論理文法の並列構文解析†

松本 裕治††

並列論理型言語上での実現に向いた，論理文法のための並列構文解析法について述べる．与えられた自然言語文法の文法カテゴリ（非終端記号）それぞれを並列に動くことのできるプロセスとして記述し，一般にCommitted-Choice Language と呼ばれる並列論理型言語の特徴を活かした実現法を提案する．論理型言語に基づく文法記述形式として，DCG (Definite Clause Grammars), XG (Extraposition Grammars), GG (Gapping Grammars) などが提案されているが，本解析法では，これらすべての文法を統一的な手法で取り扱うことができる．また，本手法では解析中に解析の途中結果を記録するための副作用を使うことなく，しかも同一処理を繰り返すことがない．さらに，文法規則自体が論理型言語のプログラムとして完全にコンパイルされているので，インタプリティブな処理がなく，逐次実行環境でも上記論理文法のための効率的な構文解析法となっている．

1. はじめに

最近の計算言語学の潮流として，LFG[6]や GPSG[5] などのように自然言語の文法をユニフィケーションを基本とする論理的枠組みによって記述しようという試みが盛んである．一方，論理プログラミングの分野では，DCG[11]，XG[12]，GG[2),3)]などの文法記述形式が提案されており，ユニフィケーションを基本とする文法の有望な記述言語と考えられる．

本論文では，これらの論理型言語に基づく文法記述形式を対象とする並列構文解析法を紹介する．ここで述べるのは，著者による DCG のための並列構文解析法[9] の拡張である．今までの方法では，基本的に文脈自由文法の構文構造を対象にしていたが，拡張版では，より記述力の高い構文構造を持つ Extraposition Grammar (XG) や Gapping Grammar (GG) を取り扱えるようになった．本論文では，GG などの文法記述形式で書かれた文法を GHC[14]，Parlog[1]，Concurrent Prolog[13] などいわゆる Committed-Choice Parallel Language と呼ばれる並列論理型言語で書かれた構文解析プログラムに変換する方法を示す．変換後のプログラムでは，単語や文法カテゴリなどの文法要素がすべて論理型言語の述語として表現されており，それら一つ一つが並列に動き得るプロセスとして働く．つまり，文法規則を解釈しながら実行するプログラムがあるのではなくて，文法自体が論理型言語のプログラムにコンパイルされる．また，個々のプロセスが構文解析木の部分木に対応しているため解析中の途中結果を副作用として登録しておく必要がない．得られるプログラムは Prolog のプログラムとして実行することも可能で，コンパイラをもつ Prolog の上では逐次実行においても効率のよい構文解析システムとして用いることができる[10]．GG などに関してはこれまで効率のよいアルゴリズムが知られていなかったが，今回の拡張により，本手法は GG のための逐次型構文解析システムとしての応用も考えられる．

2. 文脈自由文法の並列構文解析

2.1 基本アルゴリズム

我々の並列構文解析の基本的なアルゴリズムは文献 9), 10) で述べられたものと同等である．まず，その特徴を列挙する．

(1) 基本的にチャート法[7]の並列化である．

(2) 終端記号や非終端記号が並列に動くプロセスとして表されている．

(3) 構文解析木の部分木の完成がプロセスの生成に対応しており，途中結果を副作用として残しておく必要がない．

特に，これらの特徴のうち(2)と(3)について簡単に説明する．チャート法や Earley のアルゴリズム[4] などの副作用を利用する構文解析法（これらを総称してタブロー法と呼ぶ）は同一計算を避けるために副作用を用いている．本構文解析法では，途中結果をプロセスとして生かしておくために，副作用を用いなくても副作用を利用する方法と同等の効果を達成している．

ただし，一般の構文解析のアルゴリズムの目的は，与えられた入力文が与えられた文法から生成可能であるかどうかを判定することであるが，我々の構文解析

† Parallel Parsing of Logic Grammars by YUJI MATSUMOTO (Machine Inference Section, Information Science Division, Electrotechnical Laboratory).

†† 電子技術総合研究所パターン情報部推論システム研究室

では，入力文がどのように構文解析可能であるか，すべての可能な構文解析木を得ることを目標としている．したがって，タブロー法で，(構造上に異なるが)同一の頂点をもつ部分解析木の生成の抑制を行っている操作に対応することは本解析法では行っていない．

文脈自由文法に対する並列構文解析アルゴリズムを次の文法を例に用いて説明する．

(1) sentence --〉 np, vp.
(2) np --〉 det, noun.
(3) np --〉 np, coconj, np.
(4) pp --〉 prep, np.

例えば，文法規則(1)を用いてボトムアップ解析を行う場合，チャート法では，npを根とする部分構文解析木(チャート法では inactive edge と呼ばれる)が得られると，sentence --〉 np・vp というラベルを持った枝(チャート法では active edge と呼ばれる)をチャートに追加する．いずれ，その枝に隣接する位置にvpを根とする部分解析木が得られると，sentence という根を持つ新しい解析木が得られることになる．ここでは，npやvpを独立なプロセスと考え，sentence --〉 np・vp のようなデータを送受するようなモデルを考える．このデータ自身は(1)の文法規則のうち，npの直後の位置までの解析が終わったことを示している．このような複雑なデータを扱うことを避けるため，次のように文法規則の右辺に id1 などの「識別子」を与えることにする．ただし，これらの識別子は，文法規則から構文解析プログラムを生成するトランスレータによって自動的に作られるので，ユーザは気にする必要はない．

(1′) sentence --〉 np, id1 vp.
(2′) np --〉 det, id2 noun.
(3′) np --〉 np, id3 coconj, id4 np.
(4′) pp --〉 prep, id5 np.

例えば，id4 は，文法規則(3)の coconj の直後までの解析が終了した状態，つまり，チャート法における np --〉 np, coconj・np というデータに対応している．

さて，np というプロセスはどのような処理を行えばよいだろうか．np は上の文法規則の右辺中の4箇所に現れるが，右辺の先頭に現れる np とそれ以外の np では，対応する処理が異なる．つまり，右辺の先頭に現れる np は所定のデータすなわち識別子を出力すればよいが，それ以外の np は特定の識別子だけにしか反応しない．np が行うべき処理を GHC のプログラムによって記述すると次のようになる．

```
np(X, Y) :- true|
    np1(X, Y1), np2(X, Y2), merge(Y1, Y2, Y).
np1(X, Y) :- true|Y=[id1(X), id3(X)].
np2([ ], Y) :- true|Y=[ ].
np2([id4(X)|Xt], Y) :- true|
    np(X, Y1), np2(Xt, Y2), merge(Y1, Y2, Y).
np2([id5(X)|Xt], Y) :- true|
    pp(X, Y1), np2(Xt, Y2), merge(Y1, Y2, Y).
np2([_|Xt], Y) :- otherwise|np2(Xt, Y).
```

np1 が右辺の先頭に現れる np の処理の定義，np2 がそれ以外の np の処理の定義である．np1 は単に二つの識別子，id1 と id3，を出力しているだけであるが，np2 については右辺の先頭以外の個々の np について特定の識別子に対する処理が記述されている．例えば，np2 の第2番目の節は，np2 が id4 という識別子を受け取ると，新たな np というプロセスを生成することを示している．これは，文法規則(3)の coconj の直後までの解析が終了している状態で np が見つかったときに，全体として新しい np が得られることに対応している．この処理の様子を**図1**に示す．id5 は文法規則(4)の prep の直後まで解析が進んだ状態に対応しているので，np2 はそれを受け取ると pp を完成させる．

このようにして，文法規則の右辺に現れるすべての非終端記号に関する処理を並列論理型言語のプロセスとして定義することができて，自動的に並列構文解析プログラムが得られる．

構文解析の目的は，入力として与えられた単語列のすべてを使って文法を満足する文の構造を作れるかどうかを調べることである．ボトムアップ解析では入力単語列すべてを使って文法の開始記号（今の例では，sentence)を生成できれば解析が終了したことになる．そのために，sentence には，次のような節を特別に定義する．

```
sentence (begin, Y) :- true|Y=[end].
```

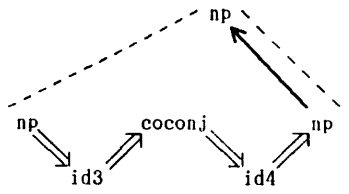

図1 解析の例
Fig. 1 Parsing example.

そして，例えば，“the man walks.” という文を解析したいときには，

the (begin, X1), man (X1, X2), walks (X2, Y),
fin (Y).

という初期ゴールを実行すればよい．fin は解析の終了を調べるための述語で，fin が end というデータを受け取ったとき，入力単語列が文として正しい構造をしていたことになる．

sentence が文法規則の右辺に現れるときには，sentenceという述語が上のnpと同様に定義されるが，そのときには，上の sentence の定義を，sentence 2 の定義に追加すればよい．

本解析法は，トップダウン予測を用いて処理効率を上げたり，DCG のような文脈自由文法に基づく文法記述形式のための構文解析システムとして用いることができるが，詳細については文献 9)，10) を参照されたい．

3. Gapping Grammar の並列構文解析

3.1 Gapping Grammar[2),3)★]

Dahl らによって提案された Gapping Grammar (GG) は，必ずしも連続しない文法記号間の書き換えを許す文法規則を含む文法理論である．例えば，

(5) object, coconj, gap(G), object --〉
coconj, gap(G), object.

などが典型的な GG の文法規則の例で，gap(G) は任意の記号列を表している．つまり，この文法規則はボトムアップに読めば，ある記号列を挟んで coconj と object があるときに，それらをまとめて左辺のような記号列に書き換えてよいということである．このように，文法規則を記述する際に，非連続な記号列を対象にすることができ，しかも文法規則の左辺に複数の記号を記述することを許している．ただし，左辺の gap(G) は右辺の gap(G) と同じ記号列でなければならない．gap は文法規則内に幾つ現れてもよい．その場合，同じ変数を引数として持つ gap は同じ記号列を表していなければならない．

図 2 に，この文法規則を用いた構文解析結果の例を示す．破線で囲まれた部分が上の文法規則が用いられた部分，楕円で囲まれたのが gap(G) に対応する部分である．

Dahl らは，GG に対するトップダウン構文解析法を提案しているが，DCG をトップダウンに実行するのと同様，後戻り型の単純な解析法を用いているため，無駄な繰り返しが頻繁に起こる可能性がある．

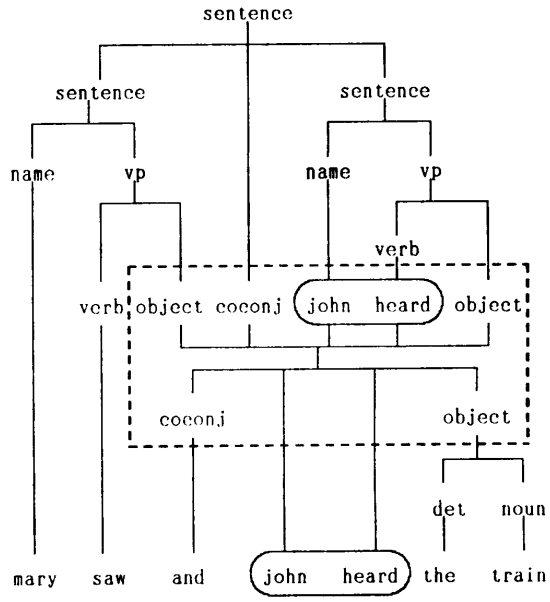

図 2 GG による解析例
Fig. 2 Sample analysis by GG.

次節では，前節で紹介した文脈自由文法のための並列構文解析法に修正を加えることにより，同一の解析の繰り返しを避け，しかも並列に実行可能な GG の並列構文解析法を提案する．

3.2 Gapping Grammar の並列構文解析

前章で述べた並列構文解析の手法を適用するため，基本的には GG のボトムアップ構文解析を前提にしている．まず，次の命題が成り立つことに注意されたい．

[命題]

GG のボトムアップ解析においては，文法規則中の gap(_) として許される記号列を終端記号列のみに限っても構文解析の可能性に影響を与えない．(命題終)

つまり，この命題の意味は，gap(_) を任意の記号列として構文解析が可能ならばそれを終端記号列のみに限っても構文解析が可能であるし，その逆も成り立つということである．詳しい証明は省略するが，gap(_) に対応する部分への文法規則の適用をなるべく遅らせた解析を考えればよい．また，この制限を与えることにより，見掛けは異なるが本質的に等価な構文解析結果の生成を抑えることができることにも注意してほしい．

さて，GG に基づいて構文解析を行う方法を説明し

★ Dahl らは，最近，Gapping Grammar のことを Discontinuous Grammar と名称変更しているが，本論文では Gapping Grammar と呼ぶことにする．

よう．前章と同様に GG の文法規則の右辺についても文法規則内の特定の位置を示す識別子を用意する．ただし，次のように，

```
object, coconj, gap(G), object -->
                    coconj, gap(G), object,
                                id 6
```

gap(_) の存在を無視して右辺の任意の記号間に識別子を割り当てる．ここでは，gap(_) が右辺の末尾には現れないと仮定する．そのような場合についての処理は，次節で述べる．

gap(_) の存在する位置に割り当てられた識別子は特別に取り扱われる．上の文法規則の場合，id 6 はある終端記号列を隔てて，object というプロセスに受け取られる必要がある．つまり，id 6 は幾つかの終端記号を読み飛ばした後に object プロセスに受け取られればよい．この読み飛ばしの処理を行うための述語 gg を定義し，例えば，入力 "John saw and Mary heard the train." を解析するための初期呼び出しを次のようにする．

```
john(begin, A),
saw(A, B1), gg(A, saw, B2), merge(B1, B2, B),
and(B, C1), gg(B, and, C2), merge(C1, C2, C),
mary(C, D1), gg(C, mary, D2), merge(D1, D2, D),
heard(D, E1), gg(D, heard, E2), merge(E1, E2, E),
the(E, F1), gg(E, the, F2), merge(F1, F2, F),
train(F, G), fin(G).
```

ここに，gg は次のように定義され，特定の識別子 (gap(_) の位置に現れる識別子) を読み飛ばす働きをする．

```
gg([ ], _, Y) :- true|Y=[ ].
gg([id6(X, Stack)|Xt], Word, Y) :- true|
    Y=[id6(X, [Word|Stack])|Yt],
    gg(Xt, Word, Yt).
gg([_|Xt], Word, Y) :- otherwise|
    gg(Xt, Word, Y).
```

gap の位置に現れる識別子がほかにもある場合には，その識別子についても id 6 と同様の定義を付け加えればよい．id 6 が引数を二つ持っていることに注意されたい．第2の引数は読み飛ばされた単語を蓄えておくためのスタックになっている．id 6 は文法規則(5)の右辺の gap(G) までの解析が終了した状態を表しているから，object プロセスに受け取られると，文法規則の左辺の記号列に対応するプロセスの列を生成する．そのとき，左辺中の gap(G) の位置には，それまでに読み飛ばされた単語列が挿入される．具体的には，object は id 6 を受け取ると次のようなプロセスを生み出せばよい．また，id 6 を生成するのは，次のように，coconj である．

```
coconj 1(X, Y) :- true|Y=[id 6(X, [ ])].
object 2([id6(X, Words)|Xt], Y) :- true|
    object(X, X1), coconj(X1, X2),
    reverse(Words, RWords),
    expand(X2, RWords, X3),
    object(X3, Y1),
    object 2(Xt, Y2), merge(Y1, Y2, Y).
```

これらの定義は，coconj プロセスと object プロセスのうち id 6 だけに関連する節を取り出したものである．reverse は，リストの反転を行い，読み飛ばされた単語列を正しい順に並べ直す．expand は，並べ直された単語列に対応するプロセスを生成する述語で，次のように定義される．

```
expand(X, [ ], Y) :- true|X=Y.
expand(X, [Word|R], Z):- W=..[Word, X, Y1]|
        call(W), gg(X, Word, Y2),
        merge(Y1, Y2, Y), expand(Y, R, Z).
```

第2章のガード部にある W=..[Word, X, Y1] は，Word を述語名とし，X と Y 1 を引数に持つ述語を作り，それをWに代入するという組み込み述語であり，Word で示される単語に対応するプロセスを作っている．ボディ部の call(W) によって，このプロセスが呼び出されている．

このように文脈自由文法用の並列構文解析プログラムに gap(_) を取り扱うための拡張を行うことによって，GG 用の並列構文解析プログラムを得ることができる．なお，gap(_) の位置に識別子を割り当てることも，そのような識別子に対して gg 節の定義することも，トランスレータが自動的に行う．したがって，GG 特有の文法規則が含まれない場合には，トランスレータは文脈自由文法の場合とまったく同じ構文解析プログラムを生成する．

文法規則(5)が構文解析の過程で用いられる様子を示したのが**図3**の概略図である．各プロセスの引数は省略した．また，この解析に直接関係しないプロセスも省略した．実際には，これら以外のあらゆるプロセスも並列に動いている．図中，白抜きの矢印はデータの送受，黒い矢印はプロセスの生成を表している．

以上が基本的な GG の並列構文解析法であるが，今までの説明では，gap(_) が文法規則の右辺の最後

の記号として現れる場合を考慮に入れていなかった．次節では，このような場合について考察する．

3.3 Extraposition Grammar の並列構文解析

Pereira によって提案された Extraposition Grammar (XG)[12]は，左外置変形を扱えるように DCG を拡張したもので，文法規則の左辺に非連続の記号列の記述を許している．例えば，関係代名詞の記述を次のように行うことができる．

relmarker … np --〉 relpronoun.

この文法規則において，「…」は任意の記号列に対応し，関係代名詞 (relpronoun) が関係節の始まりを表す目印 (relmarker) のほかに痕跡としての名詞句 (np) の役目を果たしていることを記述している．この文法規則は GG によって次のように表される文法規則と等価である．

(6) relmarker, gap(G), np --〉 relpronoun, gap(G).

このように，XG の文法規則は，右辺の最後にのみ gap(_) を許した GG の文法規則と見ることができる．本節の題名を XG の並列構文解析としたが，本節の目標は，右辺の最後にも gap(_) が現れることを許した場合の GG の並列構文解析である．

(6)のような文法規則に対しても次のように gap(_) の位置に識別子を割り当てる．

relmarker, gap(G), np --〉 relpronoun, gap(G).
id 7

前節で扱った GG の文法規則には，右辺の gap(_) の直後には具体的な記号があって，その記号に対応するプロセスがその gap(_) の位置に割り当てられた識別子を処理していた．しかし，(6)では，id 7 を受け取るべきプロセスがない．このため，(6)の右辺の解析を終了させるプロセスがなく，このような位置に割り当てられた識別子は，さらに特殊な扱い方をしなければならない．ここでは，右辺の最後に現れる gap(_) に対応する識別子の処理を gg プロセスに任せることにする．

文法規則(6)による解析の概略を図 4 に示す．図のように，たまたま runs という単語と併置された gg によって id 7 が受け付けられたとすると，rel_marker, john, saw, np というプロセス列を生成すればよい．ただし，生み出された np プロセスの出力は，この gg プロセスの隣のプロセスではなく，runs プロセスに渡されなければならない．なぜなら，id 7 を処理した gg は saw までの入力情報しか使っていないからである．このように，gg プロセスは，扱うべき識別子が，右辺の最後に現れた gap(_) のものかそれ以外のものかによって，データの出力を，併置された単語プロセスに与えるかその直後のプロセスに与えるか区別しなくてはならない．

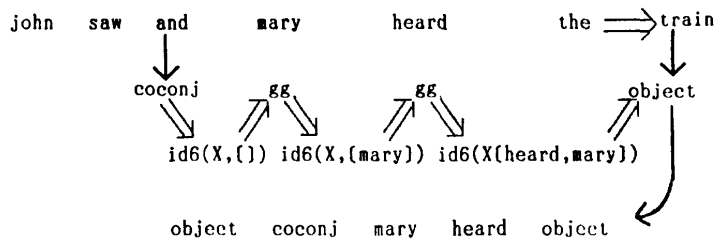

図 3 文法規則(5)による構文解析の概略図
Fig. 3 Parsing process concerning grammar rule (5).

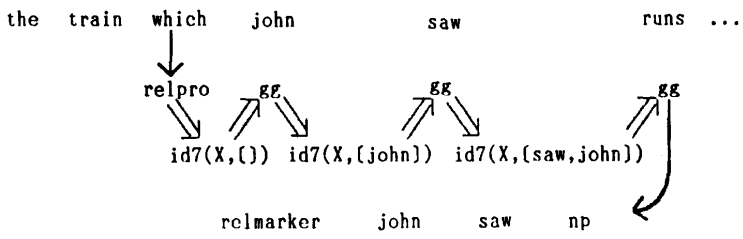

図 4 文法規則(6)による構文解析の概略図
Fig. 4 Parsing process concerning grammar rule (6).

以上の考察をもとに識別子 id 7 を扱う gg プロセスを定義すると次のようになる．

```
gg([id 7(X, Stack)|Xt], Word, NewX, Y) :- true|
    Y=[id 7(X, [Word|Stack])|Yt],
    relmarker(X, X1),
    reverse(Stack, RWords),
    expand(X1, RWords, X2),
    np(X2, Nx1), merge(Nx1, Nx2, NewX),
    gg(Xt, Word, Nex2, Yt).
```

gg は，前節のように 3 引数ではなく，4 引数になっている．第 3 引数が追加された引数である．ボディ部の第 1 行目は，今までと同様に，識別子を読み飛ばして次のプロセスへ渡すための処理である．2 行目以降が，読み飛ばしを終了する処理であり，(6)の文法規則の左辺に対応するプロセス列を生成している．このときの出力が (merge を経由して) 第 3 引数に渡されている．

このように，単語プロセスは併置されている gg プロセスからもデータを受け取ることがあるので，初期呼び出しを次のように変更する．

```
john(begin, A),
saw(AA, B1), gg(A, saw, NewA, B2),
    merge(A, NewA, AA), merge(B1, B2, B),
and(BB, C1), gg(B, and, NewB, C2),
    merge(B, NewB, BB), merge(C1, C2, C),
                ……
train(FF, C1), gg(F, train, NewF, G2),
    merge(F, NewF, FF), merge(G1, G2, G),
fin(GG), gg(G, _, NewG, _), merge(G, NewG, GG).
```

各単語プロセスが入力として受け取るのは，直前のプロセスからの出力および併置されている gg プロセスの第3引数からの出力であるので，それらをひとまとめにするための merge プロセスが追加されている．また，文末またはその直前で(6)のように文法規則の右辺の解析が終了する場合に対処するために，文末の単語および fin と併置する gg プロセスが用意されている．

右辺の末尾以外に現れる gap(_) に対応する識別子については，gg プロセスの定義は以前と本質的に同じでよく，第3引数に対しては何もしない．よって，新しい gg プロセスの定義は次のようになる．

```
gg([ ], _, NewX, Y) :- true|NewX=[ ], Y=[ ].
gg([id6(X, Stack)|Xt], Word, NewX, Y) :- true|
    Y=[id6(X, [Word|Stack])|Yt],
    gg(Xt, Word, NewX, Yt).
gg([id7(X, Stack)|Xt], Word, NewX, X) :-
    true|
    Y=[id7(X, [Word|Stack])|Yt],
    relmarker(X, X1),
    reverse(Stack, RWords),
    expand(X1, RWords, X2),
    np(X2, Nx1), merge(Nx1, Nx2, NewX),
    gg(Xt, Word, Nx2, Yt).
gg([_|Xt], Word, NewX, Y) :- otherwise|
    gg(Xt, Word, NewX, Y).
```

以上の拡張により，XG を含めた一般的な GG の並列構文解析プログラムを，与えられた文法規則の集合から自動的に生成することができる．ただし，この方法では，XG の文法規則に対して文中のあらゆる位置で読み飛ばし終了の可能性を考慮して文法規則の左辺のプロセスの展開を行うので無駄が多い．今野らの方法[8]のように予測を活かした XG の構文解析手法を並列構文解析にいかに適用できるかを探ることが今後に残された問題である．

4. むすび

代表的な論理文法の並列構文解析法について述べた．本手法の特徴として，論文の最初に次のような特徴を挙げた．

(1) チャート法の並列実現である．

(2) 終端記号および非終端記号が並列に動作するプロセスとして表現されている．

(3) 各プロセスは，構文解析中に得られる部分解析木に対応しており，これらを副作用の形で残しておく必要がない．

ここでは，これら以外に特徴として言えることを幾つか考えてみよう．まず，本手法で得られるプログラムは GHC で記述されていることからもわかるように，決定的で後戻りのないプログラムである．ただし，文法自身が持つ曖昧性は，プロセスに展開することによってすべて尽くされている．また，与えられた文法規則はすべてプログラムの形に変換され，構文解析は解釈実行の過程を一切含まない．これらの特徴は処理の実行速度の高速性を意味し，実際，文脈自由文法に対しては，本手法によって得られるプログラムを Prolog のプログラムとして逐次実行しても，実用的な構文解析システムとして用いることができる[10]．

本論文では詳細を省略したが，解析の途中でトップダウン予測を用いることができ[9),10]，無駄な処理の低減を図ることができる．また，基本的にはボトムアップ解析を行っているので，左再帰規則による無限ループの問題も起こらない．

文脈自由文法のための並列構文解析法を，Gapping Grammar の並列構文解析法として，これらの特徴を損なわないように拡張したが，考えられる問題点を指摘しておこう．GG ではまだ実用的な文法が開発されていない．したがって，GG の構文解析法としての処理速度の実用性については具体的な結果を示すことができない．しかし，小さな文法による実験の感触としては，まだ不満足であると思われる．ただし，これは Gapping Grammar 自身の問題であり，現在の Gapping Grammar の仕様を忠実に実行する限り膨大な探索空間を扱わなければならないのはやむを得ない．これは，GG の文法規則が Chomsky の0型文法の文法規則を特殊例として含んでしまっていることからもわ

かるように，記述力が豊か過ぎるのである．これは，GG の考案者たちが指摘しているように，文法の記述者の意図しなかったような解析結果を生じる原因にもなっている．今後，何らかの制限を付けられたより実用的な GG の提案を待つとともに，本研究が効率的に解析可能な GG の部分集合の特定に役立つことを期待したい．

謝辞 本並列構文解析の基本的なアイデアは，著者が英国インペリアルカレッジに留学中に生まれたものである．Prof. Kowalski を始めとする Logic Programming Group の諸氏に感謝する．その後 ICOT にて研究を続ける機会を与えてくださった電総研中島隆之パターン情報部部長，ICOT 淵一博所長に感謝する．東工大田中穂積教授には有益な助言と指導を頂き，著者が並列構文解析を始めるきっかけを与えてくださった．改めて感謝したい．ICOT 杉村領一氏には，本構文解析の逐次版の作成を通じて有益な議論をして頂いた．最後に，古川康一次長を始めとする ICOT の諸氏に感謝したい．

参 考 文 献

1) Clark, K. L. and Gregory, S.: PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, Imperial College (1984).
2) Dahl, V. and Abramson, H.: On Gapping Grammars, *Proc. Second International Conference on Logic Programming*, pp. 77-88, Uppsala, Sweden (1984).
3) Dahl, V.: More on Gapping Grammars, *Proc. the International Conference on Fifth Generation Computer Systems, Tokyo*, pp. 669-677 (1984).
4) Earley, J.: An Efficient Context-Free Parsing Algorithm, *C. ACM*, Vol. 13, No. 2, pp. 94-102 (1970).
5) Gazdar, G., Klein, E., Pullum, G. and Sag, I.: *Generalized Phrase Structure Grammar*, Basil Blackwell, Oxford (1985).
6) Kaplan, R. M. and Bresnan, J.: Lexical-Functional Grammar: A Formal System for Grammatical Representation, Bresnan, J. (ed.), *The Mental Representation of Grammatical Relations*, Chap. 4, pp. 173-281, MIT Press, Cambridge, Massachusetts (1982).
7) Kay, M.: Algorithm Schemata and Data Structures in Syntactic Processing, Technical Report, CSL-80-12, Xerox PARC (Oct. 1980).
8) 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システム，コンピュータソフトウェア，Vol. 3, No. 2, pp. 19-29 (1986).
9) 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).
10) 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，コンピュータソフトウェア，Vol. 3, No. 4 (1986).
11) Pereira, F. C. N. and Warren, D. H. D.: Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks, *Artif. Intell.*, Vol. 13, pp. 231-278 (1980).
12) Pereira, F.: Extraposition Grammars, *AJCL*, Vol. 7, No. 4, pp. 243-256 (1981).
13) Shapiro, E. Y.: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report, TR-003 (1983).
14) Ueda, K.: Guarded Horn Clauses, *Proc. The Logic Programming Conference*, ICOT (1985).

(昭和 62 年 6 月 1 日受付)
(昭和 62 年 11 月 11 日採録)

松本 裕治（正会員）

昭和 30 年生．昭和 52 年京都大学工学部情報工学科卒業．昭和 54 年同大学院工学研究科修士課程情報工学専攻修了．同年電子技術総合研究所入所．推論システム研究室にて自然言語処理，論理プログラミングの研究に従事．昭和 59～60 年英国インペリアルカレッジ客員研究員．昭和 60～62 年(財)新世代コンピュータ技術開発機構に出向．並列計算，知識表現等にも興味を持つ．日本ソフトウェア科学会会員．