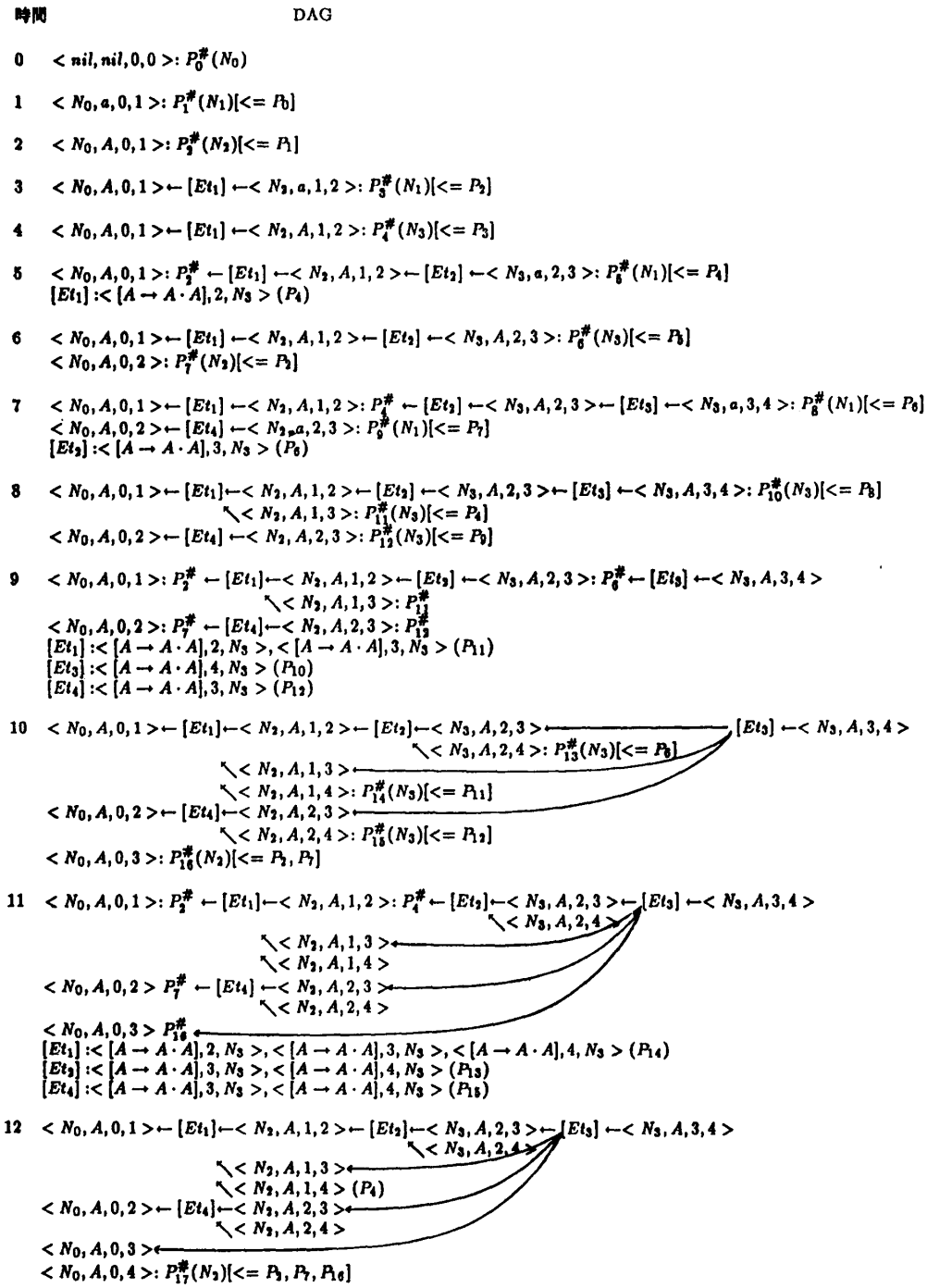


図 10 最適化アルゴリズムの解析例
Fig. 10 Parsing of the input string "$a\ a\ a\ a\ \$$" by the optimized algorithm.

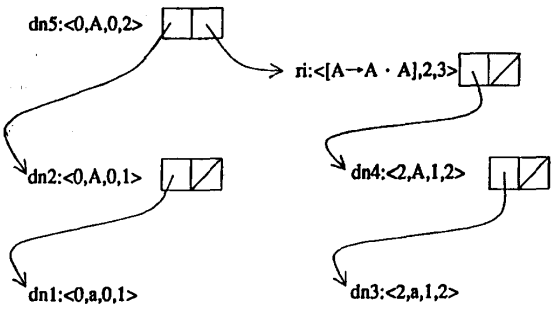


図 11 リンクの例
Fig. 11 Example of linking with pointer lists.

3.3 最適化アルゴリズムと基本アルゴリズムとの等価性

最適化アルゴリズムと基本アルゴリズムについて，以下の定理が成り立つ．

[定理 1] 最適化アルゴリズムと基本アルゴリズムの受理する言語は等しい．

[証明] 定理が成り立つことを言うには，1）基本アルゴリズムが受理する言語を最適化アルゴリズムでも受理できること，2）最適化アルゴリズムで受理するすべての言語が基本アルゴリズムでも受理されること，を証明すればよい．

1）については，最適化アルゴリズムで生成される DAG ノードの連結が，基本アルゴリズムで生成される DAG の連結を保存し，DAG ノードの生成に必要な情報が欠落しないことを言えば良い．まず，DAG メモリを使ったプロセッサ起動の制御について考える．DAG ノードは，$<N, X>$ 成分が LR 状態遷移図の特定の状態を表し，X と l, r $(l<r)$ 成分により部分解析木の頂点を表すので，同一の DAG ノードをもつプロセッサが行う shift, reduce 1 動作は等しい．したがって，既に生成された DAG ノードをもつプロセッサの起動を抑制しても，同じ動作を別のプロセッサが行っているので，その動作結果を使うことができれば解析に必要な情報を得ることになる．つぎに reduce 動作の分割について考えると，基本アルゴリズムで行う reduce 動作のために必要な情報は，その reduce 動作で使用される文法規則と reduce 動作を開始した状態番号，および入力記号へのポインタであるが，最適化アルゴリズムでは，それらを reduce 情報として保持している．この reduce 情報は，shift 動作で生成された DAG ノード $<N, w_i, i-1, i>$ $(0 \leq i \leq n)$ に割り付けられた reduce テーブル rt に格納されるので，rt に格納される reduce 情報は，$<N, w_i, i-1, i>$ を生成するプロセッサの reduce 2 動作に使用される．すなわち，基本アルゴリズムで生成される DAG ノードの連結を，最適化アルゴリズムは reduce テーブルを介して実現していることになる．したがって，解析に必要な情報の欠落はなく，他のプロセッサが行った動作結果を利用できることが保証される．よって，基本アルゴリズムの reduce 動作によって生成される DAG ノードは，最適化アルゴリズムの "分割された reduce 動作" によっても生成できる．ゆえに，基本アルゴリズムで受理できるすべての言語は最適化アルゴリズムでも受理できる．

2)については，reduce テーブルに蓄えられた reduce 情報に対して，reduce 2 動作を行うか否かを，reduce 情報の X 成分を見て決定しているため，基本アルゴリズムでは生成されない誤った DAG ノードを生成することはない．よって，最適化アルゴリズムで受理できる言語は基本アルゴリズムでも受理できる．以上から証明された．

4. 最適化アルゴリズムの評価

アルゴリズムの性能を評価するため，時間・プロセッサ数・メモリ数のそれぞれの上限を，入力長 n の関数として求める．

[補題 1] DAG のノード数と reduce テーブルのサイズは，ともに $O(n^2)$ である．

[証明] DAG ノード $<N, X, l, r>$ $(l<r)$ の N と X, および reduce 情報 $<[A \rightarrow \alpha \cdot \beta], i, N>$ の $[A \rightarrow \alpha \cdot \beta]$ と N は文法によって決まり，DAG ノードの l, r, reduce 情報の i は入力長 n に依存する．アルゴリズムの性質より，同じ DAG ノードや reduce 情報が複数存在することはない．したがって，DAG のノード数は $O(n^2)$ であり，1 個の reduce テーブルに格納される reduce 情報の数は $O(n)$ である．reduce テーブルは，shift 動作で生成された DAG ノード $<N_j, w_i, i-1, i>$ に割り付けられるため，reduce テーブルの数は入力長に比例し，$O(n)$ である．したがって，reduce テーブルの総サイズも $O(n^2)$ である．

[補題 2] DAG ノード dn, reduce 情報 ri をインデックスとした DAG $[dn]$, $Rt_{dn}.F[ri]$ へのアクセス時間は $O(1)$ である．

[証明] 補題 1 より，DAG と reduce テーブルのサイズは $O(n^2)$ である．しかし，十分大きな n に対して連続メモリ領域を割り付けておけば，DAG ノード

および reduce 情報をインデックス化できる*. したがって，DAG ノード dn あるいは reduce 情報 ri をインデックスとした DAG[dn] あるいは $Rt_{dn}.F[ri]$ へのアクセス時間は $O(1)$ である．

[補題 3] reduce 1/reduce 2 の動作を行うプロセッサが，reduce 情報を登録テーブルに書き込む時に書き込み競合が生じて待たされることがあるとしても，1 文の解析を行う間において，その待ち時間の合計は高々 $O(n)$ 時間である．

[証明] reduce テーブルに格納される reduce 情報の数が高々 $O(n)$ であることから明らか．

[補題 4] reduce 2 動作を行う 1 個のプロセッサが 1 文を解析する間に扱う reduce 情報をすべて処理するために要する時間は高々 $O(n)$ である．

[証明] reduce 2 動作を行う 1 個のプロセッサそれ自体が扱う全 reduce 情報を処理するために要する時間は，reduce 情報の読みとり先の reduce テーブルに他のプロセッサによって reduce 情報が書き込まれるのを待つのに要する時間の総和と，reduce 情報の書き込み先の reduce テーブルに reduce 情報を書き込むまでに待たされる時間の総和，および読みとり先の reduce テーブルに格納される全 reduce 情報を処理するために要する時間の総和を加え合わせたものである．reduce 2 動作を行う 1 個のプロセッサが，reduce 情報を書き込むために待たされる時間は，補題 3 より $O(n)$ である．また，同様に考えると，reduce 情報を読みとるまでに待たされる時間も高々 $O(n)$ であるため，reduce 2 動作を行う 1 個のプロセッサが 1 文の解析において扱うすべての reduce 情報を処理するために要する時間は高々 $O(n)$ である．

[定理 2] 最適化アルゴリズムの解析時間は高々 $O(n)$ である．

[証明] reduce 情報は，各 reduce テーブルごとにパイプライン的に並列に処理されるので，解析時間は，1 個の reduce テーブルに格納されるすべての reduce 情報を処理するのに要する時間に等しく，その時間は補題 3 より高々 $O(n)$ である．

プロセッサ数とメモリ空間は，解析を行う場合と，構文木を作成する場合とでは異なる．まず，解析を行う場合について述べ，次に構文木の作成の場合について述べる．

[定理 3] 解析を行う場合，プロセッサ数もメモリ空間も $O(n^2)$ である．

[証明] 解析を行う場合，DAG ノードや reduce 情報が既に他のプロセッサによって生成されているか否かのチェックを行うだけでよいため，メモリ空間の大きさは DAG メモリのサイズと reduce テーブルのサイズを加え合わせたもので，使用するプロセッサは，DAG に割り当てたものだけである．よって，プロセッサ数，メモリ数とも補題 1 より $O(n^2)$ である．

[定理 4] 構文木を作成する場合，プロセッサ数，メモリ空間の大きさとも，$O(n^3)$ である．

[証明] 構文木を作成する場合，DAG ノードと reduce 情報へのポインタを要素とするリストを格納するためのフィールド PLF が，各 DAG ノードおよび各 reduce 情報に対して必要となる．ある DAG ノード $dn:<N,X,l,r>(l<r)$ に対して，dn の PLF に格納されるリストの数は高々 $c(r-l-1)$ (c は定数) であるため，構文木を作成するために使用されるプロセッサ数，メモリ空間の大きさは，ともに高々 $O(n^3)$ である．

5. 他の手法との比較

ここで，3.2 節で述べた，最適化アルゴリズム（以下では，本手法と呼ぶ）と冨田法[2]，冨田法を拡張した沼崎らの手法[5]および PAX[1] との比較を行う．

冨田法は入力に同期した解析を行うため，チョムスキー標準形の曖昧な文法の場合では，解析時間は $O(n^2)$ である[8]．ただし，特異な文法では，解析時間が文法サイズ m に対して指数関数的 $\Omega(n^m)$ に増加する[13]ことが知られている．一方，本手法では一般の文脈自由文法で，時間・空間積が $O(n^3)$ ですみ，冨田法におけるオーダの問題を解決している．

冨田法を拡張した沼崎らの手法[5]は，重複動作を避けるためにストリーム通信の技法を用いている点，およびプロセスの割り当て方などで本手法と異なる．沼崎らの手法では，ストリーム同期のために解析時間が $O(n^2)$ となるが，本手法では高々 $O(n)$ 時間である．

次に PAX と比べてみると，本手法では，LR 法の長所[8),14]から，PAX に比べて無駄なプロセッサの起動が少ない．また，PAX ではプロセッサが曖昧さに比例[12]して $O(c^n)$ (c は文法に依存する定数) 必要となるが，本手法ではプロセッサの数は $O(n^2)$ ですむ．

* DAG ノードをインデックスとしてアクセスできる DAG メモリ，あるいは reduce 情報をインデックスとしてアクセスできる参照テーブルをあらかじめ作成することは，入力長の上限を考えたインプリメントの問題である．

6. まとめ

本稿では一般の文脈自由文法に対して解析時間 $O(n)$, プロセッサ数 $O(n^2)$ の効率的な並列構文解析アルゴリズムを提案し，その性能についての理論的な評価を行った．このアルゴリズムでは，構文木を作成する場合，プロセッサの数は $O(n^3)$ となるが，逐次型解析アルゴリズムで，構文木を作成する手間が $O(n^4)$ であることを考えると，最適な並列アルゴリズムであるといえる．

本稿では述べなかったが，本手法については，実際に必要となるプロセッサ数や時間についてのシミュレーション実験による性能評価を行っている．その結果，曖昧さがない場合などでは，プロセッサ数が $O(n)$ 程度で抑えられることがわかっているが，この点についてはさらに詳細な理論的評価が必要である．また，時間およびプロセッサ数，メモリ数のオーダの係数が大きくならないことも同時に確かめており，実際的なアルゴリズムであることを確認している．これらについては稿を改めて報告する．

今後は，本手法を市販の並列マシン上に実現し，実際の解析性能に対して評価を行うとともに，並列形態素解析[15]との融合手法の実現をはかる．

謝辞 東京工業大学の田中穂積教授と同大学院博士課程の沼崎浩明氏には，冨田法の評価について有益な助言をいただいた．九州大学の中村貞吾助手には，日頃より有益な討論をしていただく．ここに深く感謝の意を表したい．

参考文献

1) Matsumoto, Y.: A Parallel Parsing System for Natural Language Analysis, *Proc. 3rd International Conference of Logic Programming, Lecture Notes in Computer Science, 225*, pp. 396-409 (1986).
2) Tomita, M.: An Efficient Augmented-Context-free Parsing Algorithm, *Computational Linguistics*, Vol. 13, No. 1-2, pp. 1-6 (1987).
3) Rytter, W.: Parallel Time $O(\log n)$ Recognition of Unambiguous Context-free Languages, *Information and Computation*, Vol. 73, pp. 75-86 (1987).
4) 安留，青江：自然言語の曖昧構文解析に対する並列処理，情報処理学会研究報告 (SF, PL-12), pp. 109-118 (1988).
5) 沼崎，田村，田中：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，情報処理学会研究報告 (NL-74), pp. 33-40 (1989).
6) Valiant, L. G.: General Context-Free Recognition in Less than Cubic Time, *J. Comput. Syst. Sci., 10*, pp. 308-315 (1975).
7) Aho, A. V. and Ulmann, J. D. (土居訳)：コンパイラ，情報処理シリーズ7，培風館 (1986).
8) 峯，谷口，雨宮：文脈自由文法の並列構文解析，信学技報 (NLC89-7), pp. 1-8 (1989).
9) Mine, T., Taniguchi, R. and Amamiya, M.: An Efficient Parallel Parsing Algorithm for Context-free Grammars, *PRICAI '90*, pp. 239-244 (1990).
10) Knuth, D. E.: On the Translation of Languages from Left to Right, *Inf. Control*, Vol. 18, No. 6, pp. 607-639 (1965).
11) 笠井：並列計算モデルと計算の複雑さ，情報処理，Vol. 26, No. 6, pp. 568-574 (1985).
12) 高岡，雨宮：文脈自由言語のあいまい性関数について，電子通信学会論文誌，Vol. 58-D(1), No. 1, pp. 31-35 (1975).
13) Johnson, M.: The Computational Complexity of Tomita's Algorithm, *International Parsing Workshop '89*, pp. 203-208 (1989).
14) 沼崎，田村，田中：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*, pp. 183-192 (1989).
15) 峯，谷口，雨宮：日本語の並列形態素解析，第40回情報処理学会全国大会論文集，pp. 452-453 (1990).

(平成2年10月1日受付)
(平成3年7月8日採録)

峯 恒憲 (正会員)

昭和38年生．昭和62年九州大学工学部情報工学科卒業．平成元年同大学院総合理工学研究科情報システム学専攻修士課程修了．現在，同大学院博士課程に在学中．音声認識，並列自然言語処理の研究に従事．

谷口倫一郎（正会員）

昭和30年生．昭和53年九州大学工学部情報工学科卒業．昭和55年同大学院工学研究科修士課程修了．同年九州大学大学院総合理工学研究科情報システム学専攻助手．平成元年より同助教授．工学博士．画像理解，画像処理システム，並列処理システムの研究に従事．電子情報通信学会，人工知能学会各会員．

雨宮 真人（正会員）

昭和17年生．昭和42年九州大学工学部電子工学科卒業．昭和44年同大学院工学研究科修士課程修了．同年日本電信電話公社武蔵野電気通信研究所入所．以来，プログラミング言語・処理系，自然言語理解，データフロー・アーキテクチャ，並列処理，関数型／論理型言語，知能処理アーキテクチャ，等の研究に従事．現在九州大学大学院総合理工学研究科情報システム学専攻教授．工学博士．電子情報通信学会，ソフトウェア科学会，人工知能学会，IEEE，AAAI 各会員．