並列構文解析における並列度と左外置処理に関する一考察

西嶋愉一、田中穂積
(東京工業大学　工学部)

1．　はじめに

松本[松本 86a]による構文解析システムPAXは、DCG[Pereira 80]で記述された文法を、GHC[Ueda 85]のような並列論理型言語のプログラムに変換し、直接それをボトムアップに、並列に構文解析をするプログラムとして実行するシステムである。アルゴリズム自体はAXと呼ばれるが、並列論理型言語による実現をPAX、Prologによる実現をSAXと呼んでいる。PAXは副作用なしに並列に動作が可能であり、それが結果的にPrologのコンパイラ向きのコードとなっているために、逐次的に実行しても高い効率が得られる(SAX)という特徴を持ち、SAXはBUP系の構文解析システムに比べ5ないし10倍高速であることが報告されている[杉村 86]。しかし、PAXを並列に実行した場合の効率、すなわち並列度については今まで検討されていない。また、文法記述形式がDCGであるため、左外置を扱うことができず、文法の記述性にやや難がある。

本論文では以上のPAXに関する課題について考察する。まず、PAXの並列度について、並列論理型言語の解析ツールを作成し、小規模な文法について解析を行う。これにより、並列推論マシン上でのPAXの実行効率を予測することができる。第2に、PAXにスタックを用いて痕跡を扱う機能を付加することにより、文法記述形式を、DCGに左外置の処理を取り入れて拡張されたXGSに対応することを考える。これにより、左外置を含む文の高速な構文解析を行うことを可能にする。

2．　ＰＡＸの概要

以下の説明では、並列論理型言語としてGHCを用いる。また、文法は純粋な文脈自由文法に限定する。

PAXのアルゴリズムは基本的にはLeft Corner Parsing[Aho 72]であるが、Left Corner Parsingにおける2種類の処理、すなわち、現在注目している非終端記号に対して、それ自身を左隅の要素とする解析木を作ろうとする処理と、今までに得られている部分的な解析木を、より完全に近い解析木に作り上げようとする処理を同時に行うことが特徴である。

Left Corner Parsingにおいては、文法規則のどこまでが使われているかを知る必要があるが、このため、DCGで記述された文法規則の右辺に、(1)~(5)のように識別子を与える。

```
s --> np id1, vp.                (1)
np --> det id2, noun.            (2)
np --> det id3, noun id4, srel.  (3)
np --> np id5, pp.               (4)
vp --> v id6 , np.               (5)
```

ここで、id1からid5が識別子である。これらの識別子は、文法規則中の特定の位置に1対1に対応している。PAXでは、非終端記号は第1引数を入力、第2引数を出力とする2引数の述語として表現され、これらの識別子をストリームとして受け渡すことによって解析が進められる。例えば、ある非終端記号がid4を受け取ったとすると、これは(3)の文法規則のnounの位置までの解析が終了したことを意味する。

PAXでは非終端記号の完成が述語の呼び出しに対応する。非終端記号が完成した時、既に述べたように2つの仕事がある。第1の仕事は、自分自身を左隅の要素として解析木を構成することである。例えば、名詞句が完成したとすると、名詞句を左隅、すなわち右辺の先頭要素として持つ文法規則は(1)、(4)の2つあるから、これらに対応した処理として、2つの規則を同時に適用し、id1、id5を同時に出力する。具体的には、(6)のような節で定義される動作をする。

```
np1(X,Y) :- | Y = [id1(X),id5(X)].(6)
```

第2の仕事は、部分的に完成している解析木を、自分自身を用いてより完成に近づけることである。これは、文法規則の右辺の先頭以外の場所に非終端記号が存在する場合に行われる。npについて考えると、(5)の規則により、以下の定義が必要である。

```
np2([],Y) :- | Y = [].                (7)
np2([id6(X)|T],Y) :- |
        vp(X,Y1),
        np2(X,Y2),
        merge(Y1,Y2,Y).               (8)
np2([_|T],Y) :- otherwise |
        np2(T,Y).                     (9)
```

(8)によって、id6をnpが受け取った時にvpが構成される。解析に複数の可能性がある場合、(6)のように複数の識別子が渡されるから、自分につながらない識別子が入力された場合、(9)のように捨てる必要がある。(7)は入力が空の時空リストを返すもので、停止条件として使われる。

以上のような定義のうち、(5)のようなものをタイプ1節、(7)~(9)のようなものをタイプ2節と呼ぶ。名詞句が構成されるとnpが呼び出されるが、npはnp1とnp2によって(10)のように定義される。

```
np(X,Y) :- |
        np1(X,Y1),
        np2(X,Y2),
        merge(Y1,Y2,Y).          (10)
```

他の非終端記号についても同様に定義できる。

タイプ1節は自分自身が左隅となり得るすべての規則に対応する識別子の集合を出力し、タイプ2節はそれらの識別子を取り込み、他の識別子を出力したり、部分木の親となる非終端記号に対応する述語を呼び出したりする。これらの処理を独立に並列に実行し、出力をmergeでまとめあげることにより、並列性を引き出すことができる。

文法規則の変換の方法は以上のとおりである。これを用いて文を解析するには、(11)のような初期ゴールを呼び出せばよい。

```
?- i([begin],D1),open(D1,D2),the(D2,D3),
   door(D3,D4),fin(D4).          (11)
```

ここで、beginというのは文の先頭を表す識別子である。さらに、sのタイプ2節に、次のような定義を追加する。

```
s2([beginIT],Y) :- |
                Y = [endIY1],
                s2(T,Y1).        (12)
```

文が完成すると s2が呼び出されるが、s2はbeginを受け取ると、識別子endを次のプロセスに渡す。非終端記号が識別子beginを受け取るのは、その非終端記号が文の先頭にあることを意味し、sは目的としているものであるから、出力されるendは文末を意味していることになる。述語finがendを受け取ったとすると、それは入力文全体からsが構成された、すなわち解析が成功したことを示す。finはユーザによって定義され、解析の結果行うべき処理を記述している述語である。呼び出し(11)によって文が解析される様子を図1に示す。

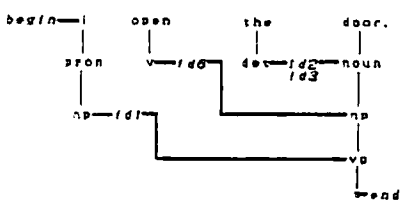

図1 PAXによる並列構文解析の様子

PAXの基本的なアルゴリズムは以上のとおりである。ここでは純粋な文脈自由文法のみを対象としたが、DCGの機能を実現する手法、トップダウン予測等の詳細については[松本 86a][松本 86b]を参照されたい。

3. PAXの並列度

3.1 PAXのプログラムの特徴

ここでは、対象とする文法は純粋な文脈自由文法に限定する。また、並列論理型言語としてはGHCを用いる。

PAXにおいて、非終端記号に関する述語は、すべて第1引数を入力、第2引数を出力に固定している。タイプ1節では、第1引数にどんなものが入っても同じ処理を行うため、入力変数のユニフィケーションに特に制限をする必要はないが、タイプ1節を展開したあと行われるのはトップダウン的な予測であり、その時点では具体的な値が束縛されている必要がある。一方、タイプ2節は、第1引数を選択条件として排他的に記述されているため、入力変数に具体的な値が束縛されていないと実行することができない。何らかの値が束縛されていれば、複数のタイプ2節のどの節がコミットされるかはヘッドユニフィケーションの時点で特定することができる。

ここで並列度を考えるに当たっては、入力変数のユニフィケーションとプロセスのサスペンドに関する制限をGHCの場合より単純化する。すなわち、入力変数に具体的な値が束縛されていないプロセスはサスペンドし、それ以上展開できない。この制限により、タイプ2節は呼び出しの時点でどの節が選ばれるか決定できる。

3.2 並列度動的解析ツール

前節において述べた考え方に基づいて、並列度の解析ツールを作成した。以下、その概略について述べる。

本解析ツールでは、解析対象プログラムにゴール（この場合入力文）を与え、そのゴールが実行される過程で並列度、ゴールの呼ばれる回数などのデータを収集する。その意味で動的である。これは並列論理型言語のための汎用の解析ツールとして使用することも可能である。ただし、PAXのプログラムだけを対象にしているため、いくつかの制限を設け、ツールを単純化している。

（1） 並列度については、現在注目しているレベルにおいてリダクション可能な（アクティブな）ゴールの数であると定義する。ここで言うレベルとは、[尾内 84]において、実時間の代りに導入された考え方で、1レベルは、ユニフィケーションが実行され、次のゴール（AND関係にあるサブゴール列から形成される）が生成されるまでを言う。例えば、図2のようなGHCのプログラムに対して、?-a.という呼び出しを実行することを考えると、その実行は図3のようになる。

```
a :- | b , c.
b :- | b1 , b2.
c :- | c1 , c2.
b1.
b2.
c1.
c2.
```

図2　プログラム例

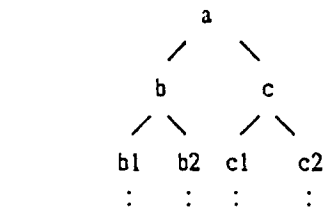

図3　実行例

図3の例では、aからb、cに展開されるまで、また、b、cからb1、b2…に展開されるまでがそれぞれ1レベルである。（5）で述べるようにOR並列を考慮しないので、ここで得られる並列度はAND並列度である。

（2）　変数に関する束縛は時間遅延なしに親ゴールおよび変数を共有する他のプロセスに転送される。

（3）　ゴールのリダクションが可能かどうかは、述語の第1引数に具体的な値が束縛されているかどうかで判断する。値が束縛されていればリダクション可能であり、束縛されていなければそのゴールはサスペンドする。PAXのプログラムはすべて第1引数を入力、第2引数を出力としているため、このように制限しても処理には影響しない。

（4）　リダクション可能なサブゴールはAND並列に実行される。本解析ツールでは、リダクション可能なサブゴールをすべて同時に、すなわち1レベルでユニファイする。

（5）　OR関係にある節については、既に述べたとおりPAXではすべて選択条件が排他的に記述されている。（3）の入力変数とプロセスのサスペンドに関する制約により、実行される時点では入力変数に値が束縛されているから、どの節が選ばれるかは既に決定されている。したがって、OR並列については考慮しない。

（6）　対象とする並列論理型言語のプログラムはガードの左側にヘッド以外の項が存在しないものに限定する。OR関係にある節の選択条件はすべてヘッドに記述しなければならない。したがって、対象とする文法は文脈自由文法、あるいはDCGでも補強項のないものに限定される。さらに、この制限のため、PAXにおいてトップダウン予測を行うtp_ck述語と、終了時に解析の成功を判定するfin述語の定義が変更されている。

3．3　考察

前節の動的解析ツール上で、文法規則数76個の文法（付録A）をPAXに変換したプログラムに例文（付録B）を与えて解析した。結果を表1-aに示す。

表1-a中、”長さ”は文の長さ（単語数）、”木の高さ”は実験に用いた文法により得られる解析木の高さ、”レベル”は解析に要したレベル、”総ゴール数”は呼び出されるゴールの数の合計である。並列度は全てのレベルにおける並列度の平均値としているので、（総ゴール／レベル）に一致する。”最大プロセス数”のうち左側は1レベル中のアクティブなゴール数の最大値、右側はサスペンドしているゴールを含んだゴール数の最大値である。

今回作成した動的解析ツールでは、呼び出されたゴールの入力変数がユニファイされていないとそのゴールはサスペンドし、その時のレベルではそれ以上実行できない。しかし、入力変数によらず同一の動作をするタイプ1節では、必ずしもサブゴールの展開を抑制する必要はなく、展開できるところは展開を進めてしまう方が時間の面では効率的であるとも考えられる。しかし逆に、サスペンドするプロセスの数は増加する。GHCの言語仕様では、パッシブパート（ガードの左側）で、その節の中から外へのユニフィケーションをしようとした時にサスペンドするようになっている。このような仕様ならば、入力変数がユニファイされていなくても、タイプ2節の呼び出しまでは実行を進めることができ、したがって、より高い並列度を期待することができる。今回作成した解析ツールでは、並列言語、および並列推論マシンが確定していないことから、仕様を単純化している。異なる戦略を用いた場合の並列度については今後の検討課題である。

PAXについて解析を行った結果から、次のようなことが言える。

（1）　並列実行の場合の実行時間に相当する、解析に要するレベル数は解析木の高さに対してほぼリニアに増加する傾向を示す。しかし、解析木の高さが同じでもレベルが大きく異なる例もあり、レベルは解析木の高さだけでなく、解析中に使用される非終端記号に影響されると考えられる。ある非終端記号が文法規則の右辺の先頭に存在するものが多い文法では、その非終端記号のタイプ1節が長いストリームを1度に出力してしまう。ストリームが長いと、それを入力として受け取るタイプ1節で

はトップダウン予測、タイプ2節ではそれ自身の呼び出し回数がストリームの長さに比例して増加し、mergeの呼び出し回数も増加する。これらはすべてレベルを増大させる要因となる。例えば、一般動詞に関する文法規則数は多いため、タイプ1節から出力されるストリームが(12)のように長くなっている。このため、レベルが増大する。

```
v1(X,Y) :- |
        tp_ck(X,vp,New_X1),
        tp_out(New_X1,
            [id58(New_X1),id59(New_X1),
            id60(New_X1),id62(New_X1)],Y1),
        vp(New_X1,Y2),
        merge(Y1,Y2,Y).                    (13)
```

（2） 並列度については、実験に用いた13個の例文については平均で13.9という値を得ている。しかし、1レベル中に展開されるプロセスの数（最大並列度）は実験に用いた比較的短い文では88が最大で、これを処理できる以上にプロセッサを多くしても効率改善には寄与しない。ボトムアップ構文解析では、構文木を葉から根へたどる形になるため、プロセス数は収束する方向に動作する。したがって、並列処理において台数効果の上がる例ではないが、実用上は十分な速度が得られるものであると言える。

（3） 並列度は、レベルと同様、解析に用いられる文法規則の複雑さに依存していると考えられる。並列度を大きくするような規則とは、実験に用いた文法規則の例では図4のような、右辺の要素が1個で、そのような規則が1つの非終端記号について数多くある場合である。実験に用いた例では、他の規則と合わせてnpのタイプ1節は(14)のような定義となっているが、この例のような場合、1つのnpのタイプ1節から1度に展開されるサブゴールの数が多くなり、並列度が上がることがわかる。

```
obj --> np.
obj_one --> np.
obj_two --> np.
pred --> np.
subj --> np.
```

図4 文法規則の例

```
np1(X,Y) :- |
        tp_out(X,[id22(X)],Y1),
        tp_ck(X,obj,New_X2),obj(New_X2,Y2),
        merge(Y1,Y2,Y3),
        tp_ck(X,obj_one,New_X3),
        obj_one(New_X3,Y4),merge(Y3,Y4,Y5),
        tp_ck(X,obj_two,New_X4),
        obj_two(New_X4,Y6),merge(Y5,Y6,Y7),
        tp_ck(X,pred,New_X5),pred(New_X5,Y8),
        merge(Y7,Y8,Y9),
        tp_ck(X,subj,New_X6),subj(New_X6,Y10),
        merge(Y9,Y10,Y).                   (14)
```

（4） 文の長さが並列度に与える影響については、定量的な傾向を知るには至っていない。文が長ければ、（3）に該当するような非終端記号が現われる回数も多くなりがちであるので、並列度は短い文に比べれば高くなる。一方、（1）に該当する非終端記号の占める割合が高いと、並列度が上がる以上にレベルが大きくなってしまう。解析に使われる非終端記号とレベル、並列度について定量化することが今後の検討課題として残されている。

4. ＰＡＸにおける左外置変形の処理

4．1 左外置変形と文法記述形式ＸＧＳ

英語において、関係代名詞節の埋め込み文は、宣言文から名詞句が欠落した句構造をしているが、これは、先行詞が埋め込み文の左側に移動したものと考えることができる。

Chomsky の痕跡理論によれば、このように語句が移動する場合、移動する前の場所に痕跡(trace)を残して移動すると考え、このような語句の移動を左外置（left extraposition）と呼んでいる。

痕跡を考慮しない場合、例えば、名詞句 "the man who loves her" について考えると、埋め込み文の主語が欠落していて、その句構造は図5のようになる。

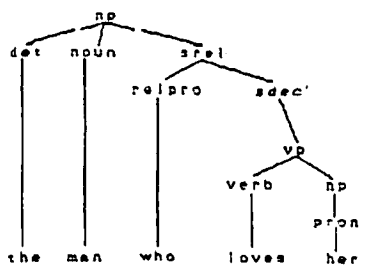

図5 名詞句"the man who loves her"の句構造

名詞句が欠けているため、完全な宣言文のカテゴリと区別して、sdec'という文法カテゴリを導入している。このような形式で文法を記述すると、名詞句の欠落した文法カテゴリに対応する文法規則をいちいち記述しなければならない。

名詞句の欠落した構造についてひとつひとつ文法規則を記述するのは、文法が大きくなればなるほど困難であり、さらに、本来の英語の文法には存在しないような、不自然な文法カテゴリを導入しなければ文法規則が記述できない。また、規則数も増大する。これらのことから、文法の見通しが悪化し、開発、修正が難しくなる。

左外置を考慮すると、これらの問題の多くは解決する。先程と同じ例、"the man who loves her"について考えると、その句構造は図6のようになる。図6においては、語句が移動しても句構造が変化していないことに注意されたい。その結果、関係代名詞節の埋め込み文についても、完全な宣言文と同じ文法カテゴリと考えることができる。

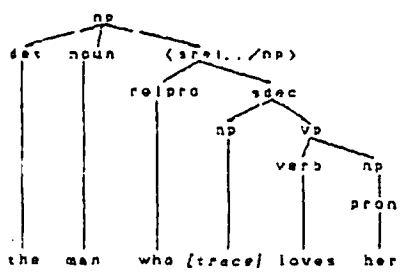

図6　左外置を考慮した名詞句"the man who loves her"の句構造

したがって、名詞句の欠落した構造をひとつひとつ記述したり、それに伴って不自然な文法カテゴリを導入する必要もなくなる。これによって、文法規則数の増大を防ぎ、文法の見通しをよくすることができる。

XGS (Extraposition Grammar with Slash Category)[今野 86]は、以上のような考え方に基づいて、BUPを拡張したBUP-XG用に開発された文法記述形式で、DCGのシンタックスを拡張した形をとっている。

文法規則に、関係代名詞節を解析するための規則を追加することを考える。DCGの場合は数多くの規則を追加しなければならなかったが、XGSでは(15)~(16)の規則を追加するだけでよい。

```
np --> np , < srel../np >.        (15)
srel --> relpro , sdec.           (16)
```

(15)の規則中の記号"../"はスラッシュと呼ばれるシンタックス・シュガーであり、例えばa../bは、カテゴリaの中に痕跡が一つ存在し、その痕跡はb（スラッシュ・カテゴリと呼ぶ）の直接構成要素であることを示す。(15)では、srel../npと記述することで、srelの中に痕跡を直接支配するnpが存在することを示している。(16)からわかる通り、左外置を考慮することによって、関係代名詞節中の埋め込み文も通常の宣言文と同じsdecとなるため、既に述べたように、名詞句の欠落したカテゴリを導入したり、そのための規則を記述する必要がないことに注意されたい。"<"（オープン）、">"（クローズ）は、Rossの複合名詞句制約を満足するためのシンタックス・シュガーであり、"<"と">"で囲まれたカテゴリから外へは語句が移動してはならないことを示す。その他、XGSに関する詳細は[今野 86]を参照されたい。

4．2　左外置処理の実現法

痕跡の処理をトップダウン構文解析の中で行う場合、外置リスト（スタック）を用いる方法が考えられている。PereiraがXGで用いているのがこの手法である。

例として、名詞句"a man who walks around"を解析することを考える。図7はnpがdet、noun、srelに展開され、nounの解析が終わった時点を示している。ここでスタックにスラッシュ・カテゴリnpをプッシュする。

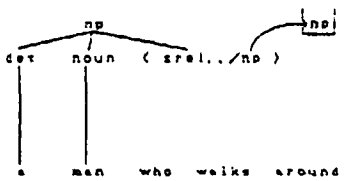

図7　トップダウン法による解析過程(1)

srelがrelproとsdecに展開され、さらにsdecをnpとvpに展開しようとするが、この時、入力文の中にはnpがないので、痕跡を発見したと解釈して、スタックからnpをポップして解析を続行し、完成させる（図8）。

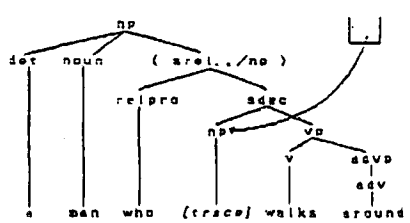

図8　トップダウン法による解析過程(2)

この方法では、スタックにスラッシュ・カテゴリをプッシュするのは、痕跡が存在しうるカテゴリ、例えば

srelの解析時に限定できるため、精度の高い解析が可能である。また、スタックからスラッシュ・カテゴリをポップするタイミングも、トップダウンに解析していって入力文の中にカテゴリが存在しない時に限定できるため、タイミングが明確であり、誤った解析を行ってしまうことがない。しかし、トップダウン構文解析には左再帰規則が扱えない、文法が大きくなると効率が極度に低下するといった欠点がある。

痕跡の処理を、スタックを用いて、ボトムアップ構文解析の中で行うことを考えてみると、まず、純粋なボトムアップ構文解析では、次にくるカテゴリの予測を行わないため、スタックにスラッシュ・カテゴリをプッシュするタイミングが明確でない。例えば関係節の埋め込み文について言えば、名詞は先行詞になりうるので、名詞が見つかった直後にプッシュすることになる。このため、痕跡の存在し得る範囲を限定するのが難しい。さらに、痕跡は入力文中には出現しないので、入力文の単語と単語の間ほとんどすべてに痕跡の存在を仮定しなければならない。この2つのことから、解析精度を上げるのが難しく、また、効率も悪化する。

PAXのアルゴリズムは基本的には純粋なボトムアップ構文解析である。BUPのgoal節のような、トップダウン的な予測を行いながらつねに解析を制御する述語は使われず、トップダウン的な予測はフィルタとして無駄な識別子を除去するのに使われているに過ぎない。したがって、単純に実現したのでは、純粋なボトムアップ構文解析の持つ問題点をそのまま持っていることになる。しかし、文法規則中の位置を識別子で同定できるため、痕跡をサーチする範囲を限定することは可能である。また、スラッシュ・カテゴリをポップする際も、スラッシュ・カテゴリに対応するプロセスを並列に実行することで、実行時間の増大に対処することが可能であると考えられる。以下、PAXにおける左外置の処理法について述べる。なお、XGSに対応するPAXの拡張版をPAX/XGと呼ぶことにする。

２章で述べたように、PAXでは解析の履歴は識別子のストリームによって表される。複数の可能性があれば、可能性の数だけのストリームが流される。XGSのためのスタックも同様に扱う必要がある。すなわち、可能性の数だけの識別子のストリームとスタックを流さなければならない。

このため、非終端記号を表すプロセスの間を流れるデータの構造は、オリジナルのPAXでは(17)のような構造をしているが、左外置の処理をするために、(18)のようにオリジナルの識別子のストリームとスタックを組にしたもののリストに変更される。

```
[id1([begin]),id2([begin])…]   (17)

[[[id1([begin]),id2([begin])],[]],
  └        ストリーム        ┘ スタック
 [[id4([begin])],x(np,[])]…]
  └   ストリーム  ┘ └ スタック ┘        (18)
```

それぞれの組において、識別子のストリームは通常のPAXと同様に解析の履歴を表し、スタックはその解析において用いられているスタックの状態を表している。従って、解析途中のある時点において、スタックからスラッシュ・カテゴリをポップする選択としない選択がある場合には、それぞれに対応するストリームとスタックの組が生成されることになる。

スタックの構造は、(19)のようなxという名前の構造体である。xの第1引数はスタックトップのカテゴリ、第2引数はスタックトップがプッシュされる前のスタックの状態を表す。空のスタックは空リスト "[]" で表す。

```
x(Top,Xold)                    (19)
```

PAX/XGにおいて痕跡を含む名詞句の解析をしている様子を図10から図13に示す。図9のような文法規則が与えられていて、名詞句"the man that loves her"を解析することを考える。

```
np --> det id1, noun id2, < srel../np >.  (20)
np --> pron.                              (21)
srel --> relpro id3, sdec.                (22)
sdec --> np id4, vp.                      (23)
vp --> v id5, np.                         (24)
```

図9　文法規則

まず各単語の辞書引きが行われ、続いて、規則(20)によって、nounからid2が出力される（図10）。この時、スタックにスラッシュ・カテゴリnpがプッシュされ、ストリームと一緒に出力される。

```
the          man                  that loves her
 |            |                    |    |   |
det→[id1,[]]→noun→[id2,x(np,[])] relpro  v  pron
```

図10　PAX/XGによる解析過程(1)

スタックが空でない時は、非終端記号の処理が終わるごとにスタックトップのカテゴリをポップし、その処理結果をひとつの組にして非終端記号の処理結果とマージして出力する。例えば、ここではnounからストリームとしてid2、スタックとしてx(np,[])が出力されている。システムはmanのあとに痕跡としてnpが存在すると仮定してスタックトップからnpをポップし、図11の

ように処理をする。

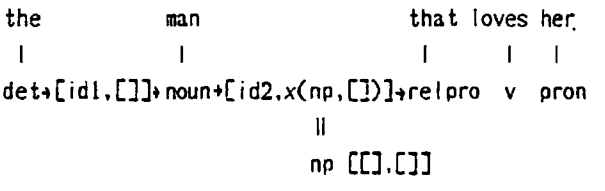

図11 PAX/XGによる解析過程(2)

この例では、名詞のあとに名詞句がくるような規則はないので、トップダウン予測によって無駄な可能性は捨てられ、ここでのnpの出力は空ストリームになる。続いてrelproの解析を行い、入力のid2によって、relproからid3とx(np,[])の組が出力される。ここでもスタックのポップを行い、id3を入力としてnpの処理を行う（図12）。

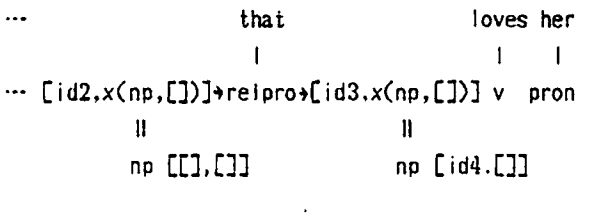

図12 PAX/XGによる解析過程(3)

id3からはnpにつながり得ることがトップダウン予測によってわかり、npは規則(23)によってid4を出力する。この結果と先のrelproの出力id3がマージされ、解析が続行される。その後、vの解析時にid3はトップダウン予測によって切られ、id4に、"loves her"からvpを構成してつないでいくことによって名詞句の解析が終了する（図13）。

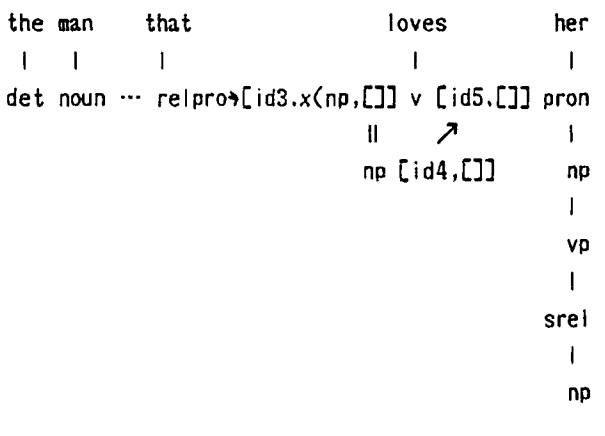

図13 PAX/XGによる解析過程(4)

以上の例からわかるように、PAX/XGではスタックを無条件でポップしているが、次のプロセスにストリームが渡されると無駄な解析はトップダウン予測で捨てられるので、それ以上無駄なストリームが流されることは抑えられる。

スタックプッシュのタイミングについては、スタックを文法規則上の位置と1対1対応する識別子と組にして流しているので、関係節につながらないような名詞についてスタックに余計にプッシュしてしまうことはない。例えば、規則(25)では名詞の解析後にスタックにnpをプッシュするが、規則(26)では何もプッシュしない。これは、名詞に入力される識別子がid4であるか、id6であるかによって区別される。

```
np --> det id4, noun id5, < srel../np >.   (25)
np --> det id6, noun id7, ncomp.           (26)
```

4．3 インプリメント

前節までで述べた左外置の処理を行うための実現法について述べる。

1) 非終端記号の定義

スタックは先に述べたとおり構造体で表し、これを識別子のストリームと組にして次のプロセスに渡すことにより実現している。例えば、(27)の文法規則は(28)から(30)のPAX/XGのプログラムにコンパイルされる。

```
a --> b id1, c id2, d.                     (27)

b1([X,InputStack],Y) :- !
    tp_ck(X,a,New_X1),
    Y = [[[id1(New_X1)],InputStack]].      (28)

c2([[id1(X)|T],InputStack],Y) :- !
    c2([T,InputStack],Y1),
    Y = [[[id2(X)],InputStack]|Y1].        (29)

d2([[id2(X)|T],InputStack],Y) :- !
    a([[X,InputStack]],Y1),
    d2([T,InputStack],Y2),
    merge(Y1,Y2,Y).                        (30)
```

ここで注意されたいのは、タイプ1、タイプ2節ともに、入力には識別子ストリームとスタックの組を1組与え、出力はそれらのリストを完全な形で出していることである。 タイプ1、タイプ2節を呼び出す非終端記号は、入力されるリストを分解し、1組ずつタイプ1、2節に渡して結果をマージする処理を再帰的に実行する。したがって、一般的な非終端記号の定義は(31)のようになる。

```
a([],Y) :- | Y = [].
a([X|Rest],Y) :- |
        a1(X,Y1),
        a2(X,Y2),
        merge(Y1,Y2,Y3),
        a(Rest,Y4),
        merge(Y3,Y4,Y).                    (31)
```

2) 痕跡のサーチ

痕跡は入力文中の単語と単語の間に存在すると考えられる。したがって、サーチは単語そのものか、または単語に直接呼び出されるカテゴリ（品詞）の処理の直後に行えばよい。単語の数は文法カテゴリの数に比べて非常に多くなるため、ここでは後者の方法をとる。すなわち、単語に直接呼び出されるカテゴリを文法規則の変換の際に選び、それらについては定義を(32)のように変更する。述語xgはスタックの状態を調べ、空でなければポップして、現在のストリームを入力としてそのカテゴリを呼び出し、そこからの出力を入力にマージして全体の出力とする述語である。ここでは予測は使わず、スタックにカテゴリがプッシュされていればポップしてしまう。具体的な定義は(33)から(35)のとおりである。

```
a([],Y) :- | Y = [].
a([X|Rest],Y) :- |
        a1(X,Y1),
        a2(X,Y2),
        merge(Y1,Y2,Y3),
        xg(Y3,Y4),
        a(Rest,Y5),
        merge(Y4,Y5,Y).                    (32)

xg([],Y) :- | Y = [].                      (33)
xg([[Stream,[]]|T],Y) :- |
    xg(T,Y1),Y = [[Stream,[]]|Y1].         (34)
xg([[Stream,x(Top,NewStack)]|T],Y) :- |
    nonterm(Top,[[Stream,NewStack]],Y1),
    xg(T,Y2),
    merge(Y1,Y2,Y).                        (35)
```

スタックが空の時は(34)でそのままコピーし、カテゴリがプッシュされていれば(35)によってポップし、ポップしたカテゴリを呼び出す。述語nontermはTopを述語名、あとの2つの引数を第1、第2引数として呼び出すためのメタ述語で、DEC-10 Prologの"=.."に相当する。

これらの定義からわかるように、タイプ1節、タイプ2節、xg述語はすべて並列に動作可能である。タイプ1、タイプ2どちらかの節からひとつでも結果が出力されれば、mergeプロセスは即座にxgに入力を与えることができる。xgはひとつでも入力が与えられれば処理を始めることができる。このように、左外置のために追加する処理を並列動作可能な形で記述することにより、並列処理を考えた場合、実行効率に与える影響は小さく抑えることが可能であると考えられる。

3)痕跡となるカテゴリのスタックへのプッシュ

(36)はスラッシュ・カテゴリを含む文法規則の例、(37)~(38)が変換されたPAX/XGコンパイルコードである。np1が終了した時点で、入力スタックとスラッシュ・カテゴリnpから新たなスタックを構成し、id11とともに次のプロセスに渡すことによってプッシュが行われる。srel自身は普通に記述されている。PAXのアルゴリズム上、srelが呼び出されるのはsrelの解析が終わった時点であるので、このような定義となる。

```
np --> np id11, srel../np.                 (36)

np1([X,InputStack],Y) :- |
    Y = [[[id11(X)],x(np,InputStack)]].     (37)

srel2([[id11(X)|T],InputStack],Y) :- |
    np([[X,InputStack]],Y1),
    srel2([T,InputStack],Y2),
    merge(Y1,Y2,Y).                        (38)
```

4) "<"、">"を含む規則の変換

(39)は"<"、">"を含む規則の例である。既に述べたように、"<"と">"で囲まれたカテゴリからは外へ語句が移動してはならない。このような処理を実現するためには、"<"、">"で囲まれたカテゴリを解析する時に、スタックを一時的に空にして解析を行い、そのカテゴリの解析が終了したらスタックを解析前の状態に戻してやれよい。このような処理をPAXで行うためには、"<"以前のスタックの状態を">"のあとまで運ぶ必要があるが、これは識別子に持たせてストリーム中に流せばよい。(39)の規則を変換したのが(40)、(41)である。

```
np --> np id12, < srel../np >.             (39)

np1([X,InputStack],Y) :- |
    Y = [[[id12(X,InputStack)],
          x(np,InputStack)]].               (40)

srel2([[id12(X,InputStack)|T],[]],Y) :- |
    np([[X,InputStack]],Y1),
    srel2([T,[]],Y2),
    merge(Y1,Y2,Y).                        (41)
```

4．4　PAX／XGの評価

第３章において用いた文法をXGSで書き換えたものおよび例文を用いて、実行効率について評価した。文法規則は8個減少して68個となるが、ストリームの構造が

変更され、非終端記号が再帰的に呼ばれるために、PAX/XGにコンパイルされた規則数は大幅に増加し、DCG版の275個から406個となった。

実験は、第３章で述べた並列度の動的解析ツールにより並列度と解析に要するレベル、さらに逐次実行の場合の解析時間について測定した。結果を表1-b、表2に示す。

システムが痕跡をサーチしながら解析を行うため、AXのようなボトムアップの構文解析アルゴリズムでは、４．２でも述べたように効率が悪化しがちである。この傾向は逐次実行の場合顕著であり、DCG版に比べ、平均で約2倍、ピークで4倍近くの時間を要している。痕跡のあるなしには関係なく、痕跡をスタックにプッシュするカテゴリ（名詞など）を多く持つ文で遅くなっている。効率の低下の原因としては、痕跡のスタックからのポップを無条件で行っていること、ストリームがスタックと組になり処理が複雑化していること、mergeを多用していることが考えられる。特に、逐次実行の際mergeは非効率的であり、逐次実行に特殊化するならば、差分リストを用いるなどの方法によってmergeは排除するべきであろう。

一方、並列処理の場合の実行時間に相当する、（解析に要する）レベルはわずかに増加するが、逆に減少しているものもある。このことから、並列実行ではDCGの場合のPAXとほぼ同等の時間で構文解析ができるものと期待される。

PAX/XGでは、行う処理が複雑になっているため、ゴールの呼び出し数の合計は増加しているが、４．４で述べたように、左外置処理のために追加した処理はすべて並列に実行できる。したがって、解析に要するレベルの増大は小さく抑えられる。レベルが減少するものの理由としては、ストリームの長さが減少することによってmergeのレベルが減少することが考えられる。PAX/XGでは、複数のストリームにひとつのスタックが組になったものが入力されると、それぞれのストリームに対してスタックが同一でなくなる可能性があるため、ストリームを分解してそれぞれをスタックと組にして出力している。このため、それぞれのストリームの長さは短くなり、mergeに要するレベルが減少する。以上のように、並列論理型言語の枠組みに、自然な形で左外置の処理を記述することによって、並列実行では、効率に対する影響が小さく抑えられることがわかった。

５．　おわりに

並列構文解析システムPAXに関して、並列度及び文法記述形式の拡張に関する考察を行った。並列度については、比較的小規模な文法（文法規則数76）についてPAXのプログラムにコンパイルし、並列論理型言語の動的解析ツールを作成して解析したところ、平均で約14弱の並列度があることが確かめられた。大規模な文法に対しては並列度はさらに大きくなると予想されるので、実用的な自然言語処理システムの構築を考えるなら並列処理は有効であると言える。しかし、並列度は、文の長さよりはむしろ、解析途上で使われる文法規則や、解析木の形、非終端記号の複雑さなどに影響され、定量的な傾向を知るには至らなかった。どのような要素にどの程度影響を受けているかについて、今後検討する必要がある。

文法記述形式に関しては、PAXにスタックを用いて痕跡に関する情報を制御する機能を付加し、文法をDCGから、左外置の処理を取り入れたXGSに拡張した。左外置の処理を純粋なボトムアップ構文解析に導入することは、効率の面で問題があると考えられてきた。実際、拡張以前のPAXと比較すると、逐次実行では相当の効率低下を記録している。しかし、逐次処理向きの最適化を行うことにより、効率改善が可能であると考えられる。本研究においては並列処理を指向して実現を行ったため、その点については検討していない。逐次処理環境で効率改善を計るならば、検討する必要があろう。

並列処理の場合は、並列論理型言語の枠組みに自然な形で左外置の処理を組み込むことにより、（解析に要する）レベルがDCGの場合とほぼ同等に抑えられることが実験的に確かめられた。プロセスの総数はかなり増大するが、その多くは並列に処理されるため、レベルに与える影響は小さい。このことから、並列処理環境ではあまり効率を落とさずに解析ができることが期待される。今後、実用規模の文法を用いて評価し、さらに改良することを考えている。

参考文献

[Aho 72] Aho, J.V. and Ullman J.D.: The Theory of Parsing, Translation and Compiling,Volume I, Parsing, Prentice-hall, 1972.

[Pereira 80] Pereira, F.C.N. and Warren, D.H.D.: Definite Clause Grammars-A Survey of the Formalism and a Comparison with Augumented Transition Networks,Artificial Intelligence. 13, pp.231-278, 1980.

[Pereira 81] Pereira, F.C.N.: Extraposition Grammars, American Journal of Computational Linguistics, vol.7, no.4, October-December, 1981.

[Ueda 85] Ueda, K.: Guarded Horn Clauses, ICOT Tech. Report TR-103, ICOT, 1985.

[尾内 84] 尾内理紀夫，清水　鋆，益田嘉直，麻生盛敏: 逐次型Prologプログラムの解析，Proc. of Logic Programming Conference '84, ICOT.,1984.

[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システム，コンピュータソフト

ウェア，Vol. 3，No. 2，1986.
[杉村 86] 杉村領一他：ロジックプログラミングをベースにした自然言語解析システムの比較，情報処理学会自然言語処理研究会，NL-57-2，1986.
[松本 86a] 松本裕治：並列構文解析，情報処理学会自然言語処理研究会，NL-53-2，1986.
[松本 86b] 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. 4，1986.

表1-a PAXの並列度解析結果

文No.	長さ	木の高さ	レベル	総ゴール数	並列度	最大プロセス数	
1	4	8	42	608	14.2	48	57
2	7	13	85	1652	19.4	60	77
3	11	13	121	1630	13.5	49	69
4	9	8	67	734	11.0	65	77
5	12	14	129	2253	17.5	61	78
6	7	7	55	524	9.5	38	48
7	7	11	69	1012	14.7	49	60
8	7	11	93	1703	18.3	65	88
9	5	8	62	1023	16.5	54	69
10	5	8	45	358	8.0	33	44
11	4	7	32	270	8.4	49	56
12	4	7	41	410	10.0	49	58
13	9	9	75	1518	20.2	57	69

表1-b PAX/XGの並列度解析結果

文No.	レベル	総ゴール数	並列度	最大プロセス数	
1	44	945	21.5	54	81
2	90	3173	35.3	127	175
3	120	3800	31.7	120	170
4	75	1340	17.9	64	100
5					
6	51	790	15.3	58	74
7	71	1400	19.7	57	79
8	80	2203	27.5	98	145
9					
10	54	1104	20.4	55	75
11	37	501	13.5	56	76
12	45	713	15.8	56	76
13	89	3046	34.2	139	194

表2 逐次実行における実行時間の比較（単位:msec.）
（10例のみ）

文No.	ＤＣＧ版	ＸＧＳ版
1	41.6	68.3
2	110.0	201.7
3	120.0	245.0
4	46.7	158.3
5	133.3	281.7
6	35.0	48.3
7	130.0	85.0
8	106.7	208.4
9	70.0	320.0
10	25.0	78.3

付録 A. 評価用文法

```
adjp —> adj.
adjp —> adj , adjp.
adjp —> adj , coconj , adj.

bep —> be_w.
bep —> be_w , not_w.

infinitive —> to_w , vp.
infinitive —> not_w , to_w , vp.
infinitive —> infinitive , coconj , infinitive.
infinitrel —> infinitive.
infinitrel —> p , np , infinitive.
infinitrel —> p , relpro , infinitive.

ncomp —> pp.
ncomp —> pred.
ncomp —> infinitrel.
ncomp —> srel.

nomhd —> n.
nomhd —> adjp , nomhd.
nomhd —> n , coconj , nomhd.

np —> det , nomhd , ncomp.
np —> det , nomhd.
np —> nomhd , ncomp.
np —> nomhd.
np —> pron , pp.
np —> pron.
np —> np , coconj , np.

obj —> np.
obj_one —> np.
obj_two —> np.

pp —> p , obj.
pp —> pp , coconj , pp.

pred —> adjp.
pred —> np.
pred —> pp.
pred —> pred , pp.
pred —> pred , infinitive.

s —> sdec.
s —> sdec , coconj , sdec.

sdec —> subj , vp.
sdec —> it_w , bep , adjp , infinitrel.
sdec —> there_w , bep , subj.

srel —> relpro , v.
srel —> relpro , v , obj.
srel —> relpro , v , obj_one , obj_two.
srel —> relpro , v , infinitive.
srel —> relpro , v , obj , infinitive.
srel —> relpro , bep , pred.
srel —> subj , v.
srel —> p , relpro , sdec.

srel —> relpro , subj , v.
srel —> srel , pp.

subclause —> subconj , sdec.

subj —> np.

that_clause —> subconj , sdec.

vp —> v.
vp —> v , obj.
vp —> v , infinitive.
vp —> v , obj_one , obj_two.
vp —> v , obj , infinitive.
vp —> vp , pp.
vp —> vp , coconj , vp.

v —> verb.
v —> verb , coconj , verb.

adjp —> qdet , adj.
s —> sq.
s —> swhq.

swhq —> whnp , sq.

sq —> modalp , subj , vp.
sdec —> subj , bep , pred.

auxd —> modalp.

modalp —> bep , [able], [to].
modalp —> modal.

srel —> relpro , vp.

vp —> vp , advp.

advp —> adv.

adv —> by_w , refl.

np —> ddet.
det —> ddet.

n —> [computer], [network], [system].
```

付録 B. 例文

1. I open the door.
2. I open the door with a key.
3. The books that are on the table are difficult to read.
4. The computer network system is a very large system.
5. This paper presents an explanatory overview of a large and complex grammar.
6. He was able to walk by himself.
7. There was a man who walked around.
8. He is the man who loves her.
9. Can you see that picture ?
10. What should I do then ?
11. There was a man.
12. He is a man.
13. I open the door and she closes the window.