自然言語処理 68－2
（1988. 9. 16）

ＬＦＧと意味解析の融合に向けて

新田善久
東京大学　工学部

[概要]

本稿では、ＬＦＧを用いた構文解析に知識からの制約を加えることにより、文脈に適した解析を行なう方式を提案する。まず、自然言語処理において意味的制約が重要な役割を果たすことを述べる。次に、ＬＦＧの語彙項目に意味的制約を記述できるように拡張することで、単一化文法の枠組みの中で知識からの制約を扱うことができることを示す。この方式を用いれば、語に関する意味的制約は知識ベースを操作するので、文脈が形成され、その文脈に適した解析を行うことになる。　そして、本方式に基づいたシステムをProlog上にインプリメントすることで、その有効性を実証する。

Toward the Unification of LFG and Knowledge Processing

Yoshihisa Nitta

The University of Tokyo
Wada Lab., Department of Information Engineering, Faculty of Engineering, University of Tokyo,
7-3-1, Hongo, Bunkyo-ku, Tokyo, 113, Japan

[Abstruct]

This paper presents a new method of natural language processing in context using extended Lexical Functional Grammar in which semantic constraints can be described. In this paper, the importance of semantic constraints in natural language processing is overviewed. A new method to deal with semantic constraints in Lexical Entries of Unification Grammar is also described. In this method, semantic constraints on words refer to knowledge base and change its data, so context can be formed in knowledge base. Finally, the implementation of this method on Prolog is shown.

1　はじめに

自然言語を処理するプロセスはいくつかのレベルに分割して考えることができる。つまり、大きく分けて単文に対する処理と文脈に対する処理があり、これらは各々さらに下位の処理に分割できる。単文に対しては、構文解析して品詞を決定し、単語の意味を決定し、その意味を合成して単文に対する意味を決定する、などのプロセスが必要であり、文脈に対しては、単文の意味から文脈を形成し、その文脈に照らして単文の意味を決定し、さらに常識に照らして言外の意味を理解する、などのプロセスが必要である［2］。

しかし、人間が自然言語を「理解」する際に行なう処理がこのようにはっきりと独立している筈はなく、これらのプロセスが平行して動作し互いに相互作用を行っていると考えるべきである。従来はいくつものレベルに分割して扱うことであまりにも個々のレベルのみにとらわれてしまい、他のレベルとの相互作用について無頓着になる傾向にあった。あるレベルと他のレベルの解析が同時に進むというモデルが、人間の言語理解モデルとして適当であるにもかかわらず、なかなかそれを表現することができない状況にあった。

本研究は、単一化文法の枠組みの中に意味解析の情報も同時に記述することで自然言語の構文解析と意味解析を融合し、知識を解析結果に反映させるための構成法を提案する。さらに、システムを試作することでその有効性を実証する。

構文解析方式としてはＬＦＧ(Lexical Functional Grammar)を、知識表現にはフレームを採用する。ＬＦＧが語句の結び付きに関するSyntacticな情報を語彙レベルで記述できる点に着目し、意味情報も記述できるように拡張を行う。すなわち、辞書中の語彙項目にフレームを操作する情報も同時に記述できるようにし、機能構造間の単一化における制約として扱う。それによって、語の多義性からくる自然言語の曖昧性を減らし、文脈に適した解析を行うことを実現する。この方法が優れている点としては、このフレームに既に解析した単文の意味を登録することで、語が評価される環境（つまり文脈）を変化させることができ、語の解釈と文脈の相互作用が自然に実現できることがあげられる［1］。そして、本方式に基づいた日本語処理システムを試作することで有効性を実証する。

2　ＬＦＧの拡張と意味処理の概要

2．1　自然言語の曖昧性と知識の必要性

自然言語においては、多義性を持った言葉が多く、意味が曖昧になる場合が多い。英語の動詞の意味を決定する場合を考えてみると、"love"とか"walk"といった比較的はっきりした意味を持つ動詞の場合は問題は少ないが、"take"や"get"などの曖昧性の高い動詞の場合はすぐに困難に直面してしまう。例えば、

①Taro took a girl to the station.
「太郎は少女を駅まで連れて行った」
②Taro took a bus to the station.
「太郎はバスに乗って駅まで行った」
③Taro took a girl by the hand.
「太郎は少女の腕をつかまえた」

という英文①～③の中における"take"という語の意味は各々「（人を）連れて行く」、「（乗り物に）乗って行く」、「（人を）捕まえる」になる。①と②では句構造は全く同じであり、異なるのは直接目的語となる名詞が「人」であるか「乗り物」であるかの違いだけである。また、①と③では前置詞句が異なるだけである。この違いを判断するには「girlは人である」、「busは乗り物である」、「handは体の一部である」などといった世界に関する知識（つまり常識）が必要であり、かつ、この知識を処理の過程で利用しなくてはならない。

英語を母国語とする人はこれらの意味が複雑に絡み合ったものとして"take"という語を捉えているのだろう。しかし、「文の意味を明確に理解」した状態では"take"が持つぼんやりとしたイメージをそのまま用いている筈はなく、文の解析が終った段階ではその中のどれかの意味が活性化されている筈である。この活性化は文の他の要素との相互作用によって起こるというのが重要な点である。

もちろん、日本語処理においても、世界に関する知識は重要である。例えば

④太郎は弟を進学させた
「太郎が弟を（強制的に）進学させた」
⑤太郎は弟に進学させた
「弟が自発的に進学し、太郎はそれを認めた」
⑥父の成功が花子を喜ばせた
「花子が父の成功を喜んだ」
⑦花子は子供を流感にかからせた
「花子が直接影響を及ぼしたわけではないが、花子の子供が流感にかかってしまった」

という例文④～⑦を考えると使役動詞「させる」のにニュアンスが文によって異なる。④では、太郎が弟の進学に対して影響を与えたことは分かるが、弟がそれを希望していたかどうかは分からない。しかし、⑤では、弟の進学は自発的であったことが分かる。⑥では無生物は意志を持たないので花子が動作者であるし、⑦では、子供が流感にかかることに関して花子が積極的に働きかけたわけではない。

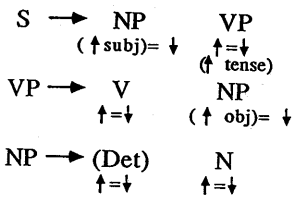
この例から、「させる」という語の意味は文中の他の語との関係によって非常に影響を受けていることがわかる。「させる」の意味は、

「にー使役文では、補文の主語は生物でなくてはならなず、補文は自発的行為である」
「主語が無生物のときは、目的語が動作主である」
「補文の内容が自然に起こるものであるときは、主語は経験者であり、補文の内容に直接働きかけはしない」

といった点を考慮しないと特定することはできない。

2．2　LFG

LFGにおいては、文の表現としては、構成素構造と機能構造の2つのレベルを設定する。構成素構造は、文の表層の句構造を表し、機能構造は、文法機能の種類（属性）とその値（属性値）を表す。構成素構造を作るには、句構造規則に単一化に関する情報を付加した規則である構成素構造規則（図1）を用いて入力文に対応する構文木を作り、これに辞書から語彙項目（図2）を挿入する。機能構造は、句構造のnodeに関する機能構造を単一化によって求めていけば、最終的に文全体に対応する機能構造が得られる（図3）。



［図1　簡単な英語に対する構成素構造規則］

taro	N	(↑ pred)='taro'
		(↑ num)=singular
loves	V	(↑ pred)='love<(↑ subj)(↑ obj)>'
		(↑ tense)=present
		(↑ subj num)=singular
		(↑ subj person)=third
the	Det	(↑ spec)=the
girl	N	(↑ pred)='girl'
		(↑ num)=singular

［図2　簡単な英語に対する語彙項目］

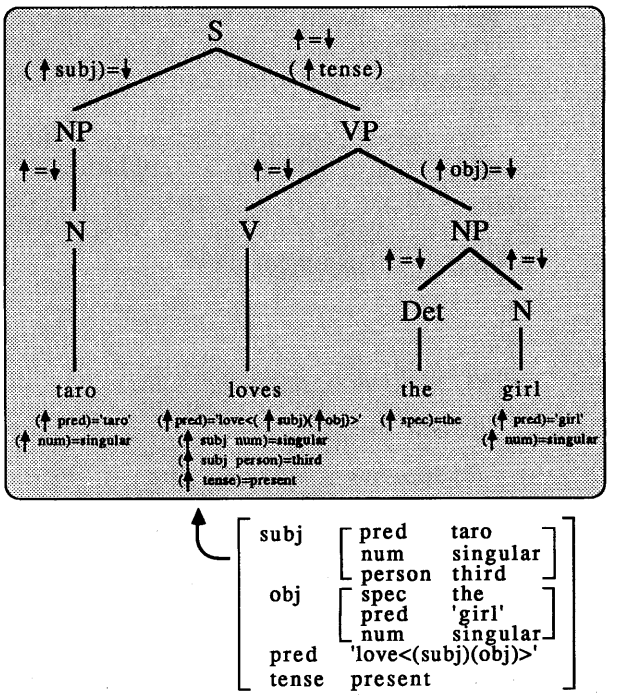
文の主動詞によって規定される意味述語は、その引数に格を割り当てている。格とは、agent（動作主）、theme（対象）、source（起点）、goal（終点）などのことである。例えば"love"という動詞は

love（誰が、誰を）

と表現できるので引数は2つあり、第1引数は動作主に、第2引数は対象に対応することになる。LFGでは引数に、文の表層のどの文法機能が現われるかを表したものを述語項構造と呼び、"love(subj, obj)"のように書く。これを用いて

"Taro loves the girl"
subj　　obj

という文から「love(taro, girl)」という表現を導き出せる。



［図3　構成素構造と機能構造］

2．3　LFGと意味処理

LFGでは、語彙レベルの持っているさまざまな情報を、単一化という操作で結び付けてしまう。この単一化への制約として、機能構造について

「属性値が存在する」
「属性値が存在しない」

といった条件を記述できた。条件を満たさないものは、単一化に失敗して、正しいものだけが残ることになる。

しかし、2．1で述べたように、ある文の中に現われる語

句は、他の語から受ける意味的な制約によって意味が異なり、これが自然言語の曖昧さをもたらす要因の１つになっていた。この「他の語から受ける意味的制約」が利用できなくては正しい解析はできない。そこで、この意味的な制約条件も記述できるように拡張を行う。つまり、

「指定する属性値がある概念とどのような関係にあるか」が語彙項目の中に記述できるようにする。

そして、１つの語について、条件と述語項構造が異なる語彙項目を複数用意しておけば、条件に合わないものは単一化の過程で失敗し、正しい述語項構造を持つものだけが残ることになる。

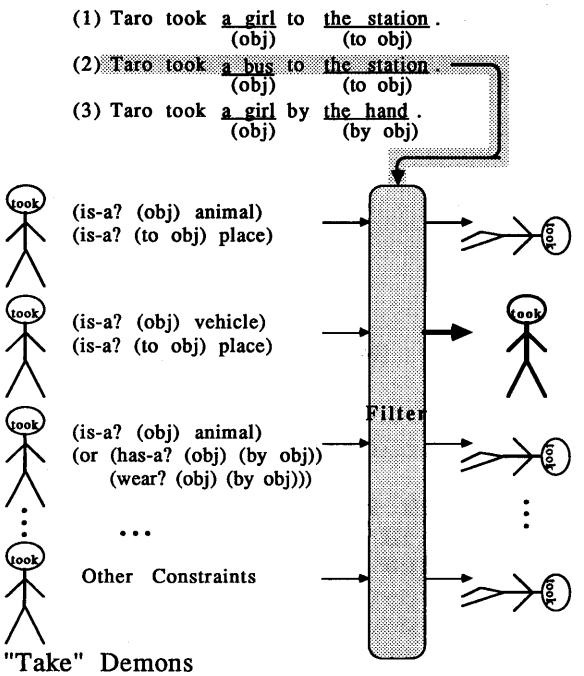
英単語"take"の述語項構造とそれに対する条件は、

「第２引数が人間の下位概念で、第３引数が場所の下位概念の場合、"take"は『連れて行く』という意味を持つ」

「第２引数が乗り物の下位概念で、第３引数が場所の下位概念の場合、"take"は『乗って行く』という意味を持つ」

「第２引数と第３引数の概念の間にhas-aという関係がある場合、"take"は『捕まえる』という意味を持つ」

なので、"Taro took a girl to the station"という英文を解析するときには『連れて行く（subj, obj, to obj)』という意味を持つものだけが生き残る（図４）。



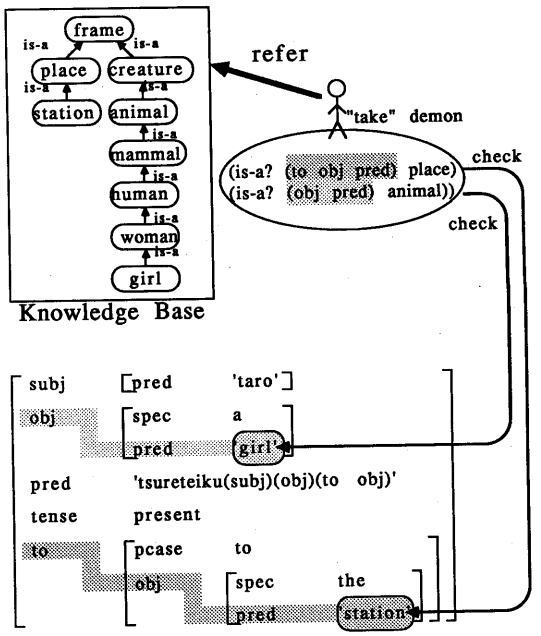
［図４　意味的制約によるフィルタ］

得られた機能構造は、機能構造の[obj, pred]と[to, obj, pred]という属性値と知識ベースの間で

「"girl"は人間の下位概念である」

「"station"は場所の下位概念である」

という常識が調べられている（図５）。



［図５　知識からの制約を用いた単一化］

３　ＵＧＡＯの構成

以上に述べた方式に基づき、日本語の入力文を解析するシステムＵＧＡＯを試作した。語彙項目や構成素構造規則をデバッグするために、分かりやすい記述ができるようにし、Ｐｒｏｌｏｇ節にコンパイルして使用する。

３．１　語彙項目

語彙項目は

単語　<==　カテゴリ，条件1，...，条件n

という形で表現する。ただし、条件は次の３種類がある。

属性名　@　属性値

「属性値を指定する」

属性名 <=> 属性名

「コントロール、つまり指示上の依存関係を指定する」

属性名 # 制約条件

「属性値が具体化されたときに実行される制約条件で、フレーム操作述語を書く」

これらの表記法に従えば、日本語に対応する語彙項目の一部は図6のようになる。

```
% 動詞「させる」終止形
saseru  <==  v, [pred] @ cause(subj,obj,xcomp),
             [tense] @ present,
             [vcase] @ shuushi_kei,
             [xcomp, vcase] @ mizen_kei,
             [obj] <=> [xcomp,subj],
             [subj,pred] # isa(animal).
saseru  <==  v, [pred] @ enable(subj,obj2,xcomp),
             [tense] @ present,
             [vcase] @ shuushi_kei,
             [xcomp, vcase] @ mizen_kei,
             [obj2] <=> [xcomp,subj],
             [subj,pred] # isa(animal),
             [obj2,pred] # isa(animal),
             [xcomp,pred] # isa(voluntary_act).
saseru  <==  v, [pred]@experient(subj,obj,xcomp),
             [tense] @ present,
             [xcomp,subj] <=> [obj],
             [xcomp,pred] # isa(voluntary_act).
% 動詞「読む」未然形
yoma    <==  v, [pred] @ read(subj,obj),
             [vcase] @ mizen_kei,
             [subj,pred] # isa(animal),
             [obj,pred] # isa(book).
% 形容詞「美しい」終止形
utukushii   <==  a, [pred] @ beautiful(subj).
% 名詞
taro    <==  n, [pred] @ taro.
hanako  <==  n, [pred] @ hanako.
manga   <==  n, [pred] @ commic,
             [pred] # isa(book).
% 後置詞
ga      <==  p, [subj,pcase] @ ga.
ga      <==  p, [obj,pcase] @ ga.
wo      <==  p, [obj,pcase] @ wo.
ni      <==  p, [obj2,pcase] @ ni.
% 文末詞
yo      <==  final, [final] @ yo.
```

[図6 日本語に対する語彙項目の例]

3．2 構成素構造規則

構成素構造規則は

```
親カテゴリ  <--
    子カテゴリ1 ( [制約11, ..., 制約1i] ) ,
    子カテゴリ2 ( [制約21, ..., 制約2j] ) ,
        . . .
    子カテゴリn ( [制約n1, ..., 制約nk] ) .
```

という形で表現することにする。但し、制約は単一化に関する情報であり、Prologのgoalで表現されている。これらのgoal中に現われる"head"と"self"というキーワードは親nodeとそれぞれの子nodeの機能構造を表す変数に置換されてから、goalが実行される。ある構成素構造規則の中に現われる変数は、全ての子カテゴリで共通のものとして扱われる。

":="というオペレータは機能構造の単一化を行う2引数の述語である。引数としてリストが与えられている場合は、パスが与えられているものとして扱う。

すると、日本語の簡単な文法は、

```
s     <--   sc([head := self]),
            final([head := self]).
sc    <--   np([head := self]),
            vp([head := self]).
sc    <--   np([head := self]),
            sc([head := self]).
sc    <--   sc([[head, xcomp] := self]),
            vp([head := self]).
vp    <--   adv([head := self]),
            v([head := self]).
vp    <--   adv([head := self]),
            a([head := self]).
np    <--   n([[head, X] := self]),
            p([head := self, define(self,X)]).
n     <--   sc([[head, xadjunct] := self,
                gap(self, X),
                control([head], [self, X])]),
            n([head := self]).
```

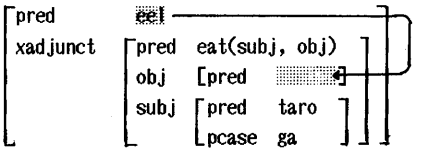
と表現できる。

名詞句が名詞と後置詞からなることを表す構成素構造規則では、後置詞が名詞句の文法機能を決めるので、後置詞の制約条件に文法機能を決める述語define(self, X)が書かれている。define(self, X)というgoalの呼び出しによって、変数Xを、後置詞の機能構造selfが規定する文法機能に具体化すると、名詞の制約条件中の変数Xも同じ値になり、結局、名詞句の機能構造のXという属性値と名詞の機能構造が単一化されることになる。

gap/2はpred属性値が規定する文法機能のうち欠けているものを捜す述語であり、これを用いて連体修飾文の中にコントロールされるべき文法機能を決めている。つまり、

「太郎が食べた　饅」

を解析すると、objが欠けているのでこの位置にコントロールされて、



となる。

３．３　ＵＧＡＯのＰｒｏｌｏｇ節へのコンパイル方式

まず、ＵＧＡＯの解析方法について述べ、これを実現するために、語彙項目と構成素構造規則をＰｒｏｌｏｇ節へとコンパイルする方式を説明する。

３．３．１　ＵＧＡＯの入力文処理アルゴリズム

解析はボトムアップに進む。つまり、日本語では一般に後に現われる語が文法機能を決定するので、解析が終った部分をstackに積んでおき、構成素構造規則の最右辺のカテゴリが現われた段階で、その構成素構造規則を適用するのである。

ＵＧＡＯのトップレベルの処理は、基本的には次のようになる。

（１）入力語があれば（３）へ

（２）　　無ければ、stackを調べ、文を表すカテゴリとその機能構造の組だけが積まれていれば成功。それ以外では、backtrack。

（３）語彙項目から、その語のカテゴリと機能構造がわかる。これを組にして、stackに積む。

（４）stackの先頭のカテゴリを最右辺にもつ構成素構造規則がなければ（１）へ

（５）あれば、（６）または（１）へ

（６）stackの先頭のカテゴリを最右辺にもつ構成素構造規則を適用し、stackの内容を書き換える。

（７）（６）または（１）へ

これは、Ｐｒｏｌｏｇ上で容易に実現できて、

```
jparse([], [s(F,_,_)], KB).
jparse([W | Rest], Stack, KB) :-
    push(W, Stack, Stack1, KB),
    pop(Stack1, Stack2),
    jparse(Rest, Stack2, KB).

push(W, Stack, [Goal | Stack], KB) :-
    jdic(W, Cat, F, KB),
    Goal =.. [Cat, F, _, _].

pop([Goal | Stack], NewStack) :-
    arg(2, Stack), arg(3, NewStack), call(Goal).
pop(Stack, Stack).
```

と書ける。ここで、jdic/4は辞書項目を調べる述語であり、辞書項目の中で知識にアクセスするために第４引数がわたされている。stackに積まれるカテゴリと機能構造の組は、構成素構造規則を適用するためにカテゴリを名前とする３引数のfunctorで表現する。stackに積むときに第１引数を機能構造に具体化しておく。構成素構造規則を適用するときに、第２引数として変更前のstackをわたし、第３引数に変更後のstackを得る。

３．３．２　語彙項目のＰｒｏｌｏｇへのコンパイル

語彙項目は

単語　　<==　　カテゴリ，条件1，...，条件n

という形で書いたが、これは、

```
jdic(単語，カテゴリ，機能構造，知識ベース）:-
        条件チェック1，...，条件チェックn.
```

という形のＰｒｏｌｏｇ節へとコンパイルする。

条件は、それぞれ次の形にコンパイルする。

属性名　@　属性値　　の場合

```
locate([F | 属性名], 属性値)
```

属性名1　<=>　属性名2　の場合

```
control([F | 属性名1], [F | 属性名2])
```

属性名　#　制約条件　　の場合

```
locate([F | 属性名], X), freeze(X, 制約条件)
```

但し、知識ベースと属性値を制約条件の引数として入れるため、制約条件の引数は２増える。

ただし、locate/2は機能構造を単一化する述語で、ここにリストが書かれていた場合には、そのパスをたどった先にある属性値と単一化する。control/2も単一化する述語である。

ここで、注意しなくてはいけないのは、freeze/2という述語である。これは、最近のPrologに組み込み述語で用意されている述語であり、

```
freeze(V, P)
```

が実行されると、述語Pは変数Vが具体化されるのを待ってから実行される。つまり、変数Vが具体化されるまでは、述語Pの実行は遅延される。単一化文法では、文のいろいろな部分が機能構造にアクセスするのでどの段階で制約条件をチェックしてよいのか判断が難しいが、freeze/2を用いれば、必要な概念が具体化されるのを待って具体化後直ちにチェックすることが、容易に実現できる。

isa/3は、知識ベースの中で概念の上下関係が存在するかどうかをチェックする述語である。

これらを用いると、

```
saseru  <==  v, [pred] @ cause(subj,obj,xcomp),
             [tense] @ present,
             [obj] <=> [xcomp,subj],
             [subj,pred] # isa(animal).
```

という語彙項目は、

```
jdic(saseru, v, A, B) :-
    %[pred] @ cause(sub,obj,xcomp)のコンパイル結果
        locate([A,pred], cause(subj,obj,xcomp),
    %[tense] @ presentのコンパイル結果
        locate([A,tense], present),
    %[obj] <=> [xcomp,subj]のコンパイル結果
        control([A,obj], [A,xcomp,subj]),
    %[subj,pred] # isa(animal)のコンパイル結果
        locate([A,subj,pred], C),
        freeze(C,isa(B,animal,C)).
```

とコンパイルされる。

３．３．３　構成素構造規則のＰｒｏｌｏｇ節へのコンパイル

構成素構造規則は

```
親  <--  子1, 子2, . . . 、子n.
```

という形で書いたが、これは、

```
子n(F0, [子n-1,...,子1 | Stack],[親 | Stack]) :-
        子nの単一化に関する制約,
                . . .
        子2の単一化に関する制約,
        子1の単一化に関する制約.
```

の形にコンパイルする。例えば、

```
np      <--     n([[head, X] := self]),
                p([head := self, define(self,X)]).
```

という構成素構造規則は、

```
p(A, [n(B,_,_) | Stack], [np(C,_,_) | Stack]) :-
        C := A,
        define(A, X),
        [C, X] := B.
```

とコンパイルされる。このＰｒｏｌｏｇ節のヘッド部は、「stack上に前置詞pがあり、その下に名詞nがあれば、代わりに名詞句npを積め」ということを表現している。その場合に満たすべき条件がボディ部に書いてあり、「前置詞・名詞・名詞句のそれぞれの機能構造を変数A, B, Cとしたときに、AとCを単一化し、Aから文法機能を特定し、Cのその文法機能を属性とする値とBを単一化せよ」ということを表している。

もちろん、途中まで間違った解析を行うこともあるが、その場合はbacktrackによって再計算を行う。

「太郎が弟を進学させる」という文を処理すると、stackの状態は図７のように変わっていく。stackの中のfunctor名は既に処理したカテゴリで、引数はそのカテゴリが表す機能構造である。

	語	top←　ｓｔａｃｋ　→end
処理↓	太郎	[n(F0,_,_)]
	が	[p(F1,_,_),n(F0,_,_)] F1とF2を単一化 [np(F2,_,_)]
	弟	[n(F3,_,_),np(F2,_,_)]
	を	[p(F4,_,_),n(F3,_,_),np(F2,_,_)] F4とF3を単一化 [np(F5,_,_),np(F2,_,_)]
	進学さ	[v(F6_,_),np(F5,_,_),np(F2,_,_)] [vp(F6,_,_),np(F5,_,_),np(F2_,_)] F6とF5を単一化 [sc(F7,_,_),np(F2,_,_)] F7とF2を単一化 [sc(F8,_,_)]
	せた	[v(F9,_,_),sc(F8,_,_)] [vp(F9,_,_),sc(F8,_,_)] F9とF8を単一化 [sc(F10,_,_)] [s(F10,_,_)]

［図７　解析中のｓｔａｃｋ］

３．３．４　フレームのＰｒｏｌｏｇによる表現

フレームをＰｒｏｌｏｇで表現するには、assert/1とretract/1を用いるのが簡単であるが、assertによって登録されるのは具体化された値だけであり、変数間の束縛やfreezeされた制約条件に関する情報は失われてしまう。また、backtrack時に元に戻さなくてはいけないことを考えると、Ordered Binary Treeを保持するＰｒｏｌｏｇの変数として表現することにした。述語名はＦＲＬと同じにし、第１引数にフレームを表す変数を書くことにした。

辞書を引くときjdic/4をコールするが、第４引数をフレームを表す変数に具体化する。３．３．１で見たように、この

変数は意味的制約条件の第１引数としてわたされる。

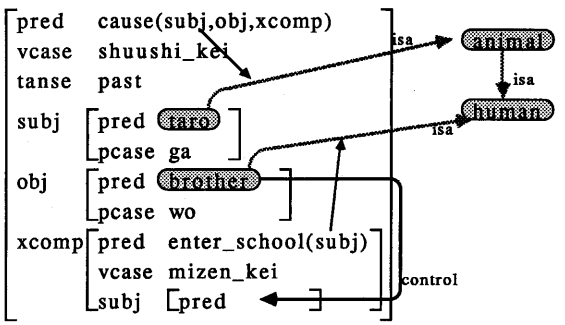
文を解析するにつれて、語に対応する概念が、フレームに登録されてゆく。

３．３．５　実行結果

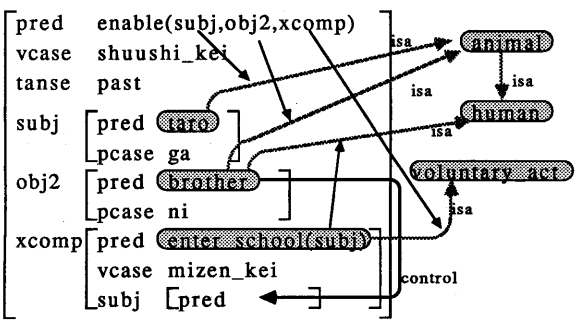
例文「太郎が弟を進学させた」と「太郎が弟に進学させた」の解析結果を図８に示す。[] で囲まれた中が機能構造である。楕円で囲まれたものは１つの概念を表し、その間の矢印（太い点線）が概念間の関係を示している。矢印（太い点線）への矢印（細い実線）は、概念間の関係が、語彙項目のどの部分への意味的制約から導かれたのかを示している。機能構造の中のpred属性値は、意味的制約によって他の概念との関係が知識ベースに登録される。。

例文の解析結果の機能構造の中のpred属性値が異なり、ニュアンスの違いが反映されている。

"taro ga otouto wo shingaku saseta"



"taro ga otouto ni shingaku saseta"



［図８　解析結果］

４．まとめ

単一化文法であるＬＦＧに、知識からの制約を記述できるように拡張することで、常識にそくした解析を行う方式を提案した。

本方式の利点としては、次の点が挙げられる。

（１）異なる「言語理解のレベル」間の相互作用を単一化という統一した枠組みで扱うことができる。

（２）語と語の間で互いにかけあう制約を用いて、文の意味表現を明確化することができる。

（３）ある語に関する制約は、その語が知識ベースの中で持つ概念にアクセスするので、語に関する情報が学習される。

５．　今後の課題

現在で用いている意味的制約は、

「花子は人間である」

「人間は腕を持ってる」

といった概念の上下関係や依存関係なので比較的軽い仕事であった。しかし、非単調推論のような複雑な問題を扱うようにすると、意味処理が非常に重くなる。また単文の解析結果を知識ベースに登録して文脈を変更するには整合性を保つ必要があるが、この整合性の判断は［６］，［７］によればフレーム問題に直面してしまい、解決するのが非常に困難になる。しかし、文脈処理をするためには避けては通れない問題であり、今後の研究を進める必要がある。

［参考文献］

［１］新田善久：Unification Grammarによる自然言語処理の研究、東京大学修士論文，1987

［２］石崎俊、井佐原均：文脈情報を抽出するための意味表現構造について、知識工学と人工知能36-3、1984

［３］Bresnan, J. (ed)：The Mental Representation of Grammatical Relations, MIT Press, 1982

［４］Iida, M. et. al.：Working Papers In Grammatical Theory And Discourse Structure, CSLI, 1987

［５］Sells, P.：Lectures on Contemporary Syntactic Theories, CSLI, 1985

［６］Hanks, S. and McDermott, D.：Nonmonotonic logic and temporal projection, Artificial Intelligence, 33, pp.379-412(1987)

［７］松原仁、山本和彦：フレーム問題に関する考察（２）、知識工学と人工知能58-6，1988