数理解析研究所講究録
213巻 1974年 153-165



(電子通信学会(1974年1月23日
資料より転載)

資料番号AL73-71

# 自然言語分析のためのプログラミング言語－PLATON－とそれによる名詞句の分析

A PROGRAMMING LANGUAGE FOR NATURAL LANGUAGE ANALYSIS AND ITS APPLICATION FOR JAPANESE NOUN PHRASE ANALYSIS

長尾 真　辻井潤一

MAKOTO NAGAO　JUN-ICHI TSUJII

京都大学工学部

KYOTO UNIVERSITY

A Programming LAnguage for Tree OperatioN -PLATON- is constructed. This accepts strings, trees and lists and transforms them in arbitrary way. Based on the argumented trasition network of W. Woods, this has the additional capability of matching strings, trees and lists, variable assigning, transformational rewriting. For an efficient rule application, the pre-checking of the applicability of a rule is introduced, and also the exit destination can be specified when a return is made from a lower state to an upper state. This avoids a tedious time-consuming back-tracking process.

An example of using PLATON is given, which analyzes Japanese noun phrases containing postposition 'NO'

## 1. はじめに

W.A.Woodsらのaugmented transition networkは、push-down automatonに種々の拡張機能をつけることによって、Turing Machineと同等の認識能力を持たせたモデルであるが、自然言語の認識機構としても、次のような幾つかのすぐれた特長を持っている。

(1)任意のtransformational grammarを記述できる。
(2)ルールの追加，変更，削除が容易なため、実験システムに向いている。
(3)ルールを表として与えるより効率が良い。
(4)任意の条件checkをLISP関数として組込むことが可能であるから、種々の意味論モデルとの結合が可能である。
(5)文法の明晰さが保存されている。

しかしながら、Woodsらのモデルでは入力シンボルを順次読込むことによって状態遷移をおこなうため、語順が比較的自由な日本語文に適用するには困難があると考えられる。日本語においても、意味のまとまりの単位である句内の語順はある程度一定している。しかしながら、句間の関係は、主として英語では語順によって表現されるのに対して、日本語では格助詞によって示めされるため、語順自体にはかなりの自由度がある。

したがって、英語の解析の際に、しばしば用いられてきた、入力文章の単語を先頭から順々にscanして解析をすすめてゆく方法は日本語に対しては不適切であると考えられる。

日本語の解析においては、pattern-matching的な手法で処理単位を切り出し、その処理単位内での構造を解析し、その結果の構造を、さらに大きな構造に組み上げてゆくといった方法が有効であると考えられる。

我々はこの様な考え方を基本として、自然言語(日本語)処理用のプログラミング言語PLATON(Programing LAnguage for Tree OperatioN)を開発した。

一般に言語解析にはTop-Down的手法と、Bottom-Up的手法があるが、networkモデルは本質的にTop-Down的手法になっている。Top-Down的手法では、ある構造の存在を予測して、対応するルールを適用してゆくわけであるが、そのルールが不適切であった場合、それをどのように回復するかが問題となる。失敗した原因を知ることによって、適切なルール適用に切換るメカニズムをモデルは備えていなければならない。Woodsらのモデルでは状態内のルールの順位によって、このメカニズムをimplicitに表現しているが、前のルール適用失敗の原因によって処理の流れを柔軟に変化させるには不十分であると考えられる。PLATONでは失敗原因をfailure-messageとして返すことによって、より柔軟な処理が可能になっている。

この資料では、PLATON言語の各種機能と、その仕様を解説し、PLATONを使用して書かれた具体的なプログラム(名詞を中心とした句の処理)について説明する。なお、PLATONは記号処理用言語LISPを用いてcodingされている。

## 2. Tree構造とList構造

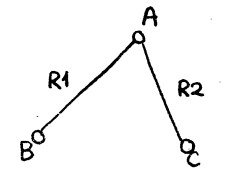
この節ではPLATONが使用する基本的なデータ構造について述べる。

自然言語の処理においては、自然言語文の解析結果をtree構造として表現することが普通おこなわれる。そこでPLATONでtree構造をどの様に表現するかを次に述べる。

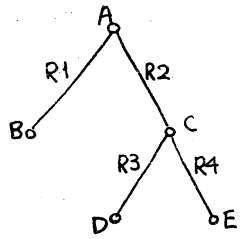
[ PLATONにおけるTree構造の定義 ]

① <tree> ::= <node> | (<node><branches>)
② <branches> ::= <branch> | <branch><branches>
③ <branch> ::= (<relation><tree>)
④ <node> ::= 任意のLISP原子記号
⑤ <relation> ::= 任意のLISP原子記号

以上の形式的定義を、具体的なtree構造とその表現形式で示すと(図-1)になる。




(A (R1 B) (R2 C))




(A (R1 B)
(R2 (C (R3 D)
(R4 E))))

<図-1> Tree構造と表現形式

自然言語の解析は仮え的な単語配列(list)から、それに対応するtree構造(parsing tree)を作り出す過程であると考えられる。その途中段階では、すでに解析が終ってtree構造に整理された部分と、まだlistのままの部分とが混在することになる。そこでtree構造とlist構造の混在を許した表現方法が必要になる。このtree構造とlist構造の混在を許したdata構造は我々のシステムの基本となるものである。以下このようなデータ構造を単に構造という。

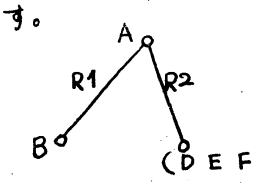
次にこの構造の形式的定義を与えるが、tree構造中のnode自体がlist構造になることをゆるすために、先の定義の④が変更される。

[ PLATONにおける構造の定義 ]

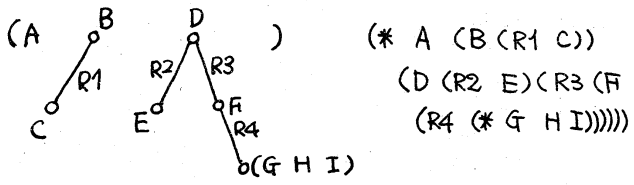
⑥ <structure> ::= <tree> | <list>
⑦ <list> ::= (* <structures>)
⑧ <structures> ::= | <structure> | <structure><structures>

④' <node> ::= 任意のLISP原子記号 | <list>

具体的な構造と、その表現形式を<図-2>に示す。




(A (R1 B)
(R2 (* D E F)))




（図－２）　構造と表現形式

## 3. パターン・マッチングの機能と方法

自然言語の文法の各ルールには、そのルールを適用するべき‘構造のpattern’がある。すなわち、書換法則の左辺にあたる部分がある。PLATONでは、この部分を変数を含む構造によって表現する。そして、この変数を含む構造と処理対象である具体的な構造とのmatchingをとることによって、そのルールを適用すべきpatternがあるかどうかを判断する。さらに処理対象の構造内での変数に対応する部分を切り出して、その部分を他のルールで処理するように指示することも出来る。

次のような変数が、2で定義した<structure>の中で使用出来る。

［PLATONにおける構造変数］

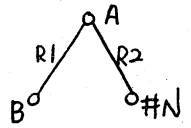
#N数字；　任意のnodeとmatching可能。
#P数字；　任意のstructureとmatching可能。
#B数字；　任意のbranchとmatching可能。
#K数字
#I数字
#J数字
；　任意個のlist要素とのmatching可能。

(*)数字は任意の自然数である。

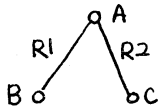
PLATONのシステム関数MATCHは上記のような変数を含んだstructureと、任意のstructureとのmatchingをとり、matchingの成否を値としてかえす。その際matchingが成功した時には各変数が具体的な構造の中で、どの部分と対応がとれたかを、ドット対にしてかえす。具体的な例でこのことを示す。(<図－３>)

◦例1◦

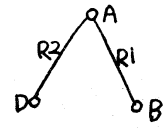
<変数を含むstructure>



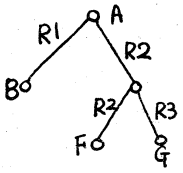

<具体的なstructureとmatchingの結果>




成功；((#N. C))



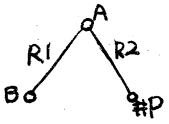

成功；((#N. D))



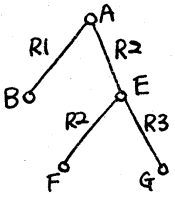

失敗

◦例2◦

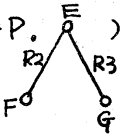
<変数を含むstructure>

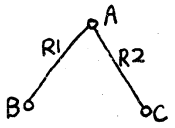



<具体的なstructureとmatchingの結果>




成功；((#P. E ))

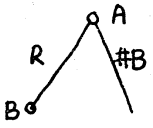


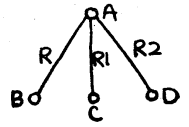

成功；((#P. C))

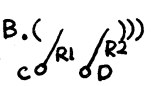
◦例3◦

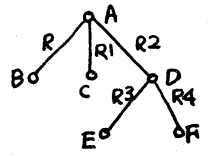
<変数を含むstructure>

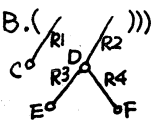



<具体的なstructureとmatchingの結果>



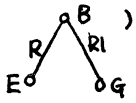

成功；((#B.(  ))



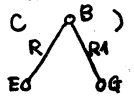

成功；((#B.(  ))

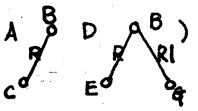
◦例4◦

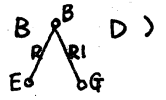
<変数を含むstructure>

(＊ A #K  )

<具体的なstructureとmatchingの結果>

(＊ A B C  )　成功；((#K.(＊ B C)))

(＊A  )　成功；((#K.(＊  D)))

(＊A B  D )　失敗

<図－3> matchingの例1～4

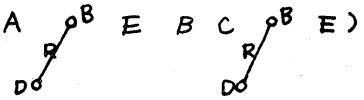
もちろん、上記変数の複数個が1つの構造の中で使用されてもかまわない。また同じ変数を2度以上使用することによって同一の構造が離れて存在することを表現することも可能である。<図－4>にこの例を示す。

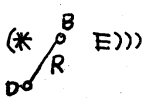
◦例5◦

<変数を含むstructure>

(＊ A #K B C #K)

<具体的なstructureとmatchingの結果>

(＊A  E)

成功；((#K.(＊  E)))

(＊A D E B C D G)

<図-4>matchingの例5　失敗

tree構造内のbranchにはすべてラベルがついているために、branchの順序が違ってもmatchingは成功する。

ただし、list変数(#K, #I, #T)の使用には、連続して使用出来ないという制限があり、

(＊ #K #I A)

は不可である。

pattern-matchingのための関数MATCHは、指定された構造パターンがどんなに深いレベルにあっても、見つけ出すことが出来るようになっている。

## 4. 状態遷移とルールの表現法

プログラムで文法を記述するのと比較して、一様なsyntaxを持ったルールで文法を記述することの利点としては、

(1) ルールの変更，追加が容易である。
(2) 文法の明快さ，明晰さが保存される。

がある。計算機による言語処理の立場から見れば、現在ある自然言語のすべての文法は不完全であると考えられる。したがって、その都度，変更，追加をすることによって、よりすぐれた計算機のための文法を作ってゆかねばならないと思われる。このことから、種々の文法を計算機にimplementし、testし、簡単に改良できるようなsystemが必要になる。(1)，(2)の特徴はこのようなsystemが是非とも備えていなければならないものだと考えられる。

逆に一様なルールの集合を基本とする解析システムの欠点としては、

(1) 柔軟な処理，特にsemanticsとの結びつきを表現することがむつかしい。
(2) 複雑で高度な処理をしようとすると、例外的な文章を扱うためのルールが増加して、効率がわるくなる。

特に(1)の欠点のために、従来の解析システムではsyntax analysisの部分を一様なルールで処理し、その結果を独立したsemantic analyzer

に引渡すという手法をとっていた。

しかしながら、実際にはsyntax analysisと、semantic analysisとは交互に、相互依存的に働く必要がある。この欠点を一様なルールを使用するシステムでは解消する必要がある。

(2)の点に関しては、Woodsらのtransition network modelでは、例外的なルールは例外的な状態(state)に置かれているために、ルール数の増大が必ずしも効率の著しい低下に結びつかない利点がある。

我々のシステムではWoods流のtransition network modelを採用し、一様なgrammarの持つ利点を保持しながら、userが自由にプログラムを記述する部分をrule formatの中に設けることによって、柔軟な処理も表現出来るようになっている。

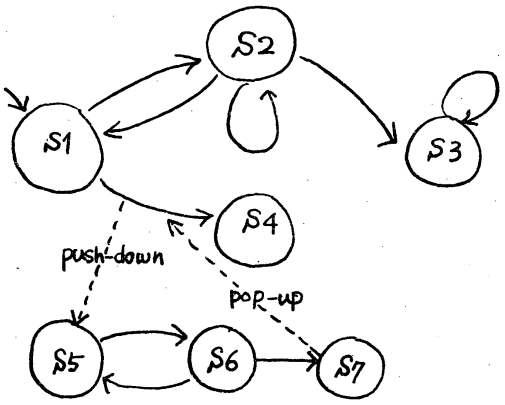
## <4.1>PLATONのルール記述の定義

PLATONの基本的なモデルをまず説明する。

モデルの基本的な部分は、複数個の状態(state)とそれらを結びつける方向性を持った枝(branch)からなるnetworkである。各branchは1個の書換ルールに対応しており、そのルールを使用し得るための条件、そのルールが使用され得るための構造パターン(書換ルールの左辺)、及びそのルール適用後の構造(書換ルールの右辺)が示されている。そして、あるbranchに対応するルールを適用した後は、そのbranchが入ってゆくstateに状態を移し、そのstateに付いたルール(branch)は、再び同じように適用してゆく。(<図-5>参照)

各stateからは複数個のbranchが出ており、各branchは優先度にしたがって一列に並べられている。あるstateにcontrolが移ってくると、システムはそのstateから出てゆくbranchを優先度にしたがって順々にcheckしてゆく。まずbranchについた条件をcheckし、処理対象となる構造がそのbranchについた構造パターンを満足するかどうかを検査し、満足しておればそのルールを適用して構造を変換し、次のstateへcontrolを移す。

PLATONでは、上述の基本的な機能の他に、push-down automaton的な機能が付加されている。即ち、branchについた構造パターンと処理対象の構造とのpattern matchingによって切り出された特定の部分構造だけを、他のstateへ移って処理をする。(push-down) そして、その部分の処理が終ると、push-downの生じたbranchに戻って処理を続けてゆく。(pop-up) この能力によって、特定の部分を切り出して、その部分の処理をし、その処理結果をより大きな構造の一部分として埋め込むことが出来る。




<図-5> 状態遷移とpush-down, pop-up

branchには通常のstate-transitionを示すbranchの他に、処理の終了を示すbranch, 処理が失敗したことを示すbranchがある。これを整理して示すと下記のようになる。

(i) NEXT-branch ; 指示されたstateに状態を移して処理を進める。
(state-transitionのためのbranch)

(ii) POP-branch ; そのlevelの処理が終了して、指定された構造を持ってpush-downの生じた枝に戻る。もし、現在が最高levelの処理であれば、全体の処理が終了したことになる。

(iii) FAIL-branch ; そのlevelの処理が失敗したことを示すbranch。指定されたerror-messageを持ってpush-downのおこった枝に戻る。

ここで(iii)のbranchはWoodsのモデルにはないものである。transition networkのモデルは、状態を移すことによって使用する規則を制限するという意味でtop-down的な手法である。また、特にPLATONでは、pattern matchingによって特定の部分構造を切り出し、その部分に特定の構造があるべきだと仮定して、その処理を行う。したがって、その仮定が誤っていた場合の処理が問題となる。

そこで、PLATONでは仮定が何故不合理であったかを(iii)のbranchでerror-messageとして上位levelに返し、上位の処理ではerror-messageを解釈して、適当なstateへ処理のcontrolを移すことが出来るようになっている。

下にPLATONプログラムの形式的な定義を与える。

[ PLATONプログラムの形式的定義 ]

<PLATON program> ::= ( <states> )
<states> ::= <state> | <state><states>
<state> ::= (<state name> <rules>)
<state name> ::= 任意のLISP原子記号
<rules> ::= <rule> | <rule><rules>
<rule> ::= (<pcon> <strx> <con>
(<trans>) <act> <end>)
<trans> ::= <transit> | <transit><trans>
<transit> ::= ((<state name> <structure1>
<vari>) <errorps>)
<vari> ::= <local variable> | <register name>
<local variable> ::= #N | #N1 | #N2 ·····
#P | #P1 | #P2 ·····
#B | #B1 | #B2 ·····
#K | #K1 | #K2 ·····
#I | #I1 | #I2 ·····
#J | #J1 | #J2 ·····
<register name> ::= /任意のLISP atom
<errorps> ::= | <errorp> <errorps>
<errorp> ::= (<error-message> <act>
<pro>)
<error-message> ::= 任意のLISP原子記号
<pro> ::= (EXEC (<trans>)) | (TRANS
<state name>)
<end> ::= (NEXT <state name> <strY>)
| (POP <strY>)
| (FIN <error-message>)
<act> ::= (SETR <register name> <structure1>)
| (SENDU <register name> <structure1>)
| (SENDD <register name> <structure1>)
| (GETR <register name>)
| (GETU <register name>)
| (GETD <register name>)
<strY> ::= <structure2> | /
<strx> ::= <structure1> | /
<pcon>, <con> ::= LISP論理関数 & びuser定義のLISP関数

(注) <structure1> は3で定義した変数名を、2で定義した<structure>の中で使用を許した構造。 <structure2> は変数名以外に、register名を使用した構造をいう。

上記の定義から解るように、<rule>は、<pcon> <strx>, <con>, <act>, <trans>, <end>からなる6字組によって表現される。以下の<4-2>, <4-3>, <4-4>においては、これら各partの意味と使用法について説明する。

**<4-2> <pcon>, <con>, <act>について**

前述したように、柔軟な処理(例えばsyntaxとsemanticsの相互参照的な処理)を行うためには、ルールの中にuserが自由に記述できるプログラム的な部分が必要である。これを保障するのが<pcon>, <con>, <act>の各partである。

ルール適用の結果をすべて構造の中に埋め込んで処理を進めてゆくのは、処理やルールをいたずらに複雑にするだけであろう。そこで処理の中間結果をレジスタに保持しておいて、他のルールからそのレジスタを参照することが考えられる。即ち、ルール間, state間の情報交換をすべて処理対象のtree-list構造を通じておこなわず、レジスタによってもおこなうわけである。レジスタのセットは主として<act>で行われる。

現在用意されているレジスタ用の<act>は、

<4-1>の定義で述べたSETR, SENDU, SENDD, GETR, GETU, GETDである。

処理途中でpush-downがおこると、その時点での処理の流れから飛び出し、全く別のstateから部分構造の処理が始まる。この部分構造の処理は全体のcontrolの流れからはlevelが1段下の処理であると考える。このようにpush-downがおこると、controlが1段下の処理に移り、pop-upがおこると、1段上位のlevelにcontrolが戻ってくると考える。SETRは現在処理中のlevelのレジスタをセットし、SENDUは1段上の, SENDDは1段下のレジスタに値をセットする関数である。同様に、GETR, GETU, GETDの各関数は、それぞれ現在の処理levelと同じlevelのレジスタ, 一段上位のlevelのレジスタ, 一段下のlevelのレジスタの値を引出すための関数である。

この様に、レジスタには処理のlevelが付いており、<register name>が同じであっても処理levelの異るレジスタは別の物であるとみなされる。このレジスタを使用することにより、state間の情報交換, 上下level間の情報交換が円滑におこなわれる。

<pcon>, <con>としては、PLATONそのものはなにも関数を用意していない。特に<con>はuserが意味モデルとして、どの様なモデルを採用するかによって、それに適した関数を、LISP関数として定義する必要がある。5.において、PLATONを使用して書かれた解析プログラムの具体な例として、'名詞句の解析'を行うプログラムを説明する。そこでは、<con>に使用される幾つかのLISP関数が説明されることになる。これらの関数は、'名詞句の解析'に際して、採用した意味モデルを扱うものであって、PLATON固有の関数ではない。

<4-3> <strx>, <stry>について

<strx>はbranchについたルールの, 書換ルールに対応していえば, 左辺にあたる構造パターンを規定する部分である。<strx>には<structure1>か/が書かれる。<structure1>は<4-1>で述べたように、変数を含んだ構造パターンである。PLATONシステムは、この<strx>に書かれた構造パターンと、現在処理中の構造とのpattern-matchingをおこない、構造パターン<strx>中の変数が、処理中の構造のどの部分に対応するかを知って、変数の結合を行う。/は、このpattern-matchingをおこなわないことを示す記号である。

<stry>は書換ルールの右辺にあたる構造パターンを規定する部分である。<stry>には<structure2>か/が書かれる。<structure2>は、<structure>の適当な部分を変数名あるいはレジスタ名にした構造パターンである。例えば、(* A #K /REG)と書くと、#Kの部分には、<strx>とのpattern-matchingの際に変数#Kとbindingされた部分構造が代入される。/REGの部分には、現在の処理levelのレジスタ名がREGであるレジスタの内容が、代入される。<end> partが(NEXT <state-name> <stry>)であれば、こうして代入された構造を持って、指定されたstateにcontrolが移る。(pop <stry>)であれば、controlは1段上のpush-downの生いたbranchに戻るが、その際<stry>によって指定された構造を持って戻る。このように<structure2>の中に、(/<register-name>)の表現を許すことによって、中間の処理結果をレジスタに保存し、構造の任意の場所にそれを埋込むことが可能になる。

<end> partが、(FM <error-message>)の場合は、指定の<error-message>を持ってpush-downの生いた枝へ戻る。

<4-4> <trans>について

<trans>部分は切出した部分構造を全く別のstateから処理を始めるように指示する部分である。例えば、<trans>部分が、

(((S1 #K #K))((S2 (* #I #J)./REG)))

であるとする。ここで#K, #I, #Jは<strx>中に使用された変数である。controlがこの部

分にくると、push-down がおこり #K に対応する部分を state S1 から始まる処理に、(* #I #J) に対応する構造を state S2 から始まる処理にそれぞれ手渡すことになる。そして、それぞれの処理が pop-up によって終ると、その処理結果をそれぞれ #K, レジスタ REG に入れる。

このように push-down, pop-up をおこなうことにより部分構造の処理をするのであるが、この部分構造の処理が失敗した時には pop-up ではなく、FM-branch によって error-message が上位の branch に戻ってくる。そこで <trans> の部分に、この error-message を解釈し、適当な処理に control の流れを変えるように指示することが出来る。例えば <trans> 部分が、

```
(  ((S1 #K #K) (ERR1 (EXEC ((S5 #K #K)
                             ((S6 (* #I #J) /REG))))
                (ERR2 (TRANS S8))
                (ERR3 (EXEC ((S7 #K #K))))
   ((S2 (* #I #J) /REG))   )
```

であるとする。#K は push-down が生じて、state S1 で処理される。この処理が戻ってくる戻り方には3種類ある。

(1) Pop-Up で戻る。(Pop-branch で戻る)
(2) error-message で戻る。(FM-branch で戻る。)
(3) NIL で戻る。(適用すべきルールがなくなり、処理が行き詰って戻る。)

push-down を生じた上位の branch では、この(1)(2)(3)に応じて次のような処理をする。

(1)の場合; #K の処理は成功し、したがって次の処理(S2で(* #I #J)の処理)に進む。
(2)の場合; 帰ってきた error-message の解釈をする。すなわち、error-message が ERR1 であれば、S5 で #K, S6 で (* #I #J) の処理をする。又 error-message が ERR2 であればこのルール適用全体を放棄して、全体の処理を state S8 からやり直す。また、対応する error-message がない(ERR1, ERR2, ERR3 以外の error-message が戻ってきた)時は、このルールの適用は失敗したことになる。
(3)の場合; このルールの適用は失敗したことになる。

なお左側の例では、S2 で (* #I #J) の処理をおこなうが、この処理には error-message 解釈部分がないので、(2)の戻り方をしても対応する処理がない。したがってこの場合は NIL で戻ってきたのと同じことになる。

control があるstate の処理に入ってくると、そのstate から出ているbranch のルールが順々に適用可能かどうか check される。適用可能なルールがあれば、そのルールを適用し、そのルールの指定する state に control が移る。NEXT-branch であれば、指定された state に control が移り処理の level は変らない。Pop-branch であれば、push-down を生じた branch へ control が戻り処理 level は一段上る。FM-branch の場合は指定された error-message を持って push-down の生じた branch に control が戻り、この場合も処理 level は一段上がる。

その state から出てゆくルールが、すべて現在の処理対象に対して適用不可能な場合には、この処理 level での処理は失敗したことになり、NIL を持って push-down の生じた branch へ戻る。現在の処理が最高 level での処理の場合には処理全体が失敗したことになる。

この <trans> の部分は、例えば左側の例では、#K と (* #I #J) の処理をそれぞれ S1, S2 から始まる処理に手渡すというように、user にとっては parallel に処理をしていると考えてプログラム出来る。もちろん実際には #K の処理を終了してから (* #I #J) の処理に進むわけで、このことを考慮に入れてプログラムすることも可能である。

## 5. 具体的な解析プログラム

現在、我々は中学校理科の教科書を対象として文章分析のプログラムを作成中であるが、この章では名詞句の処理方式とそれがPLATONにおいてどのようにルール化されるかを説明する。

PLATONで書かれる言語処理プログラムには基本的には2つのpartがある。

(1) <strx>, <stry>で表現される構造変換をおこなうpart；これは X⟶Y という通常の書換ルールを表現するpartで主として文章のsyntax処理に関係する。

(2) <con>, <act> etc で表現される意味checkのpart；ここにはuserが採用する意味論モデルに対応するcheckが行われる。

従来の言語分析プログラムにおいては、文章の持つ意味(semantics)と文章の構造(syntax)との相互依存性が必ずしも十分にとり入れられてはいなかった。これには意味が計算機にかかるほど精密に定義されにくいという理由の他に、文章構造の解析途中の適切な個所で意味処理をおこなう(syntax処理とsemantic処理の)interactiveな処理プログラムを書くことが非常に困難であるという理由が考えられる。このためプログラムは全体のsyntax処理を終了して後，semantics処理に移るというシリアルな処理がおこなわれた。しかしながらPLATONにおいては、

(1) 上記の(1)，(2)のpartが分離しているため意味分析のプログラムは独立してプログラム出来る。

(2) PLATONの1つのルールの適用は上記(1)，(2)の処理結果によって行なわれることから、syntax処理とsemantics処理を独立してプログラム出来，かつルール適用に際してはそれらの処理が同時に行なえるという利点を持っている。

### <5－1> 意味的処理

現在までの文章分析においては「名詞＋の＋名詞」型の句の処理は困難な問題であり、ほとんど手がつけられていないといってよい。しかしながら分野を限って、その分野に関する知識を計算機に与えておいてやれば、これらの処理もある程度可能になると思われる。辞書の中には、動詞がどの様な格(Case)関係を支配するかについて記述するだけでなく、名詞についてもかなり詳しく記述しておく必要がある。

「形容詞＋名詞」型の構文も'(名詞(MODIFY 形容詞))'の形式に整理するだけでは十分な意味解析とはいえない。また、「形容詞＋名詞＋の＋名詞」型構文において形容詞がどちらの名詞を形容しているのかを解析するには、名詞と形容詞の関係を辞書内に規定しておく必要がある。

我々はこういった名詞を中心とした構文を解析するために、名詞－名詞間，形容詞－名詞間の関係を記述した辞書を作成し、その辞書を検索する基本的な意味処理関数を用意した。そしてこの基本的な関数をPLATONのルールの<con>部分に使用することにより、辞書で示めされた意味モデルとsyntax処理とを結合した解析プログラムを作成した。

### (A) 辞書の構成

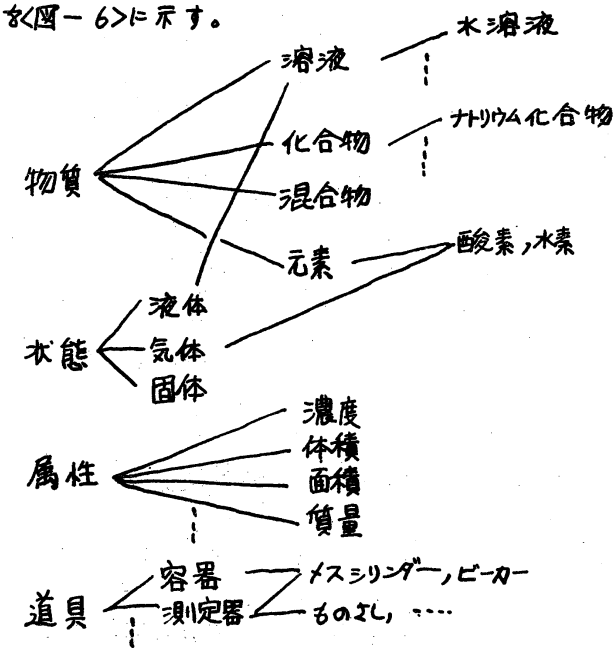
我々は日本語文の処理は'動詞中心の処理'が有効であると考え、教科書中の動詞について各動詞の格関係(Case-Frame)を記述した辞書を構成し、それを利用した文章分析プログラムをPLATONによって作成中であるが、ここでは名詞句の処理を中心にして、名詞の辞書構成について述べる。

名詞辞書のentryには次の各要素が記述される。

(1) 上位概念
(2) 下位概念
(3) 属性　(4) 部分
(5) 直接かかり得る形容詞
(6) 他の名詞との結びつき。[(1),(2),(3),(4)以外の]

(1),(2)は従来の名詞のcategory分類と同じ役割

を果すものであるが、この分類は階層的な構造を持っている。教科書を対象として分類の一部を<図－6>に示す。




<図－6>名詞の階層構造

'物質の質量''物質の体積''物質の温度'etc中の質量，体積，温度の名詞は'物質'につき得る属性であると考えられる。したがってこれらの名詞は'物質'の辞書entry中の(3)の項目中に入れられる。又'人間の手''人間の足'etc中の足，手は'人間'の部分を示す名詞であり(4)の項目中に入れられる。この様に1個の名詞は意味分類という見方から上位概念，下位概念の名詞と関連し、その名詞の持つ属性，部分という見方から(3)(4)の名詞と関係する。

例えば、

重い物質 ——→(質量が重い)物質

あたたかい物質 ——→(温度があたたかい)物質

大きな物質 ——→(体積が大きな)物質

のように、'重い''あたたかい''大きな'の各形容詞は'物質'を直接形容しているのではなくて、'物質'の'質量','温度''体積'という各属性をそれぞれ形容していると考えられる。したがってこれらの形容詞は、'物質'の辞書エントリの(5)ではなく、それぞれ'質量''温度''体積'の辞書エントリの(5)に入っている。

次に、具体的な辞書entryの一例を示す。

<例>

液体(M)

| | |
|---|---|
| superset | 物質，状態 |
| subset | 溶液，水，アルコール |
| 属性 | (体積 MODT)<br>(質量 MODM)<br>(温度 MODθ) |

溶液(M)

| | |
|---|---|
| superset | 液体，混合物 |
| subset | 水溶液，…… |
| 属性 | (濃度 MODN) |
| 部分 | (溶質 物質)<br>(溶媒 液体) |

体積(M)

| | |
|---|---|
| superset | 属性 |
| subset | |
| 直接かかる形容句 | MODT |

MODT(K)

| | |
|---|---|
| superset | MOD |
| subset | 大きい，小さい<br>数十cc，数十cm³ |

濃度(M)

| | |
|---|---|
| superset | 属性 |
| subset | |
| 直接かかる形容句 | MODN |

MODN(K)

| | |
|---|---|
| superset | MOD |
| subset | 濃い，うすい<br>数十% |

(※) Mは名詞，Kは形容詞を示す。この記法は以下でも使用する。

(B)意味処理関数

上記の辞書を検索して、名詞間の意味関係をcheckする関数として次の様な関数が用意されている。

(1)ATR；M1∈ATR[M2]の関係をcheckする。即ち、名詞M1が名詞M2の属性語かどうかをcheckする。

例えば、M1:=濃度，M2:=溶液の場合には、この関係が成立する。

(2)PW；M1∈PW[M2]のcheck。名詞M1が名詞M2の部分を示す名詞かどうかをcheckする。

(3)ATRV；M1∈$\overset{T}{\text{ARV}}$[M2]のcheck。例えば、M2:=溶液の時，溶液の属性としては、濃度，体積，温度etcが考えられる。そして、属性'濃度'の値としては'濃い''うすい''80%'，属性'体積'の値としては'大きい''小さい'etcがある。このように名詞M1が名詞M2の'属性の値'に

になっているかどうかをcheckする関数がATRVである。

(4) PWV: M1∈PWV[M2] をcheckする。水は、溶液の'部分'である溶質，溶媒のうち、溶媒の'値'となり得るから

水∈PWV[溶液] が成立する。

以上4つの関数を用いて名詞句処理のプログラムが書かれる。

## <5-2> 処理の考え方

名詞を中心とした構文を処理する方式として、「M+の+M」型の処理と、「(M+の)+K+M」型の処理の2つに分けて説明し、次にこれらの各処理を統合して全体の処理がどのように流れるかを説明する。

### (A)「M+の+M」型の処理

この型には構文中の名詞の種類によって基本的に次の4種類の処理がある。

(1)機能語の処理

(M1+の+M2)型の句において、M2にある特定の名詞がくることによって、1つの定まった意味の変換，付加をM2がM1に対しておこなう。

(例) 水の粒子の間，物質の中，物質の一部，化学変化の前後

これらの機能語の特長は、これらの語の前後で意味的なまとまりがあるということである。即ち、語の前の部分を独立に切り出して処理し、その処理結果に対してこれらの語が作用すると考えて処理する。これはPLATONのパターンマッチング機能を用いれば簡単に実現できる。

(2)名詞辞書を利用し、名詞間の関係で処理する場合

M1とM2の関係として次の様な場合が考えられる。

(i) M2∈ATR[M1]

(例) 溶液の体積，物質の比熱

この場合は、M1とM2は結合されて(M2(¥ATR M1))の形に整理される。

(ii) M2∈PW[M1]

(例) 溶液の溶質，化合物の構成元素

処理は(i)の場合に準じる。

(iii) M1∈ATRV[M2]

(例) 100ccの水，赤色の溶液

この場合M1とM2は直接つながっているのではなく、M2の属性(例えば体積，色)の値としてM1がつながる場合である。処理結果は、それぞれ(水(体積 100cc))，(溶液(色 赤色))となる。

(iv) M1∈PWV[M2]

(例) 塩化ナトリウムの水溶液，銅の化合物

(iii)の場合に準じる。処理結果は(水溶液(溶質 塩化ナトリウム))etcとなる。

(3) M2がActive要素を持つ場合

名詞の中には「する」がついて動詞化する名詞がある。このような名詞はサ変名詞と呼ばれ、この名詞が単独で使用される時も、やはり目的格，主体，道具etcの動詞的要素を支配することが多い。理科教科書中でのこれらの名詞としては、'分解''変化''検出''合成''実験'etcがある。これらの名詞の辞書エントリには、その動詞化に対する格支配関係(case-frame)が用意されているので、これを利用して'の'の処理を行う。

(4)時間，場所を示す名詞がきた場合

時間，場所の名詞がM1として使用された場合には、その時間，場所における名詞M2と解釈出来ると思われる。

### (B)「(M+の)K+M2」型の処理

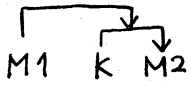
「K+M」型の処理は比較的簡単である。即ち、前述の辞書の中で、KがMのentry中の(5)'直接かかれる形容詞，名詞'の中に入っているか，又はMの属性，部分にあたる名詞の中でKによって直接形容される名詞があるかどうかをcheckする。例えば、

(a) 濃い溶液，(b) 赤い色

例(b)ではKがMを直接形容出来るから処理結果は(色(SPEC 赤い))となる。例(a)においては、溶液の属性である濃度が、Kによって形容されるから処理結果は(溶液(濃度 こい))となる。

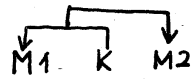
「M1+の+K+M2」型の係受関係はM1とM2，及びKとM1，KとM2の関係によって、大別して次の2種類がある。

(I) M2∈ATR[M1] 又は M2∈PW[M1] で，
かつ K∈MOD[M2] である場合
係受関係は下図のようになる。




(例) 水の高い沸点，鋼板の赤い部分

(II) M1∈ATR[M2] 又は M1∈PW[M2] で，
かつ K∈MOD[M1] である場合
係受関係は下図のようになる。




(例) 濃度の濃い溶液，沸点の高い塩化ナトリウム
これらの処理結果は、それぞれ(溶液(濃度 こい))，(塩化ナトリウム(沸点 高い))となる。

もちろん，「M1+の+K+M2」の並びであっても、M1とM2，KとM2が独立にかかって、上のように3項組(M1,K,M2)の関係を調べなくとも処理が出来る場合がある。

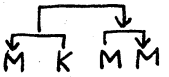
例えば、「固体の白い物質」の場合には、'固体∈ATRV[物質]' '白い∈ATRV[物質]' となって '固体' '白い' がそれぞれ独立に物質に係ることになる。この様な並びについては次の(C)の全体の処理の流れによって処理可能になる。

(C)全体の処理のFlowについて

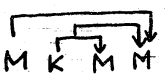
(A)，(B)において局部的な処理の進行について述べたが、次に全体として処理がどの様に流れるかを考える。実際の多くの名詞句においては、例えば、(1)「こい濃度の食塩の溶液」(KM1のM2のM3) (2)「濃度のこい食塩の溶液」(M1のKM2のM3) (3)「食塩の濃度のこい溶液」(M1のM2のKM3) (4)「食塩のこい濃度の溶液」(M1のKM2のM3) の様に、多数個のMとKからなり、その係受のパターンもその中の名詞によって異る。

(例)

(a) 濃度のこい食塩の溶液




(b) 食塩のこい濃度の溶液




この様な多数個の名詞や形容詞を含む場合の処理は、基本的には(A)(B)で述べた処理の繰返しであるが、PLATONのpush-down 枝能を用いて、柔軟に係受関係が調べられる。例で示す。

<例1>「濃度(M1)のこい(K)食塩(M2)の溶液(M3)」

(M K M)のpatternになっているが，M1とM2には(B)で述べたような関係が成立しない。したがって(M1 K)の部分は処理を保留して、(M2のM3)「食塩の溶液」の部分をpush-downして、処理する。この部分の処理結果は(溶液(溶質 食塩))となってpop-upしてくる。したがって全体の句の様子は

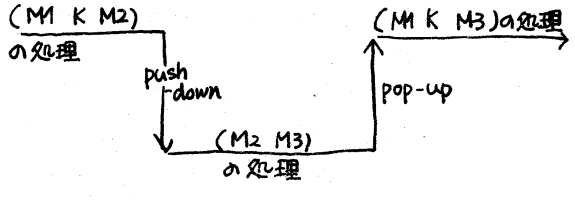
「濃度(M1)のこい(K)(溶液(M3)(溶質 食塩(M2)))」

となる。これに対して(M K M)のpatternをあてはめて、(B)で述べた関係が成立するかどうかをcheckする。すると'濃度∈ATR[溶液]' 'こい∈MOD[濃度]' となって条件が成立する。したがって全体の処理が完了して、処理結果は、

(溶液 (溶質 食塩) (濃度 こい))

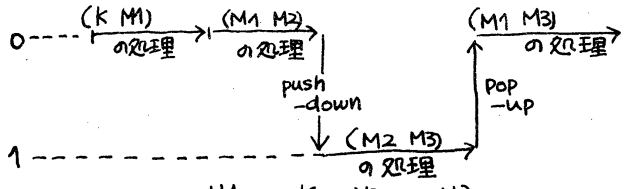
となる。この様に、一度(A)(B)で述べられた処理をおこない，その処理が失敗すると、push-downをおこして処理を一段進め，その処理結果を埋込んだ構造に対して、もう一度(A)(B)の処理をおこなうことを処理の基本的なflowとしている。なお、push-downの際のstateは全体の処理の出発点となるstateと同じであり、したがってpush-downされた処理も最高レベルでの処理も全く同じ過程である。

この例の場合のflowを図示すると、下の様になる。



<例2>「濃い(K) 濃度(M1) の 食塩(M2) の 溶液(M3)」

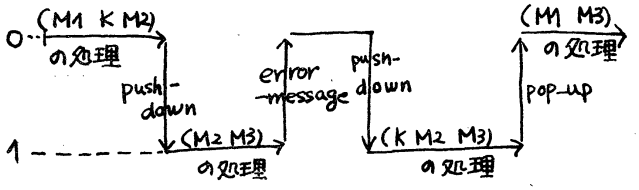
<例1>の場合と同じように図示すると下図のようになる。




<例3>「食塩(M1) の 濃い(K) 濃度(M2) の 溶液(M3)」

この場合には、push-downした後の処理が失敗し、その失敗原因をerror-messageとして返すことにより、処理の流れがcontrolされる。すなわち'濃度の溶液'という句は解析不可能で失敗する。これは'濃度∈ATR[溶液]'であるが、日本語においては、属性語(この場合では'濃度')が、本体の語より前に単独で現われることがないからである。そして'こい濃度の溶液'ならば解析可能である。

この場合の処理の流れは下図の様になる。



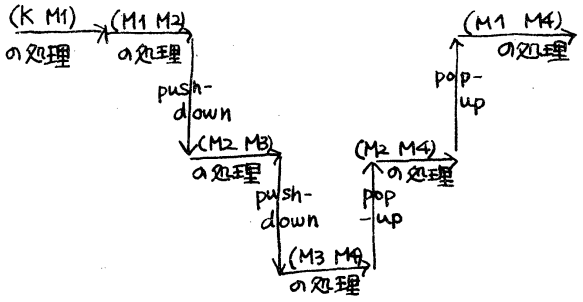

これら<例1><例2><例3>の処理結果はいずれも

(溶液 (溶質 食塩)(濃度 こい))

となり、同じである。また以上の例ではpush-downが1段であるが、プログラムとしては、push-downしたlevel 1の処理も、level 0の処理と全く同じ処理をするので、level 1の処理途中で再びpush-downが生じてlevel 2の処理へ移ることも当然可能である。この例としては、

<例4>「濃い(K) 濃度(M1) の 食塩(M2) の 100cc(M3) の 溶液(M4)」

この例の処理のflowは下図のようになる。




処理結果は、

(溶液 (溶質 食塩)(濃度 こい)(体積 100cc))

となる。

自然言語処理のためのプログラミング言語を作ることの意味は次のようなところにある。(1)人間が頭で考えた処理方法がなるべく素直に表現できること、(2)試行錯誤的に行なうために書き替えが容易であること、(3)他人が見てその内容がわかりやすいこと、(4)アルゴリズムの記述能力がすぐれていること

我々の作ったPLATONはこのような条件を十分に満たしているとは言えないまでも、かなり便利なシステムになっているものと考えている。現在これを用いて日本語分析のプログラムを書いているが、システムとして特に大きな問題はない。このシステムは京都大学大型計算機センターのFACOM 会話型LISPの上で動いているが速度は会話型として許される範囲内にある。

文献

長尾真：自然言語処理のためのソフトウェア
昭和48年電気四学会論文集 No.348
W. Woods: Transition Network Grammar for Natural Language Analysis, CACM Vol.13, No.10, 1970