TM-0410

並列構文解析について

松本裕治

November, 1987

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

並　列　構　文　解　析　に　つ　い　て

松　本　裕　治
（ＩＣＯＴ）

1987年10月21日～23日
神戸市　関西地区　大学セミナーハウス

電子情報通信学会
データフローアーキテクチャと並列処理時限研究専門委員会
情報処理学会　　日本ソフトウェア科学会

並列構文解析について
On Parallel Parsing

松本　裕治
Yuji MATSUMOTO
（財）新世代コンピュータ技術開発機構
Institute for New Generation Computer Technology

あらまし
自然言語処理のための並列構文解析法を紹介する。最初に、著者らの論理型言語に基づく並列構文解析法について説明し、次いで、最近話題のコネクショニストモデルに基づく構文解析について紹介する。

1．はじめに

自然言語の文法の記述について、最近の文法理論（ＬＦＧ、ＨＰＳＧ等）では文法のクラスとして文脈自由文法程度を設定しておいても充分な記述が可能であることが示されている。このような文法形式に基づく構文解析について考えると、文脈自由文法は本質的に曖昧な文法を含んでおり、一般の文脈自由文法を対象とする構文解析には決定的な解析法がなく典型的な探索問題となっている。

最近、コネクショニズムや論理型言語の並列実行等の話題と相俟って、並列処理に関する研究が盛んである。構文解析はこのような並列処理によって飛躍的な効率改善が期待できる恰好の問題であり、いくつかの研究が見受けられる。

ここでは主に文脈自由文法を想定し、これを対象とする並列構文解析法を紹介しようと思う。よく知られた文脈自由文法の構文解析アルゴリズムが、逐次的な計算モデル上では入力文の長さをｎとすると$O(n^3)$の計算時間が必要であることが分っている［Aho 72］。つまり、入力文の長さの増加にしたがって、その３乗に比例する解析時間が必要な訳である。一方、並列計算については計算時間を評価するための共通に認められた土俵（並列計算モデル）がないのが問題ではあるけれども、例えば、ＰＲＡＭ（Parallel Random Access Machine）を想定すると、文脈自由文法の認識問題には$O(\log n)$しか計算時間が必要でないことがわかっている［Rytter 85］。ただし、ＰＲＡＭというのはメモリを共有する多数のプロセッサからメモリへの同時アクセスを仮定していたり、あるデータの状態に対するプロセッサの割り当てが一定の時間でできることを仮定していたりして、並列計算モデルとしては現実的ではない。また、時間的には効率のよい$O(\log n)$のアルゴリズムでは、かなりの量の余分な計算が起こってしまう。

ここでは、効率上の観点からも無駄な処理が少なく、しかも並列実行が期待できる構文解析法を紹介しようと思う。

まず、次章では、並列論理型言語での実現を目的とした著者らの並列構文解析アルゴリズムを説明する［Matsumoto 86］。これはもともと文脈自由文法用の並列構文解析法として考えられたものであるが、最近、文脈依存文法や更にそれ以上の記述力をもつ文法形式についても同じアルゴリズムが適用できることがわかった［Matsumoto 87］。また、解析中に同一処理の繰り返しを行わないので、逐次型のアルゴリズムとしてもEarleyのアルゴリズム［Earley 70］やチャートパージング［Kay 80］と比較しても劣らないので、Prolog上での実用的な構文解析システムとしても利用できる［松本 86］［杉村 86］。

第３章では、文脈自由文法による構文解析をコネクショニストモデルに基づいて実現した例として、Fantyのシステムを紹介する［Fanty 85］。基本的には、ＣＹＫアルゴリズム［Aho 72］で用いられているテーブルを素直な形でコネクショニストネットワークとして実現したものである。

第４章では、文脈自由文法に基づく構文解析とはかなり観点が外れるが、最近の並列構文解析の話題として有名なWaltz & PollackのMassively Parallel Parsing［Waltz & Pollack 85］を紹介する。構文解析の観点から外れると言ったのは、彼らの対象が文の構文解析にあるのではなくどちらかというと意味解析まで含めた曖昧性の除去のための枠組みをコネクショニズムの立場から提案しているからである。

2．並列論理型言語による並列構文解析

ＧＨＣやPARLOGなどの並列論理型言語のプログラムは、通信を交わし合うプロセスの集まりによって記述される。ここでは、文脈自由文法を対象にして、どのような考え方によって自然な並列構文解析が実現できるかを説明する。

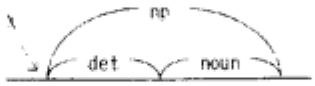
次のような簡単な文脈自由文法を考えてみよう。鉤括弧（'['、']'）内は単語（終端記号）であり、それ以外は文法カテゴリ（非終端記号）である。

【文法１】
(1) sentence --> np, vp.
(2) np --> det, noun.
(3) noun --> adj, noun.

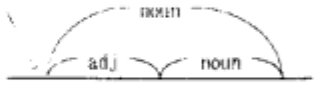
```
(4) noun --> noun, relc.
(5) relc --> [that], vp.
(6) vp --> verb.
(7) vp --> verb, np.
(8) det --> [the].
(9) adj --> [beautiful].
(10) noun --> [man].
(11) noun --> [woman].
(12) verb --> [walks].
(13) verb --> [loves].
```

図１．文法規則(2)の使用例

図２．文法規則(3)の使用例



(イ)図１の文法規則の接続関係



(ロ)図２の文法規則の接続関係

図３．文法カテゴリの接続関係

この文法に含まれるどの文法規則も右辺に三つ以上の文法カテゴリを含んでいない。以下では、このような所謂チョムスキー標準形の文脈自由文法に限って説明する。これは議論を簡単にするためであって、このような制限をもたない文法に対するアルゴリズムの拡張については最後に概略を述べることにする。

さて、単語や文法カテゴリなど文法規則を構成する基本要素をプロセスと考えるようなパージングモデルを仮定して見よう。入力文として与えられた単語列のそれぞれの単語が自立的にプロセスとして動き、隣り合うプロセス同志が通信をし合いながら文を作っていくようなモデルである。

その場合、単語は何をすればよいだろうか。例えば、'man'という単語は文法規則(10)によれば名詞（noun）であるから、これはすぐに 'noun'に生まれ変わればよい。これは(10)の文法規則をボトムアップに使用することに相当する。このように、右辺にただ一つの要素しか含まない文法規則に相当する処理は非常に単純に実現できる。例えば、文法規則(6)によれば、動詞（verb）は直ちに動詞句（vp）を生めばよい。それではそれ以外の形の文法規則に含まれる文法カテゴリはどう処理すればよいだろうか。

文法規則(2)について考えてみよう。これを用いてボトムアップに解析するというのはつまり、冠詞（det）と名詞が隣合わせて存在するときにこれらから名詞句（np）を生み出すことに相当する。det も noun もプロセスとして表現することを考えているから、この場合はどちらかのプロセスが自分の存在を一方のプロセスに伝えてやればよい。今、このような文法規則を扱うときに、右辺に現れる文法カテゴリのうち常に左の要素が右の要素に対して従属的な立場にあると決めてしまおう。従って、det は自分の存在を示すために自分の名前を右のプロセスに送り、右のプロセス noun はそれを受けて np というプロセスを生み出せばよい訳である（図１）。その他の文法規則についても同様であり、特に noun を右辺に含む文法規則(3)についても図２のように noun は同様の役割を果たす。

従って、det や adj が行う処理というのはこの場合非常に単純である。それぞれ自分の名前を右のプロセスに送るだけでよい。プログラムで書けば次のようになる。

```
(14) det(X,Y) :- true ! Y=[det(X)].

(15) adj(X,Y) :- true ! Y=[adj(X)].
```

このプログラムはＧＨＣによる記述であり、各述語の第一引数と第二引数はそれぞれ自分の左のプロセスからの入力と右のプロセスへの出力に相当している。

よって、(14)の意味は、det は右からの入力に関係なく左側へ det(X) というデータを送るということである。det(X) がリストに含まれた形になっているのは、一般にプロセスが授受するデータは複数個あるのでそれらを集合として表現するためにリストを使うからである。この例では送るデータがたまたま一つだけであったに過ぎない。adj についての定義(15)についても同様である。さて、これら二つが自分の名前だけでなく入力引数 X も同時に送っていることに注意してほしい。図３を見ると、det が送ったデータを noun が受け取ったときにできる np はもともと det の右にあった要素と接するのであるから、det が受け取っていたのと同じデータ（すなわち X）を受け取る必要がある。X 送られているのはこのようにいずれ np ができた時にその入力としてこの入力引数を渡すためである。

noun の定義も同様にプログラムによって記述してみよう。noun は文法規則(4)の右辺にも含まれているので、図１、図２に対応する処理以外にこれに対応する処理も行なわねばならない。

```
(16) noun(X,Y) :- true !
        Y = [noun(X)|Tail],noun2(X,Tail).

     noun2([],Y) :- true ! Y = [].
     noun2([det(X)|Tail],Y) :- true !
        np(X,Y1),noun2(Tail,Y2),
        merge(Y1,Y2,Y).
     noun2([adj(X)|Tail],Y) :- true !
        noun(X,Y1),noun2(Tail,Y2),
        merge(Y1,Y2,Y).
```

```
noun2([_|Tail],X) :- otherwise |
    noun2(Tail,X).
```

noun(X) というデータが出力されている部分が文法規則(4)に含まれている noun に相当する。noun2 の定義が図１、図２の右下の noun の働きに対応する。

noun2 の第２番目のＧＨＣ節は図１の noun に相当し、det(X) というデータを受け取ると np というプロセスを生成している。np が第一引数として X を受け取っているのは、図３に示されているように、この新しく得られた np の受け取るべき入力データが X に他ならないからである。また、noun2 は、他にも入力データを受け取るかも知れないので、残りのデータ（Tail部分）に対して再び noun2 が呼ばれている。このＧＨＣ節の本体部の np と noun2 の出力が merge（第一引数と第二引数のリストの和集合を第三引数に返す述語）によって一つのリストにまとめ上げられ、全体の出力となっている。これは、図３の(イ)において noun と np の右端が一致しており、これらの右隣にある文法要素が受け取るのがこれら両方の出力である必要があるからである。noun2 の第三のＧＨＣ節についても同様である。第一の節は、入力データを処理し終わったときにプロセスを終了させるための定義であり、第四の節は、入力データが det(X) でも adj(X) でもなかったとき（すなわち、名詞にとって関係ないデータであったとき）にそれを読み飛ばすための定義である。

その他の文法カテゴリについても同じようにして、並列論理型言語による定義が可能である。図５に【文法１】をＧＨＣプログラムに変換した結果を示す。このプログラムを用いると、例えば(17)の文の構文解析を行うには、(18)のようなゴール列を初期ゴールとして呼べばよい。

```
(17) The man loves the beautiful woman.

(18) the([begin],D1),man(D1,D3),loves(D3,D4),
       the(D4,D5),beautiful(D5,D6),woman(D6,D7),
       fin(D7).
```

この入力文が解析される様子を図６に示す。図中、太い矢印はプロセスの生成に相当し、細い矢印はデータの送受に対応する。あるプロセスは図４の(1)のように入力データの値とは無関係にデータを出力し、また、あるものは図４の(2)のように入力データに依存して新しいプロセスを生成する。構文解析の手続きにおいて逐次性が生じるのは図４の(2)のような場合だけであって、その他の部分がすべて並列に動作できる。特に、図４の(1)の場合に、入力引数がたとえ変数のままであっても、データを出力できることに注意してほしい。

プログラムの実行の結果知りたいのは入力文から sentence が構成できるかどうかであるので、図５のプログラムで sentence の定義だけが少し特殊になっている。[begin]というのは、(18)からわかるように入力文の先頭の単語に与えられる値である。もしこれが sentence に受け取られるとすると、その意味は、得られた sentence の左端が文頭に一致しているということである。この sentence の右端が文末に一致していることが解析が成功する条件である。それを調べるために、sentence は、[begin] を受け取ると end という新たなデータを右のプロセスに送る。fin というプロセスが(18)の文末に置かれているが、これが文の完成を調べるためのプロセスであり、単に end というデータを待っているだけのプロセスである。上の説明からわかるように、end というデータは文頭から完成した sentence だけが発生するデータであり、これが文末に位置する fin に受け付けられたときに初めて文の解析が成功したことがわかるのである。

図６を最終的に得られた sentence から逆向きに眺めてみると、これが構文解析木に対応していることがわかるだろうか。これにより、この並列構文解析が与えられた文法に基づくボトムアップ構文解析を実現していることがわかって頂けると思う。

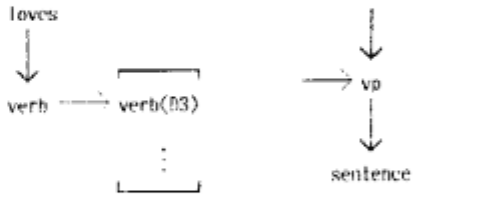
構文解析のアルゴリズムの計算の複雑さとして、注意しておきたいのは、一旦ある文法カテゴリを根にもつ構文解析木が得られるとそれはプロセスとして生まれ、その左に隣接するすべてのプロセスからの出力を受け取り、そのデータを調べ尽くすまで生き続けることである。また、まったく同じプロセスは生成されることがなく、このような意味で重複計算がない。

トップダウン予測

ここまでに説明した並列構文解析は純粋にボトムアップ構文解析であり、所謂トップダウン予測を行っていない。ここでは、今まで述べた並列構文解析法に簡単にトップダウン予測を取り入れることができることを説明する。

例えば、文頭に動詞が現れたと仮定しよう。求めたいのは sentence であり、文法１からは、動詞を先頭とする文はどのようにしても構成できない。ところが、文頭の単語が冠詞ならば、文を構成する上で問題がない。このようなことがわかるためには解析の各段階でどのような文法カテゴリを求めようとしているのかを利用するようにしなければならない。

図４の(1)のように入力データに関係なく新しいデータを生成している部分があった。トップダウン予測はこのようなデータの出力を一時的に抑制する処理とし



(1)入力データに無関係に新しいデータを出力　　(2)入力データを受けて新しいプロセスを生成

図４．プロセスのデータの授受

```
np(X,Y) :- true |
    Y = [np(X)|Tail],np2(X,Tail).
np2([],Y) :- Y = [].
np2([verb(X)|Tail],Y) :- true |
    vp(X,Y1),np2(Tail,Y2),merge(Y1,Y2,Y).
np2([_|Tail],Y) :- otherwise |
    np2(Tail,Y).

vp([],Y) :- Y = [].
vp2([np(X)|Tail],Y) :- true |
    sentence(X,Y1),vp2(Tail,Y2),merge(Y1,Y2,Y).
vp2([_|Tail],Y) :- otherwise |
    vp2(Tail,Y).

noun(X,Y) :- true |
    Y = [noun(X)|Tail],noun2(X,Tail).
noun2([],Y) :- true | Y = [].
noun2([det(X)|Tail],Y) :- true |
    np(X,Y1),noun2(Tail,Y2),merge(Y1,Y2,Y).
noun2([adj(X)|Tail],Y) :- true |
    noun(X,Y1),noun2(Tail,Y2),merge(Y1,Y2,Y).
noun2([_|Tail],Y) :- otherwise |
    noun2(Tail,Y).

det(X,Y) :- true | Y = [det(X)].

adj(X,Y) :- true | Y = [adj(X)].

rel([],Y) :- true | Y = [].
rel([noun(X)|Tail],Y) :- true |
    noun(X,Y1),rel(Tail,Y2),merge(Y1,Y2,Y).
rel([_|Tail],Y) :- otherwise |
    rel(Tail,Y).

that(X,Y) :- Y = [that(X)].

verb(X,Y) :- true | Y = [verb(X)|Y],vp(X,Y).

sentence([],Y) :- true | Y = [].
sentence([begin],Y) :- true | Y = [end].
sentence([_|Tail],Y) :- otherwise |
    sentence(Tail,Y).

the(X,Y) :- det(X,Y).

beautiful(X,Y) :- adj(X,Y).

man(X,Y) :- noun(X,Y).

woman(X,Y) :- noun(X,Y).

walks(X,Y) :- verb(X,Y).

loves(X,Y) :- verb(X,Y).
```

図５．文法１の並列構文解析プログラム

入力文：the man loves the beautiful woman.

初期ゴール：the([begin],D2),man(D2,D3),loves(D3,D4),the(D4,D5),beautiful(D5,D6),woman(D6,D7),fin(D7).

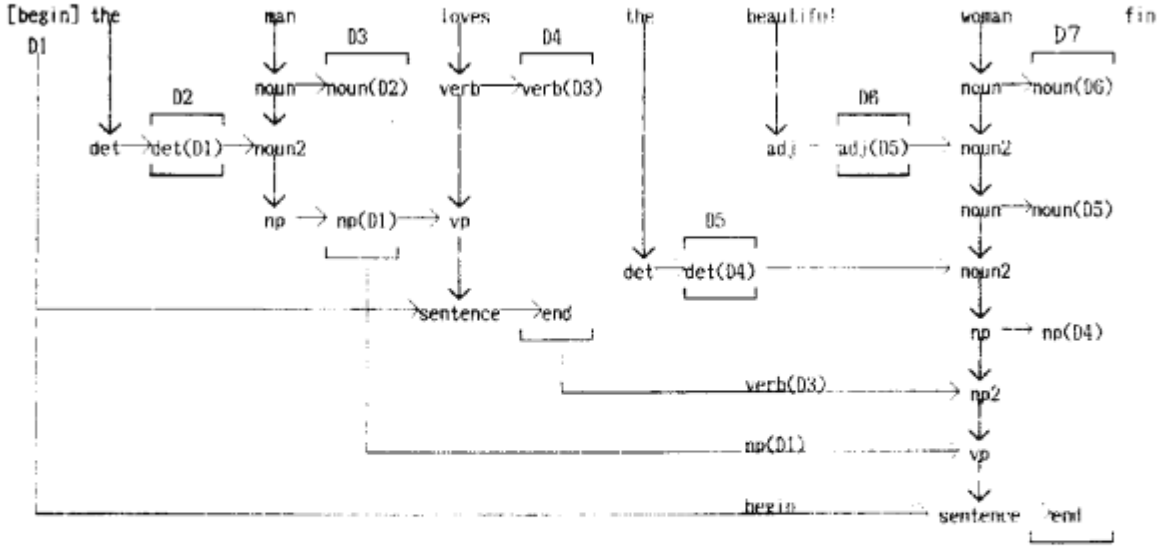


図６．並列構文解析の実行例

て実現できる。

例えば、冠詞が得られて、det(X) というデータを出力しようとしたと仮定しよう。det(X) を生成する処理は、文法１の文法規則(2)によると、いずれ np を作り上げるための処理である。したがって、np を予測しているデータが入力引数 X の中に含まれていないならば、det(X) というデータを生成するのは無駄である。npを予測しているデータについて説明しよう。例えば、verb( ) は np を予測している。なぜなら、文法規則(7)により、このデータを処理できるのが np だからである。また、np を左端の子として持ち得る文法カテゴリを予測しているデータが X に含まれていてもよい。このようなデータも np を予測しているものと考えることができる。例えば、begin というデータは文頭の単語に与えられるデータであるからもちろん sentence を予測していると考えられるが、これはまた、np をも予測している（文法規則(1)からわかる。また、同様に文法規則(2)により、begin は det も予測している）。

このように、新しいデータを生成するプロセスがどのようなプロセスを作るためにそのデータを出力しようとしているかということと、新しくできるであろうプロセスを予測しているデータがどのようなものであるかということは、文法規則が与えられた時点で予め計算しておくことが可能である。

細かい処理手順は省略するが、実際のシステムではトップダウン予測をより積極的に利用し、データを発生するプロセスに対してその入力データの中からそのプロセスがいずれ作ろうとしている文法カテゴリを予測しているデータのみを取り出すフィルタを割り当てている。フィルタは他のプロセスと並列に動き、もしフィルタの出力が空ならば、そのフィルタが割り当てられたプロセスはデータを出力しないようにする。ただし、データの出力はフィルタの計算結果を最後まで待つ必要はなくて、フィルタを抜け出して来るデータが一つでもあれば、プロセスは隣へデータの送り出しを行ってもよい。

チョムスキー標準形以外の文脈自由文法について

右辺に三つ以上の要素を含む文法規則がある場合には、プロセスが送受するデータに一工夫が必要になる。つまり、

```
a --> b, c, d.
```

のような文法規則があるときに、文法カテゴリ b は自分の名前を送るだけでよいが、それを受け取ったプロセス c は自分の名前を送るのでは情報が不完全である。このため実際の構文解析システムの実現においては、まさにこの文法規則の c の役割を実現するため、この文法規則の c と d の間の位置を指示する特別の識別子（例えば、id1 などという固有の項）を割り当て、この文法規則内の c の役割を次のＧＨＣ節により実現している。

```
c2([b(X)|Tail],Y) :- true |
        Y = [id1(X)|Y1],c2(Tail,Y1).
```

また、これに伴い d に関するプログラムにも次のＧＨＣ節を追加する。

```
d2([id1(X)|Tail],Y) :- true |
    a(X,Y1),d2(Tail,Y2),merge(Y1,Y2,Y).
```

同様に、三つ以上の要素を持つ文法規則の右辺の二番目以降に現れる文法カテゴリの間にこのような固有の識別子が与えられ、上のようなプログラムが定義される。詳細については省略するが、このような考え方で任意の文脈自由文法に対して本並列構文アルゴリズムが適用できることは明らかであろう。

最後に一つだけ注意しておく。実際のシステムでは、文法規則が（制限された）Definite Clause Grammar（ＤＣＧ）［Pereira 80］で記述されることを仮定している。ＤＣＧにおいては、例えば、

```
a --> b, { prog1 }, c, ...
d --> b, { prog2 }, e, ...
```

のように、文法規則中にPrologのプログラム（この例の場合 prog1,prog2）を挿入することができる。（また、ここでは省略したが文法カテゴリが引数をもつことができる。）そして、この例のように、同じ b であっても、prog1 と prog2 の計算の結果が異なるかも知れないので、b が得られた場合に単純に b(X) というデータを出力するだけでは事が済まない。実際のシステムでは、右辺の先頭の文法カテゴリにも固有の識別子を割り当て、それを出力させるようにしている。

３．コネクショニストネットワークによる文脈自由文法の並列構文解析

［Fanty 85］によるコネクショニストモデルに基づく並列構文解析について説明する。彼の考え方は基本的に文脈自由文法に対するＣＹＫ（Cook-Younger-Kasami）アルゴリズムに従っている。ＣＹＫアルゴリズムというのは、文脈自由文法が与えられたときに図７のような表を作ることによって構文解析を行う方法である。この図は、図の右上に書かれた五つの文法規則を用いて入力系列 aabbb を解析したときに得られる表である。この表は、系列の開始点が i、系列の長さが j の箱（以後（ i , j ）と呼ぶ）の中に非終端記号 A があるとき、入力系列の第 i 番目から第 i ＋ j － 1 番目の終端記号からなる部分系列によって非終端記号 A が構成できることを表わしている。（１，５）の箱に S があるのはこの入力系列が正しい文であったことを意味している。

表の作り方は非常に簡単である。一般に、（ i , j ）の箱に B という記号、（ i ＋ j , k ）の箱に C という記号があり、A → B C という文法規則が存在するとき、（ i , j ＋ k ）の箱に A という記号を書き込めばよい。表では（１，５）の箱に S を書き込むのに貢献した記号に下線を引いておいた。

Fanty の考え方は、このアルゴリズムに極めて忠実にネットワークを構成することである。ネットワークは基本的な次の３種類の構成要素および特殊な構成要素１種類によって作られる。

terminal unit：系列の長さ１の列のそれぞれの箱に対応する位置に終端記号の種類の数だけ terminal unit が置かれる。入力文の第ｉ番目の入力記号が ａ であるとき、（ｉ，１）の箱の ａ に対応するterminal unitが発火される。

nonterminal unit：表のすべての箱に対応する位置に非終端記号種類の数だけnonterminal unitが置かれる。各nonterminal unitは、対応する非終端記号をもっている。nonterminal unitは、次のmatching unitによって発火される。

matching unit：各nonterminal unitに対応して、その非終端記号を左辺にもつ文法規則とその規則を用いることが可能な位置の組み合わせに対してmatching unitが置かれる。例えば、Ａ → Ｂ Ｃ という文法規則があるとき、（ｉ，ｊ）の位置の Ａ に対応するnonterminal unitに対し、（ｉ，１）と（ｉ＋１，ｊ－１）、（ｉ，２）と（ｉ＋２，ｊ－２）．．．（ｉ，ｊ－１）と（ｉ＋ｊ－１，１）のｊ－１通りのそれぞれの組み合わせに対してmatching unitが置かれる。つまり、一つのmatching unitは一つの文法規則の特定の位置での使われ方に対応しており、文法規則の右辺に対応する二つのnonterminal unitが発火しているとき、左辺に対応するnonterminal unitを発火させる。

$ unit：文末をしめすための構成要素である。terminal unitと同様に、系列の長さ１の列に一つずつ置かれる。（ｉ＋１，１）の箱の $ unitは、（１，ｉ）の箱の非終端記号 Ｓ（文を表わす非終端記号）につながっている。

これらのユニットの種類および配置を知れば、構文解析の方法は明らかであろう。入力文の長さがｎならば、１≦ｉ≦ｎの各ｉについて（ｉ，１）の位置の入力文の第ｉ番目の終端記号に対応するterminal unitを発火させる。また、（ｎ＋１，１）の位置の $ unitを発火させる。matching unitによって次々にnonterminal unitが発火して行く。（１，ｎ）の位置の非終端記号 Ｓ に対応するnonterminal unitが発火することが、入力文が正しい文であったことに相当する。

以上の装置によって入力文が正しい文かどうかは決定できるが、実用的には、構文解析木などの構文解析の結果を得る必要がある。そのために、上で述べたユニット間の結合はすべて双方向になっている。また、各ユニットは、二種類の発火状態を持つ。

入力文からのユニットの発火動作をボトムアップ処理と呼ぶとしよう。ボトムアップ処理中に発火したユニットは、活性化状態（active）になる。（１，ｎ）の位置にある文記号 Ｓ が活性化されると、（ｎ＋１，１）の位置の $ unitによってそれは第二の状態である過活性状態（hyperactive）になる。これからがトップダウン処理である。過活性状態のnonterminal unitは、それに結合しているmatching unitに自分が過活性であることを知らせる。matching unitが既に活性状態ならば、それは過活性状態になり、さらに自分の下につながっているnonterminalもしくはterminal unitに自分が過活性であることを伝える。この信号を受け取ったユニットは自分が活性である場合に限り過活性状態になる。このようにして、Ｓ を作るために貢献していたすべてのユニットが過活性状態になり、ここから構文解析に関する情報を得ることができる。

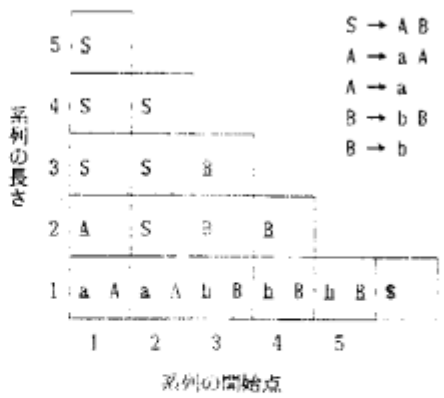


図７．ＣＹＫアルゴリズムによる解析

アルゴリズムおよびネットワークの構成が非常に簡単であるという特徴を持つ反面、この方法にはいくつかの問題点がある。

一つは、入力文の長さの上限がネットワークの大きさによって制限されてしまうことである。図７のような表に対応するネットワークを事前に作っておく訳であるから、用意しておいた表の横幅以上の長さの文はこのネットワークでは解析することができない。

二つめは、解析可能な文の長さをｎとすると、ネットワークにはｎの３乗に比例する数のユニットが必要なことである。ただし、これは文法規則がチョムスキー標準形である場合のことで、文法規則の右辺に現れる要素の数の最大値がｍとすると、ネットワークにはｎのｍ＋１乗に比例するユニットが必要である。しかし、本当に問題なのは、必要なユニットの数のオーダーではなくその係数の大きさである。例えば、入力文の長さの上限を15、非終端記号の数を10、同一非終端記号を左辺に持つ文法規則の数が５つずつと仮定すると、matching unitの数は、文法がチョムスキー標準形のときは 34,000個、文法規則の右辺の要素の数がすべて３とすると 153,750個必要になる。著者も述べているように、実際にはすべての箱の位置にnonterminal unitが必要な訳ではないので、この数のmatching unitがすべて必要とは限らない。例えば、前置詞句を作るのに少なくとも三単語必要であることがわかっているならば、表の第二行までには前置詞句に対応するnonterminal unitが不必要であるし、そうすると matching unitの数も少なくて済む。しかし、入力文の上限を増

やすに従って、このために不必要になるユニットの割合は減少すると考えられる。

また、これはＣＹＫアルゴリズム自体の問題であるが、この方法では、前節の並列構文解析システムのようなトップダウン予測を解析に活かすことができない。

4．Massive Parallel Parsing

本章では、[Waltz & Pollack 85] のコネクショニストモデルに基づく自然言語文の並列解析を簡単に紹介する。まえがきでも述べたように、彼らの仕事は構文解析というよりも、それに伴う曖昧性の解消をコネクショニストモデルによって説明したことに意味がある。例えば、

John shot some bucks.

という文を解析することを考えてみよう。入力文の各単語はいくつかの品詞に属し、いくつかの異なる意味概念を持っている。品詞や概念が節点として表現され、互いに相反する概念が抑制枝で結ばれ、両立する概念間が活性化枝によって結ばれたネットワークを考える。

構文解析としては、チャートパージングが用いられており、複数の構文解析結果が存在する場合には、それぞれの構文解析木について両立する非終端記号間は活性枝で結ばれ、両立しない非終端記号間は抑制枝によって結ばれる。

また、上の例文は、「ジョンが鹿を撃った」という意味と「ジョンが（賭で）金を得る」という意味に解釈できる。よって、Huntingなどという文脈情報があれば、前者の解釈が優先されることになる。

彼らは、Huntingという文脈情報が与えられたネットワークが一定時間の後に前者の解釈に落ち着くことを示している。

構文解析を自然言語文の解析の過程においてどのようなレベルで取り扱うかについては様々な議論がある。文解析のための指針（guideline）として文の構文構造を用いる考え方もあるが、文に曖昧性が生じるのは構文処理機構の副作用であり、解析の指針となる原理は「曖昧性を解消しようという普遍的な意識（universal will to disambiguate）」であるとする彼らの取っている態度は非常に興味深い。別の例として、

The astronomer married a star.

という文の解釈において、star の意味として、最初、astronomerによって「星」という意味が優っていたのに、その後、married が持つ意味的な制約によって、次第に、「映画スター」という意味に活性化が移行していく例が示されており、この例も非常に興味深い。

5．あとがき

並列構文解析について、いくつかのアプローチを紹介した。それぞれ立場や特徴が異なるので単純に比較することは難しい。

最初に示した著者らの並列構文解析法は、並列論理型言語という並列環境に載せるという考えのもので、そういう意味では汎用の環境での実現を目指している。

別のアプローチとしては、専用のマシンなり、環境なりを設定した方法がある。後半のコネクショニストモデルによるアプローチはこちらの種類に属する。その中でも二つの比較的両極端なアプローチを紹介した。

Fantyの方法は、構文解析専用の結合マシンを設計しようというもので、コネクショニストといっても、人間の行っている構文解析過程とは程遠い。

一方、Waltzらのアプローチは、人間の文理解をある程度模倣したアプローチであり、いかにもコネクショニストモデルという名にふさわしい。また、彼らは個々の概念をさらに細かいプリミティブに分け、概念間の両立や抑制関係の記述に用いており、これによってより広範囲の文脈情報を利用することを実験している。

【参考文献】

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

[Earley 70] Earley,J., "An Efficient Context-free Parsing Algorithm," C.ACM, Vol.13, 1970.

[Fanty 85] Fanty,M., "Context-Free Parsing in Connectionist Networks," TR174, Department of Computer Science, University of Rochester, Nov. 1985.

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

[Matsumoto 86] Matsumoto,Y., "A Parallel Parsing System for Natural Language Analysis," Proc. 3rd ICLP, London, 1986.

[松本 86] 松本裕治、「論理型言語に基づく構文解析システムＳＡＸ」、コンピュータソフトウェア、Vol.3, No.4, pp.308-315, Oct. 1986.

[Matsumoto 87] Matsumoto,Y., "Parsing Gapping Grammars in Parallel," to appear as ICOT Technical Memo, 1987.

[Pereira 80] Pereira,F. and Warren,D., "Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augumented Transition Networks," Artificial Intelligence, Vol.13, pp.231-278, 1980.

[Rytter 85] Rytter,W., "Parallel Time O(log N) Recognition of Unambiguious CFLs," Lecture Notes in Commuter Science 199, pp.380-389, 1985.

[杉村 86] 杉村領一、松本裕治、「構文解析システムＳＡＸのＣＩＬによる実現」、情報処理学会全国大会、Oct. 1986.

[Waltz & Pollack 85] Waltz,D.L. and Pollack,J.B., "Massively Parallel Parsing: A Strongly Interactive Model of Natural Language Interpretation," Cognitive Science, Vol.9, No.1, pp.51-74, 1985.