自然言語処理　５３－２
（１９８６．１．３１）

並列構文解析

松本裕治＊
（電子技術総合研究所　パターン情報部）

１．はじめに

最近の計算言語学の流れを顧みると、自然言語処理のための文法記述形式として文脈自由文法が見直されている。自然言語が文脈自由文法によって記述可能かどうかということについては議論の分かれるところであるが、言語学の最近の流れの中に自然言語の構文構造の記述法として文脈自由文法、もしくは、文脈自由文法に機能拡張を施したものを設定しようという動き [Pullum 82] [Gazdar 82] [Kaplan 82] があるのは確かである。

PereiraおよびWarrenによって提案されたＤＣＧ（Definite Clause Grammars） [Pereira 80] はホーン節論理を基礎とし、単一化操作とPrologのプログラムによって補強された文脈自由文法であると考えることができる。ＤＣＧは構文解析のプログラムとしてだけでなく、自然言語の文法記述言語としてかなりの能力を備えた記述形式である。本論文では、ＤＣＧを文法記述法と仮定し、それを高速かつ並列に実行する構文解析システムを提案する。基本的にはＤＣＧで書かれた文法規則や辞書項目を並列論理型言語のプログラムに変換し、それをいずれ実現されるであろう並列マシン上で実行するという形を取る。本論文では、そのような論理型言語としてParlog [Clark 84] を採用するが、考え方は容易にＧＨＣ（Guarded Horn Clauses）やConcurrent Prologなどの他の並列論理型言語へも応用可能である。現段階のシステムは、Parlog処理系 [Gregory 84] および Prologの上で実験を行っている。

自然言語の構文解析の手法の他の流れとして決定的構文解析に関する話題も確かにある [Marcus 80] が、我々は非決定的な構文解析を前提とすることにする。これは、ＤＣＧを文法の記述言語として採用したこともその理由の一つであるが、個々の文法規則を宣言的にも手続的にも互いに独立に記述できるようにしたかったからである。ただし、本システムでは、基本的に上昇型（bottom-up）の戦略を用いているため、文法規則の記述法に多少の制限が加えられることがある。本論文では、我々の並列構文解析の手法を紹介することを目的とし、文法記述の方法論やその他のユーティリティについては別の機会に紹介する。

次節以下の概略を述べると次のようになる。まず、第２節では、我々のシステムの基本となるアルゴリズムについて説明する。この段階のアルゴリズムは、単純な後戻り解析を行い、効率的にも難がある。第３節および第４節では、この方法の効率改善を行うと共にその考え方が並列処理に容易に応用可能であることを説明する。

第５節では、本システムの評価を行うために、Chart Parsing [Kay 80] との比較を行う。本システムの動きが上昇型のChart Parserと本質的に同じであることを示す。従って、本システムは、効率的には Chart Parserに劣らず、しかも論理プログラミング言語にコンパイルされた形で、副作用なしに並列に動作可能という特徴を持っている。

最後の節では、残された問題点やその解決法に対する考察をおこない、今後の可能性を探ることにする。

２．基本的アルゴリズム

本節以下の数節では、純粋の文脈自由文法を対象とする。文法規則の非終端記号が引数を持つことができたり、文法規則内に Prologのプログラムが記入できるというＤＣＧの特徴を導入する話題については最後の節で簡単に触れることにし、本論文の対象外とする。また、取り扱う文法規則に対する制約として次の条件を置かねばならない。まず、文法には空系列を生成するような規則を含んでいてはならない。次に、周期的（cyclic）な規則の集合を含んでいてはならない。周期的な規則の集合とは、ある非終端記号からの記号系列の生成過程において、他の記号を生成することなしに自分自身の生成に至るような一連の規則のことである。これは、左再帰的な規則の集合の存在よりは弱い制限である。これらの制限は、上昇型戦略を採用したことによるが、一般に下降型(Top-down）戦略では許されない左再帰的規則は、それが周期的規則でない限り、含まれていてもかまわない。

我々の方法は基本的には左隅構文解析法（Left-Corner Parsing） [Aho 72] である。解析は上昇型に行われるが、一度ある非終端記号が得られると、与えられた文法規則を用いてその記号を最も左端に持つ構文解析木が得られないかを調べるか、もしくは今までに得られていた部分的構文解析木の中で自分自身によってより完全に近い解析木に拡張可能なものがあるかを調べる。この処理はあらゆる可能性が尽くされるまで続けられる。我々は、以前、ＤＣＧで記述された文法にこのような手続的意味を与えるシステムの開発をおこなった。このシステムはＢＵＰと呼ばれている [Matsumoto 83,84] 。

本節では、ＤＣＧによって記述された文法に同様の手続的意味を与える別の方法を提案する。一般に、文は左から右に、かつ下から上へ、解析される。文中の単語や解析途中で得られた非終端記号は、もしそれがある文法規則の右辺中の最も左の要素と一致すると、その文法規則を用いて新しい非終端記号を作ろうとする。より正確には、もしその文法規則が右辺にその他の要素を持っているとすると、それらの記号はそれ以後に解析される要素として予測されることになる。また、得られた単語や非終端記号がすでに

＊現所属　（財）新世代コンピュータ技術開発機構　第一研究室

その場所で予測されているのであれば、それらはそのような予測を埋めるためにも用いられる。このような基本的な解析法を仮定して、これを実現するためにどのような仕組みが必要であるかを考えてみよう。

左隅解析法の処理の経緯を記録するためには、ある時点までに使われている文法規則を記録するだけでなく、それぞれの規則のどの部分までが解析に使われているかを知っていなければならない。これには、よく行われているように文法規則の右辺の中に今まで解析に使われた部分と使われていない部分を区切るマークを付けてもよいが、より簡単には個々の文法規則の右辺内のそれぞれの位置に、他の位置と区別をするための識別子をあたえることによっても可能である。例えばすべての単語と非終端記号がPrologの述語として定義され、それぞれが二つの引数を持つと仮定しよう。一つは文中の自分より左からの情報を受け取るための引数、もう一つは自分の右へ情報を流すための引数であるとする。これらの引数によってやり取りされるデータは構文解析途中における履歴を表わし、実は上で述べた識別子の系列として表現可能である。以上の考察により次のような文法規則の変換によって上述の解析法を実現することができるのが分かる。すなわち、（1）の文法規則を（2）のようなPrologプログラムへ変換することを考える。（1）の文法規則中、idという記号を先頭に持つ数字は規則内の位置を同定する識別子である。

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

```
(2) b(X,id1(X)).
    c(id1(X),id2(X)).
    d(id2(X),Y) :- a(X,Y).
```

この例によると、Prologのあるゴールが識別子 'id1' を先頭要素とするデータを受け取ったとすると、それは非終端記号 'b' が既に見つかり文法規則（1）の id1 の部分までの解析が終了したことを意味する。

より単純な（3）のような文法規則は（4）のような Prologの節に変換される。

```
(3) vp --> verb.
```

```
(4) verb(X,Y) :- vp(X,Y).
```

終端記号をふくむ例は挙げなかったが、終端記号と非終端記号は同様に扱われ、まったく区別する必要がない。

次に示す文法規則の例（5）は、（6）のような Prologのプログラムに変換される。文法規則中の数字は識別子である。

```
(5) s --> np, id1 vp.
    np --> det, id2 noun.
    np --> det, id3 noun, id4 relc.
    relc --> [that], id5 s.
    vp --> verb.
    vp --> verb, id6 np.
    det --> [the].
    noun --> [man].
    noun --> [woman].
    verb --> [loves].
    verb --> [walks].
```

```
(6) np(X,id1(X)).
    vp(id1(X),Y) :- s(X,Y).
    det(X,id2(X)).
    noun(id2(X),Y) :- np(X,Y).
    det(X,id3(X)).
    noun(id3(X),id4(X)).
    relc(id4(X),Y) :- np(X,Y).
    that(X,id5(X)).
    s(id5(X),Y) :- relc(X,Y).
    verb(X,Y) :- vp(X,Y).
    verb(X,id6(X)).
    np(id6(X),Y) :- vp(X,Y).
    the(X,Y) :- det(X,Y).
    man(X,Y) :- noun(X,Y).
    woman(X,Y) :- noun(X,Y).
    loves(X,Y) :- verb(X,Y).
    walks(X,Y) :- verb(X,Y).
```

さて、（6）に示すプログラムは、それだけで文脈自由文法（5）のための後戻り構文解析プログラムになっている。例えば，文、"the man walks."を解析したい場合には、（7）に示すような Prologのゴール文を呼べばよい。

```
(7) the(begin,X),man(X,Y),walks(Y,end).
```

ゴール文中の 'begin', 'end'という項は、文の先頭と末尾を指定する識別子である。連続する二つの単語が共有変数を持っていることに注意してほしい。これらは解析の途中結果を左から右に渡すための通信チャネルとして働く。プログラムを完全なものにするためには、上の定義以外に次の節が必要である。

```
(8) s(X,Y) :- X==begin,Y==end.
```

これは、's'が呼ばれた時に、その文が与えられた単語すべてから構成されているかどうかを確かめることに相当する。つまり、構文解析の終了条件を示している。

図1に（7）で示した Prologプログラムの実行の過程を示す。図中、直線の矢印は、あるゴールが他のゴールに呼び出されたことを、二重線の矢印は、他のゴールの単一化によって自分自身の引数の値が変化したことを示している。この Prologプログラムは、バックトラッキングによって、与えられた文に対するあらゆる可能な解析を行う。

3．並列構文解析

本節では、前節で示した手法を改良することを考える。

前節のプログラムをよく観察すると、二種類の節があることがわかる。すなわち、頭部の述語の第一引数に変数を持つ節と、具体的な項を持つ節である。以後、これらの節を、タイプ１節およびタイプ２節と呼ぶことにしよう。それぞれの述語に含まれる引数は、第一引数が左からの情報の受け取り口として、第二引数が右への情報の送り手として働く。従って、タイプ１節の第一引数が変数であるということは、このような節は、左から送られて来る情報とは無関係に自分自身の動作を行うことを意味する。実際、タイプ１節は左からの情報をとは無関係に、自分自身を左端の子とする木構造を作ろうとする。（６）の一番最初の節がこの例であり、受け取ったデータ（識別子の系列）の先頭に識別子 'id1'を付けて次のプロセスに渡す。

タイプ２節は受け取ったデータの先頭の要素を自身の第一引数の値と比較する。もしそれらが等しければ、その節の本体部の記述に従って、新しい非終端記号の呼出しを行なったり、新しいデータ構造を作り出したりする。（６）の第二番目の節はその例で、もし受け取ったリストの先頭の要素が 'id1'であれば、非終端記号 's'に対応する呼出しを行う。 's'の呼出しとともにこの識別子はリストの先頭から削除されて 's'に渡される。これはつまり、この識別子が、これ以後の処理にとってもはや必要ないからである。（６）の第六番目の節は第二引数にもデータを含んでいる。これは、もとの文法規則では（５）の第三番目の規則の中央に現れる規則に相当し、この文法規則の 'id3'の位置から 'id4'の位置へ処理が進んだことを示している。したがって、 'id3'を受け取ったデータから取り除き、'id4'を付け加えて渡すだけで、その本体部では何も行なわれない。

このアルゴリズムは、それ自体ではあまり効率のよいものではない。後戻りを利用して探索を行なうため、入力文長に対して指数時間の計算複雑さをもつ。その原因の一つは、タイプ１節がそれまでの解析の過程に関係なく一定の処理を行なうためである。また、複数のタイプ１節が後戻りを利用して処理されるのも効率に影響を与えている。この無駄を省くためには、タイプ１節に受け取られる情報を集合として一括して扱い、タイプ１節によって行なわれる処理を繰り返させないようにするのが自然である。タイプ１節が行なう処理はそれまでの処理の経緯に関係ないから、各識別子はその集合の一つずつの要素ではなく、集合そのものに作用するように定義する必要がある。タイプ２節もこの考え方に合うように定義しなおさねばならない。また、タイプ１とタイプ２の節の行なう処理は互いに独立に実行可能なので、並列に行なうことができる。

この考察によるプログラムの修正は、実は並列論理型言語に非常に相性がよく、自然な形で実現することが可能である。ParlogやＧＨＣのような並列論理言語での習慣に従って、集合をストリームとしてリストを用いて表現することにしよう。例えば、（９）のようなタイプ１節の集合は（１０）のような Parlogの節にまとめられ、（１１）の

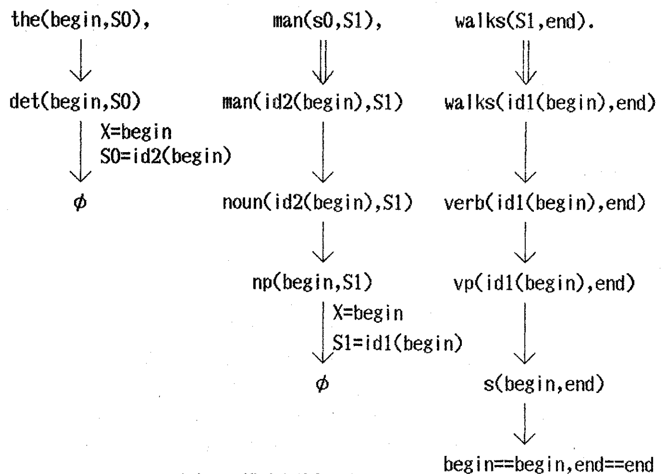

図１．構文解析の実行例

ようなタイプ２節の集合は（１２）のように定義しなおされる。

```
(9)  det(X,id2(X)).
     det(X,id3(X)).

(10) mode det1(?,^).
     det1(X,[id2(X),id3(X)]).

(11) noun(id2(X),Y) :- np(X,Y).
     noun(id3(X),id4(X)).

(12) mode noun2(?,^).
     noun2([],[]).
     noun2([id2(X)|T],Y) :-
          np(X,Y1), noun2(T,Y2),
          merge(Y1,Y2,Y).
     noun2([id3(X)|T],[id4(X)|Y]) :-
          noun2(T,Y);
     noun2([_|T],Y) :-
          noun2(T,Y).
```

モード指定によって、各節の第一引数が入力引数、第二引数が出力引数であることが示されている。ただし、入力として与えられる第一引数の内容は、もはや単なる識別子の系列ではないことに注意する必要がある。この引数に与えられるのはストリームであり、それぞれの要素は別のストリームを取り込んだ識別子という再起的な構造をしている。

タイプ１節は、その節に対応する非終端記号を基にした左隅解析を始める。具体的には、右辺の左端の要素が自分自身に等しいすべての文法規則について、その要素の直後に割り当てられた識別子を集めストリームとして送る。（１０）はその例である。タイプ１節は第１引数が変数であるような節と定義したが、そのような節は（９）のようなものだけではなく、第２引数も変数のものもあり得る。

例えば、（１３）のような節の集合は（１４）のように変換される。

```
（１３）verb(X,Y) :- vp(X,Y).
        verb(X,id6(X)).
```

```
（１４）mode verb1(?,^).
        verb1(X,[id6(X)|Y]) :- vp(X,Y).
```

もし、このような節が２つ以上ある場合には、それらの出力は１本のストリームにマージされる。

タイプ２節はｏｒプロセスの集合として定義される。（１２）において、第２、第３の節がそれぞれ（１１）の２つの節に対応し、ストリームの先頭の要素がそれぞれの担当する要素に等しい場合に、定められた処理を行なう。タイプ２節が行なう処理は、タイプ１節とは異なり、不完全な解析木を完成させたり、より完全に近い構造にしたりする。例えば、第２番目の節は、ストリームの先頭が 'id2'で始まるデータであるとき、名詞句に対応するプロセスを生み出し、同時にストリームの残りの要素について自分と同じプロセスを呼び出している。'id2'を受け取ったということは、（５）の２番目の文法規則によって解析が途中まで進んだことを意味し、このプロセスが行なっていることは、名詞によって、この部分的な構造を完成した名詞句に造り上げることに対応している。第３番目の節では、プロセスを生み出す代わりに新らしい識別子が出力ストリームに与えられている。これは、（５）の３番目の規則において、処理が 'id3'の位置から 'id4'の位置まで進んだことを意味する。また、第１番目の節は、ストリームが空になった時の処理を示し、第４番目の節は、他のどの節にも先頭の識別子が受け付けられなかった場合にのみ起動され、その識別子を捨てる働きをする。第３番目の節とこの節がピリオドではなくセミコロンで区切られていることに注意されたい。これは、Parlogにおいて、逐次的ｏｒを表わすデリミタである。

タイプ１およびタイプ２の節は互いに独立に動作することが可能であるので、並列に動かせることができる。したがって、一般に、非終端記号は次のように定義される。

```
（１５）mode noun(?,^).
        noun(X,Y) :-
            noun1(X,Y1), noun2(X,Y2),
            merge(Y1,Y2,Y).
```

このように、並列に動きうるプロセスをａｎｄ並列として定義し、それらの出力をマージすることによって並列性を引き出すことができる。一方、これらの節の定義を見てもわかるように、以上のプログラムは、Parlogだけでなく、Prologにおいても実行可能である。Prologにおいては、マージは非効率的な演算であり、差分リストによって代用するのがよい。差分リストとは、二つのリストの差によって一つのリストを表現する方法である。（１６）から（１９）で示される節は、（１０）、（１２）、（１３）および（１４）を差分リストを用いて書き直したものである。それぞれの述語の第２、第３引数が、差分リストを構成している。

```
（１６）det1(X,[id2(X),id3(X)|Yt],Yt).
```

```
（１７）noun2([],Y,Y) :- !.
        noun2([id2(X)|T],Y,Yt) :- !,
            np(X,Y,Y1), noun2(T,Y1,Yt).
        noun2([id3(X)|T],[id4(X)|Y],Yt) :- !,
            noun2(T,Y,Yt).
        noun2([_|T],Y,Yt) :- !,noun2(T,Y,Yt).
```

```
（１８）verb1(X,[id6(X)|Y],Yt) :- !,vp(X,Y,Yt).
```

```
（１９）noun(X,Y,Yt) :- !,
            noun1(X,Y,Y1),noun2(X,Y1,Yt).
```

上のプログラムでは、差分リストは常に出力変数として扱われ、本体部に現れる非終端記号の出力をまとめて頭部の出力とするように働くので、簡単な省略形によって、プログラムをより簡潔に記述することができる。実は、ＤＣＧの表現法をそのまま使うのが最も簡単な方法である。（１６）から（１９）の節をＤＣＧを用いて書き直したものが（１６）'から（１９）'で示す節である。

```
（１６）' det1(X) --> !,[id2(X),id3(X)].
```

```
（１７）' noun2([]) --> !.
          noun2([id2(X)|T]) --> !,
              np(X),noun2(T).
          noun2([id3(X)|T]) -->
              [id4(X)],!,noun2(T).
          noun2([_|T]) --> !,noun2(T).
```

```
（１８）' verb1(X) --> [id6(X)],!,vp(X).
```

```
（１９）' noun(X) --> !,noun1(X),noun2(X).
```

本節で示したプログラムの修正に伴って、文を表わす非終端記号 's'および入力文を解析するための初期呼び出しを修正しなければならない。一つの方法は初期呼び出しを（２０）のように定義し直し、's2'の定義を Parlogの場合には（２１）のような、Prologの場合には（２２）のような定義を追加すればよい。

```
（２０）the([begin],X),man(X,Y),walks(Y,Z),
        fin(Z).
```

```
（２１）s2([begin],[end]).
```

```
（２２）s2([begin]) --> [end],!.
```

文が完成すると 's2'が呼ばれるが、's2'は、[begin]を受け取ると [end]を出力として次のプロセスへ送る。プロセスが識別子 [begin]を受け取るのは、そのプロセスに相当する非終端記号が入力文の文頭から発見されたことを意味し、's'は目的としていた非終端記号であるから、出力として渡される識別子 [end]は、文末を意味することになる。'fin'がこの識別子を受け取るとすると、それは入力文の単語すべてを用いて文が構成されたことになる。'fin'はユーザによって定義される述語で、構文解析の結果、ユーザが行ないたい処理を記述することになる。

４．トップダウン予測

前節までで紹介した構文解析法は、ボトムアップ解析法であり、トップダウン的な情報は何一つ使っていなかった。本節では、プログラムの簡単な変更によって本アルゴリズムを、トップダウン予測を行なうアルゴリズムに改良し、より効率よいものに変えることができることを示す。ここでいうトップダウン予測とは、ＬＩＮＧＯＬ[Pratt 75]でいえばオラクル、Chart Parsing [Kay 80] では到達可能性と呼ばれているものである。

例として、（５）の第２番目の文法規則を考えて見よう。この規則は、名詞句の可能な構造として限定詞と名詞の並びがあることを示すものである。構文解析の処理中に限定詞が発見されたとしよう。この文法規則を解析のために使用することは、この規則に割り当てられた識別子を送ることに対応する。ここで注意しなければならないことが二点ある。一つは、この規則を使うのであれば、いま発見された限定詞の直後に名詞を予測すべきであることである。名詞の構成を奨励することによって、この規則の完成を促進することができる筈である。このような場合、名詞はこの場所におけるトップダウン予測と呼ばれる。もう一つの注意すべき点は、新らしい文法規則を用いる場合には、その規則が最終的に作り出す非終端記号が少なくとも一つのトップダウン予測によって予測されていなければならないことである。今の例では、文法規則の左辺が名詞句であるから、限定詞が発見されたときにこの規則を使うためには、名詞句もしくは名詞句を左端の子孫に持ち得るような非終端記号が予測されていなければならない。もし、そのような非終端記号が一つも存在しないならば、この規則によって得られる名詞句は、つながるべき場所がなく、全く無駄な処理を行なってしまうことになる。このように、トップダウン予測は、無駄なボトムアップ解析を避けるために非常に有効である。非終端記号が他のどの非終端記号の左端の子孫になり得るかは、文法規則の集合が与えられた時点で事前に計算可能である。非終端記号間のこの関係をリンク関係と呼ぼう。任意の文法規則において、右辺の左端の非終端記号と左辺の非終端記号の間にはリンク関係がある。これらの基本的なリンク関係の反射的かつ推移的な閉包が求めるべきリンク関係である。

この考え方を我々の構文解析法に取り込むには幾つかの方法が考えられるが、ここでは、我々が現在取り入れている方法を紹介する。非終端記号が受け取るストリームは、識別子に冠されたストリームよりなるストリームである。識別子は、ある特定の文法規則のある特定の位置に与えられたものであるから、それに続く非終端記号は一意に定まる。この非終端記号こそが、この識別子を含むストリームが生成された位置における一つのトップダウン予測である。識別子と非終端記号の間の対応は文法規則をプログラムに変換する際に得ることができる。また、特殊な識別子である 'begin'は 's'、すなわち文を予測し、'end'は何も予測しない。

トップダウン予測は、タイプ１節の定義を一部修正することによって実現できる。第２節で示した定義では、タイプ１節は、文法規則の右辺の左端の非終端記号に対応し、識別子を一つ生成するように定義されていた。トップダウン予測を利用するためには、受け取ったデータの先頭の識別子の予測する非終端記号が、元の文法規則の左辺の非終端記号とリンク関係を持っていないときに、その実行を行なわないようにすればよい。並列処理用のタイプ１節は、ストリームを受け取り、一般に複数の識別子を出力するが、その内のある特定の識別子について見れば、その識別子が出力されないのは、その識別子に対応する元の文法規則の左辺の非終端記号が入力ストリーム内のどの識別子によっての予測されない場合である。逆に言うと、入力ストリーム中に、左辺の非終端記号を予測する識別子が一つでもあれば、その文法規則に対応する識別子を出力に与えてもよいということになる。

現在のシステムでは、トップダウン予測は、ストリーム内の要素のフィルタリングを行なうプロセスとして実現されている。つまり、このプロセスは、タイプ１節へ与えられるべきストリームを取り込み、不要な要素を取り除く。（１０）で示したタイプ１節の定義は（２３）のようになる。

（２３）
```
det1(X,Y) :-
        tp_check(X,np,New_X),
        tp_output(New_X,[id2(New_X),id3(New_X)],Y).
```

この定義において、'tp_check'がフィルタリングを行なうプロセスで、入力ストリーム 'X'内で、名詞句につながり得るものだけを 'New_X'に通す働きをする。'tp_output'は、'New_X'が空のときには、空リストを、'New_X'が空でないときには、第２引数で示されるリストを 'Y'に出力する。これら二つのプロセスが並列に動作可能なことに注意されたい。特に、'tp_output'は、'New_X'の完成を待つ必要はなく、一つでも要素が 'New_X'に与えられるやいなや 'Y'への出力を開始することができる。この例でいえば、'New_X'に何らかの出力が与えられるとすぐに、'id2'、'id3'、二つの識別子が送り出される。このとき、これらの識別子が引き連れている 'New_X'の値は完成されているとは限らず、'tp_check'によって残りの部分の計算が続けられていてもよい。'id2'と 'id3'が両方とも同一のプロセスによって扱われているのは、どちらの識別子も同じように名詞句を構成する規則より派生されたものであったためで、異なる非終端記号へつながるものが存在する場合には、それぞれに対してフィルタが用意され、やはり独立に動く

ことができる。

５．Chart Parsingとの比較

我々のシステムの評価を行なうため、本節では、Chart Parsing[Kay 80]との比較を行なう。Chart Parsingは、チャートと呼ばれるデータ構造を作り出すプロセスより成る。チャートとは、言わば、他のアルゴリズムでもよく行なわれるように、解析の途中結果を保存するための表の働きをするものであるが、特に概念的には、グラフとして表わされる。チャートの節点は、入力文の各単語の切れ目を表わし、弧は、部分的もしくは完全な構文構造を表わす項をラベルとして持ち、それが覆う範囲の単語列がその項で示された構造を持つことを表現する。部分的に完成された構造を表わす項は、空スロットを持ち、完全な構造に対する項は、空スロットを持たない。これらの項は、チャートがグラフ表現されるときには、それぞれ、活性弧(active edge)、および、不活性弧(inactive edge)と呼ばれる。チャート法は、その著者も言うように、実際は、アルゴリズムではなく、アルゴリズムの枠組み(algorithm schema)である。チャートの弧を作り上げていく過程は、その枠組みに与えられるコントロールに依存する。ここでは、上昇型のコントロールをもつチャート法と、我々のアルゴリズムとの類似性を指摘することにする。上昇型のコントロールをもつチャートは、二つの規則に従って動作する。一つは、不活性弧を基にして、文法規則の中でその弧に対応する非終端記号を右辺の左端にもつ規則より新らしい活性弧を作る。もう一つの規則は、活性弧の空スロットに、不活性弧を取り込む処理である。

我々のアルゴリズムと上昇型チャート法には密接な関係がある。チャート法における二つの規則が、我々の方法では、それぞれタイプ１節およびタイプ２節の役割に対応している。不活性弧は、非終端記号の呼び出しに対応し、活性弧の働きは、非終端記号間を渡る識別子によって行なわれる。

両者の間の最も重要な違いは、我々の方法では、アルゴリズムが Parlog、もしくは、Prologに完全にコンパイルされている点である。チャートでは、二つの弧が接しているかどうかは、弧の開始場所(location)および長さ(length)を用いて調べられる。一般に、チャートは、これらの情報と共に項の値を格納した表として表わされる。我々のアルゴリズムでは、このような表は全く必要ない。弧の開始場所に相当する情報は、初期呼び出しにおける共有変数によって取って代わられているし、項に関する情報は、全てストリーム内の識別子によって表現されている。識別子が自分の中にストリームを持ち込んでいるによって、新らしい弧についての場所や長さの計算が不要になっている。また、チャート法と同様に、同じ文法規則が複数個あるのでなければ、同じ構造が繰り返し作られることもない。図２は、（２４）の文法を用いて上昇型コントロールによって作られるチャートの、表としての表現を示すものである［Kay 80］。項内の疑問符は、活性弧の不完全な部分に対応する。

```
（２４）s --> np,vp.
        np --> a,n.
        np --> prp,n.
        vp --> v,a.
        vp --> v,av.
        a --> [failing].
        a --> [hard].
        n --> [students].
        v --> [looked].
        av --> [hard].
```

この文法から変換により得られる Parlogプログラムの主要部分を示したものが（２５）である。

```
（２５）np1(X,[id1(X)]).
        a1(X,[id2(X)]).
        prp1(X,[id3(X)]).
        v1(X,[id4(X),id5(X)]).

        vp2([id1(X)|T],Y) :-
            a(X,Y1),vp2(T,Y2),
            merge(Y1,Y2,Y).
        n2([id2(X)|T],Y) :-
            np(X,Y1),n2(T,Y2),
            merge(Y1,Y2,Y).
        n2([id3(X)|T],Y) :-
```

	場所	長さ	項
1	0	1	[failing]a
2	0	1	[failing]prp
3	1	1	[students]n
4	2	1	[looked]v
5	3	1	[hard]a
6	3	1	[hard]av
7	0	1	[[failing]a [?]n]np
8	0	1	[[failing]prp [?]n]np
9	0	2	[[failing]a [students]n]np
10	0	2	[[failing]prp [students]n]np
11	0	2	[[[failing]a [students]n]np [?]vp]s
12	0	2	[[[failing]prp [students]n]np [?]vp]s
13	2	1	[[looked]v [?]a]vp
14	2	1	[[looked]v [?]av]vp
15	2	2	[[looked]v [hard]a]vp
16	2	2	[[looked]v [hard]av]vp
17	0	4	[[[failing]a [students]n]np [[looked]v [hard]a]vp]s
18	0	4	[[[failing]prp [students]n]np [[looked]v [hard]a]vp]s
19	0	4	[[[failing]a [students]n]np [[looked]v [hard]av]vp]s
20	0	4	[[[failing]prp [students]n]np [[looked]v [hard]av]vp]s

図２．チャートの例

```
      np(X,Y1),n2(T,Y2),
      merge(Y1,Y2,Y).
a2([id4(X)|T],Y) :-
    vp(X,Y1),a2(T,Y2),merge(Y1,Y2,Y).
av2([id5(X)|T],Y) :-
    vp(X,Y1),av2(T,Y2),merge(Y1,Y2,Y).

failing(X,Y) :-
    a(X,Y1),prp(X,Y2),merge(Y1,Y2,Y).
hard(X,Y) :- a(X,Y1),av(X,Y2),
    merge(Y1,Y2,Y).
students(X,Y) :- n(X,Y).
looked(X,Y) :- v(X,Y).
```

入力文 “failing students looked hard.”を解析するための初期呼び出しは、（２６）のようになる。

```
（２６）failing([begin],D1),students(D1,D2),
        looked(D2,D3),hard(D3,D4),fin(D4).
```

このプログラムの実行の様子を、図３に示す。図中、矢印は、述語の呼び出し、もしくは、ストリームの流れを表わす。矢印に付けられた数字は、本システムの処理の流れと上昇型チャート法における処理との間の対応を示している。数字の添えられていない矢印は、非終端記号からのタイプ１節およびタイプ２節の生成に対応し、計算オーダーには影響を与えない。入力文の各単語から始まる処理は、どれも独立に行ない得ることに注意されたい。プロセスが他のプロセスに影響されるのは、そのプロセスがストリーム内のデータを参照しようとするときだけで、それ以外の処理は全て並列に行なってもよい。例えば、'looked'から始まる処理は、たとえ 'D2'が変数のままであっても続けることができる。図中、垂直方向は、時間軸に対応し、左右に水平に配置されたプロセスは、並列に動作可能である。この図より、本アルゴリズムの計算時間は、解析木の高さにほぼ比例することが分かる。すなわち、本アルゴリズムを使えば、文脈自由文法による構文解析が、高々、入力文の長さに比例する時間で行なえることになる。

我々のアルゴリズムは、以上で分かるように、チャート法に比べて、時間的に劣らないうえ、論理型プログラミング言語により実現されていることから、変数の値が共有されているため、チャートより少ない空間しか必要としない。新らしい項が得られる度に、チャートでは部分項のコピーが行なわれるが、我々の方法では、それが行なわれない。

６．結論

並列論理プログラミング言語による並列構文解析法について述べた。ここで説明したプログラムは、Parlogのみならず、Concurrent PrologやＧＨＣなど、Relational Lnaguage [Clark 81] より派生した他の多くの並列論理型言語で実現することができる。第３節で述べたように、本システムの Prolog による実現も極めて実用的である。実際に、DEC-10 Prolog上での実験では、著者らのＢＵＰに比べて、一桁に近い実行効率を得ている。

ここで考えておかなければならない問題が二点ある。本論では、文脈自由文法のみについて論じ、ＤＣＧ規則の取り扱いについては説明しなかった。ＤＣＧ規則では、非終端記号は任意数の引数を持つことができるし、右辺には、任意の Prologのプログラムを記述することが許されている。このような Prolog プログラムを補強項と呼ぼう。

補強項を本システムで扱うのは、さほど難しくはない。だたし、補強項によって記述されるプログラムが、Prologではなく Parlogなどの並列型言語で記述されなければならないという制限が止むなく課されてしまう。ＤＣＧ規則を本システムのプログラムに変換するときには、原則として右辺の一つの非終端記号に対して一つの節が作り出される。右辺に現れる補強項は、その文法規則中、直前に現れる非終端記号に対応するものとして生成された節のガード部に置かれるように変換される。本システムは上昇型の手続的意味を持っているから、右辺の左端の非終端記号より左側に現れる補強項は、意味をなさない。変換時には、そのような補強項は、左端の非終端記号に対応する節のガード部に置かれる。第３節で示した変換では、同一非終端記号を持つタイプ１節は一つにまとめられていたが、異なる補強項を持つタイプ１節は単純にまとめることができない。このようなタイプ１節は別々に定義され、それらの出力は後でマージされ

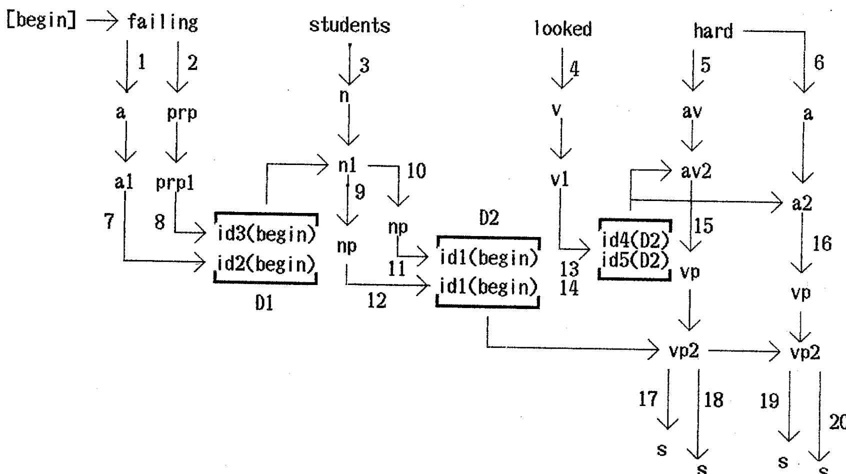

図３．並列構文解析の実行例

る。最初に述べたように、補強項は並列論理型言語で記述されなければならないが、Prologによる実現では、補強項は決定的な Prologプログラムでなければならない。ここで言う決定的とは、解が二つ以上ないという意味である。これは、補強項がガードとして使われることから来る制限である。

もう一つの問題は、非終端記号が引数を持ち、その中に変数が含まれる場合である。このような場合には、同一変数が二つ以上のストリームに流されることがあり得る。ストリーム中を流れるデータは、トリー状の構造をしており、異なる枝にあるデータは、異なる環境に属する。異なる環境内で変数を共有することは可能であるが、一方への値の代入が、他方をも書き換えることがあってはならない。この問題に対する最も簡単な解法は、変数が異なる環境に渡される度に、コピーすることである。しかし、そのような単純な取り扱いをしていたのでは、記憶領域の爆発につながる。より穏やかな方法は、タイプ2節のガード部が成功したときにのみその引数に含まれている変数をコピーすることである。現在 Prolog上で作成中のシステムは、このような考えをもとに開発中である。

［謝辞］

本システムの原形は、著者がロンドン大学インペリアルカレッジに留学中に行なわれた。著者は、帰国後、電総研から新世代コンピュータ技術開発機構へ出向したが、その間に、いろいろ手を加えて現在のシステムにまで発展した。著者の英国留学を援助していただいたブリティッシュカウンシルを始め、以上の研究機関やその他様々な方から助言や励ましの言葉を頂いた。ここに感謝の意を表する次第である。

References

[Aho 72] Aho,A.V. and Ullman,J.D.,:The Theory of Parsing, Translation, and Compiling, vol.1, Parsing, Prentice-Hall, 1972.

[Clark 81] Clark,K.L. and Gregory, S.,"A Relational Language for Parallel Programming," Imperial College Research Report DOC 81/16, July,1981.

[Clark 84] Clark,K. and Gregory,S.,"PARLOG: Parallel Programming in Logic," Imperial College Research Report DOC 84/4, April,1984.

[Gazdar 82] Gazdar,G.,"Phrase Structure Grammar," SLL15, P.Jacobson and G.K.Pullum(eds.), D.Reidel, pp.131-186, 1982.

[Gregory 84] Gregory,S.,"How to Use Parlog," Department of Computing, Imperial College, Oct. 1984.

[Kaplan 82] Kaplan,R.M. and Bresnan,J.,"Lexical-Functional Grammar: A Formal System for Grammatical Representation," in 'Mental Representation of Grammatical Relations' J.Bresnan(ed.), MIT Press, pp.173-281, 1982.

[Kay 80] Kay,M.,"Algorithm Schemata and Data Structures in Syntactic Processing," XEROX PARC, CSL-80-12, Oct.1980.

[Marcus 80] Marcus,M.P.:A Theory of Syntactic Recognition for Natural Language, MIT Press, 1980.

[Matsumoto 83] Matsumoto,Y.,et al.,"BUP: A Bottom-Up Parser Embedded in Prolog," New Generation Computing, vol.1,no.2, pp.145-158, 1983.

[Matsumoto 84] Matsumoto,Y.,Kiyono,M. and Tanaka,H., "Facilities of the BUP Parsing System," Proc. of Natural Language Understanding and Logic Programming, Rennes, 1984.

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

[Pratt 75] Pratt,V.R.,"LINGOL--A Progress Report," 4th IJCAI, pp.422-428, 1975.

[Pullum 82] Pullum,G.K. and Gazdar,G.,"Natural Language and Context-free Languages," Linguistics and Philosophy 4, pp.471-504,1982.