ICOT Technical Report: TR-200

TR-200

論理型言語に基づく構解析システムSAX

松本裕治、杉村領一

September, 1986

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

論理型言語に基づく構文解析システムＳＡＸ

ＳＡＸ：A Parsing System based on Logic Programming Languages

松本裕治　杉村領一
Yuji Matsumoto, Ryoichi Sugimura
（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

構文解析システムＳＡＸは、ＤＣＧ（Definite Clause Grammars）で記述された文法を対象とする自然言語処理用のシステムで、ＤＣＧをPrologのプログラムに変換するトランスレータより成る。変換の手法自体は、ＤＣＧで記述された文法規則から並列論理型言語への変換を目的として考えられたものであるが、逐次型言語の上で実現されても効率のよい構文解析システムとして利用することができる。ＳＡＸ（Sequential Analyzer for syntaX and semantiCS）は、特に逐次型言語の上で開発されたシステムの呼び名である。並列論理型言語の上で実現されたシステムは、ＰＡＸ（Parallel AX）と呼ばれる。変換によって得られたPrologプログラムは、Prologに完全にコンパイルされたものになっており、副作用を用いないことやプログラムの動作が決定的になっていることなどの特徴がある。本システムは、特に、コンパイラを有するPrologに向いた構文解析システムである。

１．はじめに

論理型言語Prologに基づく文法記述形式ＤＣＧ（Definite Clause Grammars）[Pereira 80]は、文脈自由文法を構文規則の基本とし、単一化（unification）とPrologのプログラムによって補強された記述形式である。ＤＣＧによって記述された文法規則は、Prologの節に一対一に変換され、下降型の構文解析システムとして用いることができる。ただし、よく指摘されているように、この下降型構文解析は左再帰規則によって解析が無限ループに陥る危険性があることや、繰り返し計算により効率上の問題があることなどの欠点がある。

著者らは、その欠点を回避するために、ＤＣＧで記述された文法規則を上昇型の構文解析プログラムに変換する方法を提案した。このシステムは、ＢＵＰと呼ばれている[Matsumoto 83,84]。著者らは、その後、ＢＵＰで用いていた構文解析を並列論理型言語の上でも実現できるように改良することを目的として並列構文解析法を考えた[松本 86]。本論文で、紹介するのは、この並列構文解析システムの逐次版である。ＢＵＰと同様に本システムもＤＣＧによって記述された文法規則を対象とし、それをPrologプログラムに変換することによって上昇型の構文解析シス

テムを実現する。ＢＵＰに比べて本システムの特徴は、文法規則がPrologプログラムに完全にコンパイルされた形になっており、また、副作用をまったく用いないので、コンパイラを持つProlog処理系に向いていることである。

本システムは、飽くまで汎用の構文解析システムを目指しているが、任意のＤＣＧの記述を取り扱うためには、考慮しなければならない問題点がある。次章以下の二つの章では、まず、文脈自由文法を用いて本システムの基本的な考え方を示し、後半で、ＤＣＧを取り扱うにあたっての問題点とその対処法について考察する。

２．構文解析システムＳＡＸの基本的変換法

本節では、文脈自由文法を用いて、ＳＡＸの変換方法を説明する。文法規則の記述は、ＤＣＧに従い、終端記号を鉤括弧で括って表現する。ＳＡＸのアルゴリズムは、基本的には左隅構文解析法である。すなわち、或るまとまった非終端記号（または、終端記号）が構成されると、それを基にして、より大きな解析木を構成しようとするが、そのとき、その（非）終端記号を左端の要素とする新しい解析木を作ろうとする。つまり、この非終端記号を右辺の先頭の要素として持つ文法規則を選び、その文法規則を完成させようとする。このような文法規則の右辺の二番目以降の要素は、新しい（非）終端記号が得られる度に、埋め合わされて、完全な解析木へと作り上げられていく。

つまり、或る（非）終端記号が完成されたときには、二通りの仕事がある。一つは、その要素を右辺の先頭に持つ文法規則について、その規則を展開することであり、もう一つは、部分的に展開された文法規則を、より完全なものに近づけることである。これらの処理は、Earleyのアルゴリズム［Earley 70］やチャートパーザ［Kay 80］などの文脈自由文法のための構文解析アルゴリズムで行なわれているのと同等な処理である。

それでは、これをどのように論理型言語に実現すればよいだろうか。まず、各文法規則の右辺に、次のように、それぞれの場所を示す識別子を設けることにする。

```
(1) sentence --> np, id1 vp.
(2) np --> det, id2 noun.
(3) np --> det, id3 noun, id4 rel_clause.
(4) np --> np, id5 coconj, id6 np.
(5) rel_clause --> [that], id7 sentence.
(6) vp --> verb.
(7) vp --> verb, id8 np.
(8) vp --> vp, id9 coconj, id10 vp.
(9) det --> [the].
(10)noun --> [man].
```

```
(11)noun --> [woman].
(12)verb --> [loves].
(13)verb --> [walks].
(14)coconj --> [and].
```

id1からid10までが、識別子である。これらは、特定の文法規則内の特定の位置を表わしている。例えば、id1は、(1)の文法規則のnpまでの解析が終わった状態を表わしている。ＳＡＸでは、終端記号および非終端記号がすべてPrologの述語として表現されており、上のように割り当てられた識別子をそれらが受け渡しすることにより解析が進められる。各（非）終端記号は、２引数の述語として表現され、一つが前からの情報を受け取るための入力引数、もう一つが次の述語へ情報を渡す出力引数である。ＳＡＸでは、（非）終端記号の完成は、それに対応する述語の呼び出しに対応する。例えば、名詞句（np）が完成したとしよう。npを右辺の先頭に持つ文法規則には、(1)と(4)の二つがある。これらに対応した処理を行なうために、名詞句npは、id1とid5の二つの識別子を同時に出力する。具体的には、次の節によって定義される動作を行なえばよい。

```
(a) np1(X,[id1(X),id5(X)]).
```

この節の述語名がnp1になっているのは、npが完成したときには、文法規則の右辺の先頭にあるnpだけでなく、右辺のその他の位置にあるnpに対する処理も必要で、そのための別の処理を定義するための述語も必要だからである。例えば、(4)や(7)の右辺の先頭以外のnpについて次の様な定義を与えなければならない。

```
(b) np2([],[]).
    np2([id6(X)|T],Y) :-
        np(X,Y1),np2(T,Y2),append(Y1,Y2,Y).
    np2([id8(X)|T],Y) :-
        s(X,Y1),np2(T,Y2),append(Y1,Y2,Y).
    np2([_|T],Y) :- np2(T,Y).
```

以上で示した(a),(b)の述語の定義を名詞句に対するタイプ１の述語およびタイプ２の述語と呼ぼう。名詞句が完成したときにはnpが呼び出されるが、npは、np1とnp2によって次の様に定義されている。

```
(c) np(X,Y) :-
        np1(X,Y1),np2(X,Y2),append(Y1,Y2,Y).
```

他の（非）終端記号についても同様である。但し、例えば、上の文法のconjの様に、文法規則の右辺の先頭に一度も現れないものは、タイプ２の述語しかもたないので、(c)の様な定義は、必要なく、(d)の様に定義される。タイプ１の述語しか持たない（非）終端記号は、勿論、タイプ１の述語のみによって定義される。

```
(d) coconj([],[]).
    coconj([id5(X)|T],[id6(X)|Y]) :- coconj(T,Y).
    coconj([id9(X)|T],[id10(X)|Y]) :- coconj(T,Y).
    coconj([_|T],Y) :- coconj(T,Y).
```

タイプ１の述語は、識別子の集合を出力し、タイプ２の述語は、このような識別子を取り込んで、他の識別子を出力したり、他の非終端記号に対応する述語を生成したりする。(a)から(d)の定義を見てわかるように、このようにして出力された識別子の集合は、すべてappendなどによって一つのリストにまとめ上げられている。(b)や(d)などのタイプ２の述語に共通に見られる先頭の節は、与えられた入力が空のときに空リストを出力するための節であり、最後の節は、入力リストの先頭に現れる識別子がその非終端記号にとって使い得ないものであったときにその識別子を捨てるための節である。

(a)の定義で、出力として送り出される識別子が第一引数（X）を取り込んでいることに注意されたい。この理由と、識別子の受け渡しを説明するのが図１である。図は、文の一部が、文法規則(4)によって解析されている様子を示している。図では、受け渡しされる識別子の一部しか示されていないが、(a)によって出力されたid5(X)が、(d)の二番目の節によってid6(X)という新たな識別子を出力し、それが(b)の二番目の節によって受け付けられて、新しいnpを呼び出している様子が示されている。このとき、Xという変数が、新しいnpの入力引数として渡されていることに注意されたい。識別子が入力引数を運んでいたのは、このように新たな非終端記号が生まれたときに、正しい入力データを渡すためである。

ＳＡＸの基本的な変換法は、以上である。しかし、二つのリストを連結するためにappendを用いるのは効率がよくないため、実際のシステムでは、この部分を差分リスト（difference lists）を用いて処理している。従って、(a),(b),(c),(d)は、次の様に定義すればよい。

```
(a)' np1(X,[id1(X),id5(X)|Yt],Yt).
(b)' np2([],Y,Y) :- !.
     np2([id6(X)|T],Y,Yt) :-
        np(X,Y,Y1),!,np2(T,Y1,Yt).
```

```
     np2([id8(X)|T],Y,Yt) :-
        s(X,Y,Y1),!,np2(T,Y1,Yt).
     np2([_|T],Y,Yt) :- !,np2(T,Y,Yt).
(c)' np(X,Y,Yt) :- np1(X,Y,Y1),np2(X,Y1,Yt).
(d)' coconj([],Y,Y) :- !.
     coconj([id5(X)|T],[id6(X)|Y],Yt) :- !,
        coconj(T,Y,Yt).
     coconj([id9(X)|T],[id10(X)|Y],Yt) :- !,
        coconj(T,Y,Yt).
     coconj([_|T],Y,Yt) :- !,coconj(T,Y,Yt).
```

定義の所々にカットシンボルが挿入されているのは、これらの節が決して後戻りをしないからである。こうすることによって、末尾再帰（tail recursion）の最適化を行なっているProlog処理系で効率を稼ぐことができる。差分リストによるこれらの表現は、ＤＣＧの記法を借りると、更に簡潔に記述できる。DEC-10 Prologを始め多くのPrologには、ＤＣＧトランスレータが内蔵されていることが多い。従って、著者らが開発したＳＡＸトランスレータも、実際には、次の(a)"から(d)"までの様な節を生成する。

```
(a)" np1(X) --> [id1(X),id5(X)].
(b)" np2([]) --> !.
     np2([id6(X)|T]) --> np(X),!,np2(T).
     np2([id8(X)|T]) --> s(X),!,np2(T).
     np2([_|T]) --> !,np2(T).
(c)" np(X) --> np1(X),np2(X).
(d)" coconj([]) --> !.
     coconj([id5(X)|T]) --> [id6(X)],!,coconj(T).
     coconj([id9(X)|T]) --> [id10(X)],!,coconj(T).
     coconj([_|T]) --> !,coconj(T).
```

ＤＣＧの記法を借りたこれらの記述の方がわかり易いのではないだろうか。また、上でも述べたように、ＳＡＸでは、終端記号と非終端記号の区別を行なう必要がない。例えば、(9)の規則は次のように変換される。

```
(9)" the(X) --> det(X).
```

もし、'the'について複数の定義があれば、それらがすべて節の本体部に記述され

る。他の終端記号についても同様である。

さて、この変換によって得られたPrologプログラムを用いて、例えば、'The man walks.'という文の構文解析を行なうには、次のような節をゴールとして呼べばよい。

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

ここに、beginは、文の先頭を表わすための識別子である。また、最終的に求めたいものは、文すなわちsentenceである。解析が成功するのは、与えられた文の単語をすべて用いてsentenceが構成された時である。従って、sentenceのタイプ２の定義に、次のような節を追加する。

```
(f) sentence2([begin|T]) -->
        [end],!,sentence2(T).
```

つまり、beginという文頭の識別子は、sentenceのみに理解されて、そのとき、endという新たな識別子を生成する。(e)のfinという述語は、このendという識別子を理解するための述語である。文の解析が成功するのは、finが、endを受け取る時、かつその時に限る。図２に、ＳＡＸによる構文解析の様子を示す。図中、太い矢印は、述語の呼びだしに対応し、細い矢印は、識別子の遣り取りに対応する。入力文の単語から始まって、構文解析木を作り上げていく過程が、述語間のデータの受け渡しによって行なわれているのがよくわかる。X,Y,Zは、単語間に置かれた共有変数である。D1は、[begin]の代わりに用いた。識別子は、かなり大きなリスト構造を引数として持つ構造になるが、Prologでは、構造共有（structure sharing）が行なわれているので、実際には、大した負荷ではない。また、一つの構文要素（名詞句、動詞句）などが完成するのは、前にも述べたように、それに対応する述語の呼び出しに相当するが、このような構文要素は、一度作られると、二度と作り直されることがない。これは、ＳＡＸの大きな特徴で、副作用なしに、Earleyのアルゴリズム［Earley 70］、ＬＩＮＧＯＬ［Pratt 75］、チャートパージング［Kay 80］などで行なわれているような、表（well-formed substring table）を用いた構文解析法と同等の処理を行なうことができる。

３．トップダウン予測

前節で述べた変換法によって得られるPrologプログラムは、文脈自由文法に対する上昇型の構文解析システムを実現していた。しかし、そのアルゴリズムでは、下降型の予測はまったく用いられていなかった。例えば、構文解析の始めには、文を

構成することが目的であるから、入力文の先頭として許されている構文要素は何であってもよい訳ではない。つまり、文の解析木において左端の子となり得るものだけが文の先頭要素として許される。このことは、文の解析において、文中のいかなる部分についても言えることである。このように、文の解析途中の個々の状況において、どのような非終端記号を作るべきかという上からの予測（トップダウン予測）を積極的に利用することによって、全体の解析空間を狭めることが可能である。これは、ＬＩＮＧＯＬではoracle、チャートでは到達可能性（reachability condition）、ＢＵＰではlink述語、などと呼ばれて実現されていたものである。

或る非終端記号（仮に、bとしよう）を親（木の根のこと）とする構文解析木が見つかったとして、その非終端記号に対応する述語の呼び出しが起こったとしよう。この述語に対するタイプ１の述語（b1）の動作を考えてみよう。この述語は、新たな識別子を作って送り出すが、その時に、受け取った識別子のリストをその引数として埋め込む。新しく作られる識別子ひとつひとつは、bを左端の子として持つ解析木を作り始めることに対応している。従って、それぞれの識別子について、それがいずれどのような非終端記号を親とする木になるかはわかっている。b1が或る識別子（例えば、idx）を生成するとして、それに対応する木の親になる非終端記号をaと呼ぼう。b1は、入力として、識別子のリストを受け取る訳であるが、識別子は、特定の文法の特定の位置を表わしているから、その識別子に続く非終端記号は一意に決まる。このような非終端記号を、その識別子によって予測される非終端記号と呼ぼう。b1が、識別子idxを生成するためには、b1が受け取った識別子のリストの中に少なくとも一つはaを左端の子とする非終端記号を予測する識別子がなければならない。つまり、受け取ったリストの中にこのような識別子が含まれていないなら、aを親とする解析木の構築を試みても無駄、すなわち、idxを生成しても無駄という訳である。

例えば、今、限定詞が見つけられて、det1が呼び出されたと仮定しよう。det1は、id2という識別子を生成しようとするが、これは文法規則(2)に基づいて解析を始めることに相当している。この解析の結果いずれ得られる非終端記号は名詞句npである。つまり、det1にとっては、id2を出力するためには、入力引数に受け取った識別子のリストの中に、npを左端の子として持ち得る非終端記号を予測している識別子が少なくとも一つ存在しなければならないということがわかる。どの非終端記号がどの非終端記号の左端の子孫と成り得るかという関係をlink関係と呼ぶと、これは、文法規則全体が与えられれば事前に計算可能である。つまり、a --> b.... という文法規則があれば、link(b,a)という関係が成り立つことになり、link関係は、これらをもとにして、反射的かつ推移的な関係として定義できる。

以上の考察に基づいて、トップダウン予測を考慮した場合の変換結果を付録に示しておく。プログラム中、linkとid_pairという述語の定義は省略したが、link(det,np)は上でも述べたように、非終端記号detが、npの左端の子孫に成り得るという

関係を表わし、id_pair(id1,vp)は、識別子id1が動詞句vpを予測していることを表わしている。

tp_filter(X,np,New_X)は、識別子のリストXを受け取り、先頭の識別子から順に、それが予測している非終端がnpを左端の子孫とする親となり得ないものをリストから取り除いていく。tp_outは、こうしてできたNew_Xを受け取り、それが空の場合は、出力（第４、５引数）として空の差分リストを返す。New_Xが一つでも要素を含んでいれば、それは、いま対象としている非終端記号を上から予測している識別子が存在するということであるから適当な識別子のリストを出力する。プログラムを見てわかるように、これは、第２、３引数で与えられた差分リストを第４、５引数に渡すことによって行なわれている。tp_filterが、与えられた非終端記号のつながり得る識別子を探すだけではなくて、不要な識別子を捨てていることにも注意されたい。これにより、それ以降のtp_filterの処理の負担を軽減することに役立っている。タイプ１の述語は、二つ以上の識別子を出力することもあり、また、右辺がただ一つの要素から成る文法規則から得られたタイプ１述語は、左辺に対応する述語の呼び出しを直接行なったりする。これらそれぞれに対してtp_filterが用意されるべきであるが、同一の非終端記号を作り上げようとしているものは、tp_filterの出力を共用している。また、自分自身と同じ非終端記号を作ろうとしている場合、つまり、右辺の先頭の要素が左辺の要素と一致する文法規則に対応するものについては、tp_filterによるチェックは不必要なため省略されていることに注意されたい。

tp_filterについては、ここで示したように、積極的に不要な識別子を取り除くフィルターとして実現する方法と、初めて必要と思われる識別子が現れた時点でその先の処理を止め、リストの残りの要素を渡してしまう方法が考えられる。著者らの実験では、文法規則が大きい程、不要な識別子が生成されることが多く、積極的に不要な要素を取り除く前者の方法の方が効率改善に貢献するという結果を得ている。

４．ＤＣＧからの変換法

４．１　引数と補強項の処理

前節までは、文脈自由文法を対象にＳＡＸの変換法について述べたが、本節では、ＤＣＧで記述された文法規則を扱う方法と、そのときに考慮しなければならない問題点について説明する。

例を用いるのが最もわかり易いであろう。(g)から(k)に示した文法規則からの変換の結果、nounに関するものだけを取り出したのが、(l)から(o)である。

```
(g) np(FNP,np(DET,NOUN)) -->
        det(FDET,DET),noun(FN,NOUN),
```

```
        {det_n(FDET,FN,FNP)}.
(h) np(FNP,np(DET,NOUN,RELC)) -->
        det(FDET,DET),noun(FN,NOUN),
        {det_n(FDET,FN,NFN)},
        rel_clause(FRELC,RELC),
        {n_relc(NFN,FRELC,FNP)}.
(i) np(FN,np(NOUN)) --> noun(FN,NOUN),{np_n(FN)}.
(j) noun(FN,noun(NOUN,PP)) -->
        noun(FN1,NOUN),{mod_pp(FN1)},
        pp(FP,PP),{n_pp(FN1,FP,FN)}.
(k) noun(FN,noun(NOUN,PRED)) -->
        noun(FN1,NOUN),{mod_pred(FN1)},
        pred(FPR,PRED),{n_pr(FN1,FPR,FN)}.

(l) noun(X,A,B) --> noun1(X,A,B),noun2(X,A,B).
(m) noun1(X,A,B) -->
     ({mod_pp(A)},tp_out(X,[id4(X,B,A)|T1],T1):[]),
     ({mod_pred(A)},
        tp_out(X,[id5(X,B,A)|T2],T2):[]),
      {tp_filter(X,np,New_X)},
     ({np_n(A)},np(New_X,A,np(B)):[]),!.
(n) noun2([],_,_) --> !.
(o) noun2([id1(X,DET,FDET)|T],FN,NOUN) -->
        {det_n(FDET,FN,FNP)},
        np(X,FNP,np(DET,NOUN)),!,noun2(T,FN,NOUN).
(p) noun2([id2(X,DET,FDET)|T],FN,NOUN) -->
        {det_n(FDET,FN,NFN)},
        [id3(X,DET,NOUN,NFN)],!,noun2(T,FN,NOUN).
(q) noun2([_|T],A,B) --> !,noun2(T,A,B).
```

ＤＣＧでは、文脈自由文法の文法規則において、非終端記号を表わす述語に任意個の引数を含ませることと、本体部に任意のPrologの述語（これらを補強項と呼ぶ）の挿入を許している。

非終端記号が持つ引数については、ＳＡＸでも非終端記号は述語として表現されているので、そのまま同じ述語に持たせることができる。従って、ＤＣＧの記述においてｎ引数を持つ非終端記号は、入力リスト用の引数と出力用の差分リストを含めて、合計ｎ＋３引数を持つことになる。

ＳＡＸによる変換後のPrologプログラムでは、元の文法規則が幾つもの節に分解されてしまっている。具体的に言うと、文法規則の右辺に現れる終端および非終端記号それぞれに対して、一般に一つの節が作られる。従って、それらの間で共通に使われている変数については、何らかの手段で節を越えて受け渡しを行なわねばならない。そのため、前節までのように識別子が入力リストだけを運ぶのではなく、他の節で必要となる引数をも同時に運ぶように変更されている。各識別子が運ぶべき変数は、次のように決定される。識別子は、文法規則の固有の位置に割り当てられている。その識別子が割り当てられた文法規則の本体部の最後に頭部の述語を置いた列を想定し、その識別子が割り当てられた位置でその列を分離してみる。その場所に、補強項がある場合には、その補強項とそれに続く（非）終端記号との間で分離する。このとき、分離されてできた二つの列の両方で参照されている変数が、その識別子が運ばなければならない変数である。よって、このような変数の組み合わせは、識別子毎に異なり、変換時に自動的に決められる。

補強項については、文法規則内でその直前にある（非）終端記号に対応する変換後の節の本体部に置かれる。例えば、(g)に含まれている補強項は、(o)に現れており、(j)に現れる最初の補強項は、(m)の２行目に見られる。(m)の構造に注意してほしい。タイプ１の節は、文法規則の右辺の先頭に現れる同一非終端記号の処理を一手に引き受けるが、ＤＣＧのように、引数や補強項が個々に異なるような場合には、それぞれの識別子について独立に補強項の実行などを行なわねばならない。(m)によると、まず最初にmod_pp(A)の実行を行ない、成功すればtp_outによってid4が出力されるが、もし、mod_pp(A)の実行が失敗すれば、[] を通って、次のmod_pred(A)の実行に移る。一方、タイプ２の節は、（非）終端記号の文法規則の右辺における出現に１対１に対応しているから、補強項は、(o)や(p)に見られるように、その節の選択条件として使われている。

４．２　ＤＣＧの取り扱いに関する問題点

ＤＣＧで記述された文法規則をＳＡＸに変換する方法について、前節で述べたが、この変換法ではどのようなＤＣＧの記述も扱える訳ではなく、主に二つの制限がある。

その一つは、環境のコピーの問題、もう一つは、補強項の後戻りに関する問題である。一つずつ説明しよう。環境のコピーの問題とは、構文解析に曖昧性がある場合に起こる。例えば、(m)において、id4を出力する動作とid5を出力する動作は、本来、別の環境で行なわれなければならない。と言うのは、これらは、nounがまったく別の解析に用いられているのに対応しているからである。この例では、AとB二つの変数が、これらの異なる環境へ渡されている。もしこれらの変数が運ぶデータに未束縛の変数が含まれており、一方の環境でその変数に対して値の束縛が起こった時に他の環境へ影響を与えてはならない。この問題に対する一つの解決策は、受け渡しされるデータに変数が含まれることを禁ずることであり、もう一つは、解析

の途中で環境が分れる直前に変数のコピーを作ってそれぞれの環境に渡すことである。

補強項の後戻りの問題というのは、ＳＡＸで補強項の計算が一度しか行なわれないことに起因する。本来、ＤＣＧによる記述では、構文のみならず補強項などによって得られるデータなどにも曖昧性があることが許されている。PrologによるＤＣＧのトップダウン解析でも、ＢＵＰによるボトムアップ解析でも、Prologの後戻りを用いて、補強項の計算から得られる複数の解を抽出することができた。しかし、ＳＡＸによって得られるPrologプログラムは、決定的に動作し、決して後戻りを行なわない。従って、もし、補強項の計算が複数の解を持つ場合には、二つ目以降の解を得ることができない。

この問題についても二通りの解決策を考えている。一つは、補強項として、曖昧性のあるPrologプログラムの記述を禁ずることである。こうすれば、勿論、システムの変換法に何の変更も行なう必要がない。もう一つの方法は、freeze機能を持つProlog（例えば、Prolog II [Colmerauer 82]、ＣＩＬ[Mukai 85]など）によってＳＡＸを記述してしまい、freeze機能によって、補強項の計算のタイミングを遅らせることである。著者らは、この方法により、ＤＵＡＬＳ[安川 85]の構文解析部分をＣＩＬを用いて記述した。これにより、ＢＵＰによる実現に比べて、一桁以上の実行効率を得ている。これについては、別の機会に述べる[杉村 86]。この実現法では、実質上、構文解析と意味解析のタイミングをずらしていることになっている。これについての有利不利は詳しく考察しなければならない問題であるが、この実現法では、構文解析が終わり、freezeされていた補強項についての計算を行なう前に引数内の変数をコピーすることにより、環境コピーの問題も最小限の処理量で解決することができる。

５．おわりに

ＤＣＧによって記述された自然言語文法を処理することを目指した構文解析システムＳＡＸについて述べた。その特徴を振り返ってみよう。

まず、上昇型のアルゴリズムを用いることによって、下降型解析について生じる左再帰によるループなどを避けることができた。次に、副作用を用いず、しかも、同一構造の再計算を行なわない構文解析法を実現することができた。Prologで記述されているものの、後戻りが行なわれないので、コンパイラにより高速なコードを生成することが期待できる。

また、ＤＣＧによって記述されたどんな文法規則をも扱い得るようにするために考察しなければならない問題点について述べ、解決策について論じた。

謝辞

本システムの原形である並列構文解析に関する研究は、第一著者がロンドン大学

インペリアルカレッジに滞在中に行なったものである。研究の機会を与えられたProfessor Kowalskiを始めとするLogic Programming Groupの諸氏、また、英国滞在を援助していただいたブリティッシュカウンシルに感謝する。ＩＣＯＴにおいて引き続き研究を行なうにあたり、電総研およびＩＣＯＴの様々の方からの助言や励ましに感謝したい。特に、電総研推論システム研究室およびＩＣＯＴ第一、第二研究室の諸氏との議論とコメントに感謝する。また、著者が並列構文解析の研究を始めたきっかけは、東京工業大学田中穂積教授の示唆によるものである。記して感謝したい。

参考文献

[Colmerauer 82] Colmerauer, A.: Prolog II: Reference Manual and Theoretical Model. Internal Report. Groupe Intelligence Artificielle. Universite d'Aix-Marseille II. 1982.

[Earley 70] Earley, J.: An Efficient Context-Free Parsing Algorithm. Comm. ACM. Vol.13, No.2. pp.94-102. 1970.

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

[Matsumoto 83] Matsumoto, Y., Tanaka, H., Hirakawa, H., Miyoshi, H. and Yasukawa, H.: 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 Understandin and Logic Programming. Rennes. 1984.

[松本 86] 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.

[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.

[Mukai 85] Mukai, K., Yasukawa, H.: Complex Indeterminates in Prolog and its Application to Discourse Model. New Generation Computing. Vol.3. pp.441-466. 1985.

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

[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、情報処理学会第３３回全国大会、1986.

[安川 85] 安川秀樹ほか：談話理解システムＤＵＡＬＳの概要、ＩＣＯＴ TM-118. 1985.

付録

文法例より得られた変換後のPrologプログラム（トップダウン予測を含む）

```
np(X) --> np1(X),np2(X).
vp(X) --> vp1(X),vp2(X).

np1(X) --> {tp_filter(X,sentence,New_X)},
     tp_out(New_X,[id1(New_X)|T1],T1),
     tp_out(X,[id5(X)|T2],T2).
np2([]) --> !.
np2([id6(X)|T]) --> np(X),!,np2(T).
np2([id8(X)|T]) --> vp(X),!,np2(T).
np2([_|T]) --> !,np2(T).

vp1(X) --> tp_out(X,[id9(X)|T],T).
vp2([]) --> !.
vp2([id1(X)|T]) --> sentence(X),!,vp2(T).
vp2([id10(X)|T]) --> vp(X),!,vp2(T).
vp2([_|T]) --> !,vp2(T).

det(X) --> {tp_filter(X,np,New_X)},
     tp_out(New_X,[id2(New_X),id3(New_X)|T],T).

noun([]) --> !.
noun([id2(X)|T]) --> np(X),!,noun(T).
noun([id3(X)|T]) --> [id4(X)],!,noun(T).
noun([_|T]) --> !,noun(T).

rel_clause([]) --> !.
rel_clause([id4(X)|T]) --> np(X),!,rel_clause(T).
rel_clause([_|T]) --> !,rel_clause(T).

coconj([]) --> !.
coconj([id5(X)|T]) --> [id6(X)],!,coconj(T).
coconj([id9(X)|T]) --> [id10(X)],!,coconj(T).
coconj([_|T]) --> !,coconj(T).
```

```
that(X) --> {tp_filter(X,rel_clause,New_X)},
     tp_out(New_X,[id7(New_X)|T],T).

sentence([]) --> !.
sentence([begin|T]) --> [end],!,sentence(T).
sentence([id7(X)|T]) -->
        rel_clause(X),!,sentence(T).
sentence([_|T]) --> !,sentence(T).

verb(X) --> {tp_filter(X,vp,New_X)},
     tp_out(New_X,[id8(New_X)|T],T),
     vp(New_X).

the(X) --> det(X).
man(X) --> noun(X).
woman(X) --> noun(X).
walks(X) --> verb(X).
loves(X) --> verb(X).
and(X) --> coconj(X).

tp_filter([],_,[]) :- !.
tp_filter([ID|Rest],Nonterm,[ID|Rest1]) :-
     functor(ID,Id,_),id_pair(Id,Expected),
     link(Nonterm,Expected),!,
     tp_filter(Rest,Nonterm,Rest1).
tp_filter([_|Rest],Nonterm,New_list) :-
     tp_filter(Rest,Nonterm,New_list).

tp_out([],_,_,Y,Y).
tp_out([_|_],Y,Yt,Y,Yt).
```

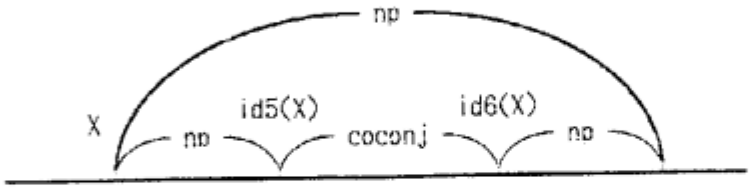

図１　識別子の受け渡しと解析木の構成

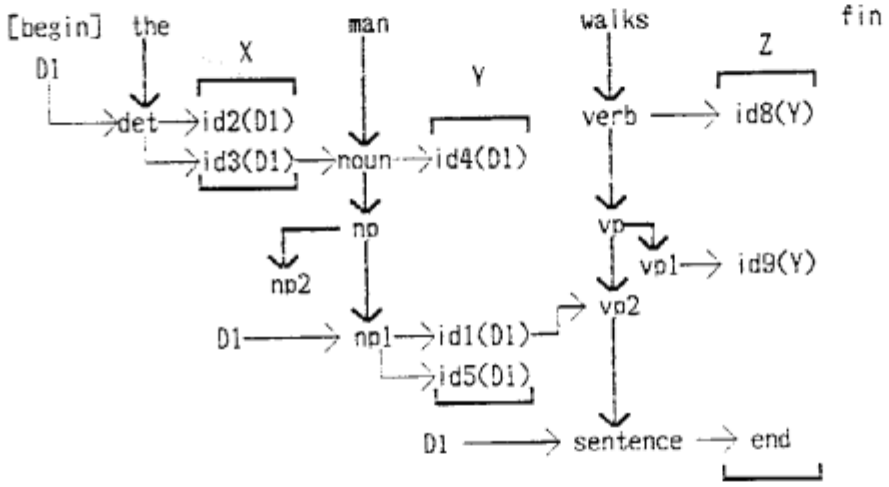

図２　ＳＡＸによる文の解析例