Fセッション

機械翻訳と知識記述

1. 知識表現の手法と言語処理用辞書の記述

田中穂積
東京工業大学工学部教授

計算可能な知識

私は長く、自然言語の処理を研究してきました。自然言語を処理する際、一番たいせつなものは知識をどのようにして計算機にのせるかです。いろいろな方法を、自分なりに考えてきたつもりですが、どうもいま一つ納得がいきませんでした。

ところが、たまたま第5世代コンピュータ開発の計画が始まったときに、Prologという言語を知りました。これまで頭のなかでいろいろ考えたり、プログラムを作っていたことのほとんどが、Prologというプログラミング言語の枠組で解決できるのではないか、という考え方にある時期とりつかれました。そこで、最近考えていることをこれから話します。機械翻訳には、直接関係はないかもしれませんが、そのベースになるものだと考えています。

私が考えてきたことは、「計算可能な知識とはどういうものだろうか」ということです。計算可能ということですから、コンピュータに入力して、ある種の働きをするものでなければなりません。辞書や百科事典にはいろいろな知識が書かれてありますが、それを計算可能にするには、特別の処理が必要になります。辞書に書かれた知識は主として普通の文章です。

しかし、それだけでは不十分で、その他に書かれた文章を解釈するインタープリタ（これを推論と呼ぶ）とを抱き合わせにして、初めて計算可能な知識になると考えています。ところが、普通の文章をインタープリートすることが自然言語処理ですが、それがまだ十分でないため、計算可能な知識は、もう少し人工的な形式をもった知識でなければなりません。それが、推論機構の裏づけにより計算可能な知識になるのだと考えます。

文法に関する知識と辞書に書かれる知識

従来の知識表現の形式をみると、そのあたりが分離していたのではないかと思います。例えば、言語学的な知識のなかに文法があります。これはある意味で形式です。その形式を誰が推論するかというと、人間が推論するわけです。ですから、ある一定の形式で書かれた文法、セオリーでもよいのですが、それはいろいろな人の頭のなかの推論メカニズムによって理解できる知識になっています。計算機でみると、パーザー(構文解析部)というものが推論に対応しています。パーザーと文法がいっしょになって、何か文法として働くコンピュテーショナルな文法になるのだろうと思います。

そこで、このパーザーに相当する部分を子細

に観察してみると、文法規則を扱うインタープリタの機能の大部分がPrologのインタープリタで代用できることがわかってきました。

しかし、意味処理などについてはどうかということが、私の研究の動機になりました。それがある程度、可能であることを話します。

日本でも知識に関する研究が、非常に活発に行われています。しかし、それらのほとんどは、自分なりの形式を考えて、自分なりの推論をするプログラムを作成するという方法でした。一方私は、意味処理のための推論をPrologの組み込み機能に任せられないか、ということを考えました。そうなると、プログラム作成労力が軽減し、「誰でも自然言語処理を扱えるような時代になる」ということにもなります。

自然言語に関する知識には、一つは言語学的な知識があると思います。そのなかの文法についての話は今日はあまりできません。どちらかというと、意味処理用辞書について触れたいと思います。言語学的な知識についての概略を知りたい方は、『Rereirao80』や、ICOT（新世代コンピュータ開発機構）の松本さんの研究がありますから、それらを参考にしていただければよいと思います。

言語処理における知識の役割

今日の話は意味処理用辞書が中心ですが、できれば常識のようなことにも触れます。

常識というとオーバーないい方かもしれません。できていることはほんのわずかです。しかし、いろいろな言語的な処理をしようと思うと、少なくとも非常に簡単な常識をもってないとうまくいきません。

それで自然言語処理において、知識がどのような役割を果たしているかを説明します。図1は機械翻訳の例です。一つはアナリシス、すなわち、言語を解析する部分があります。それに対して、解析した結果はある構造になります。この図はたまたま長尾先生のいわれる、ピボット（ある特有の言語に依存しない中間言語構造）のような表現になっていますが、とにかくある構造ができます。その構造を使って別の文をだすと、ジェネレーション（生成）と翻訳結果がえられます。

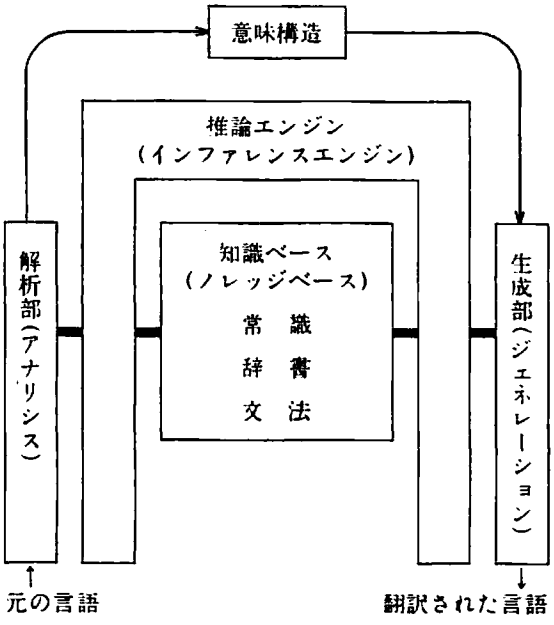


図1　機械翻訳による言語処理

したがって、アナリシスとジェネレーションの部分は、言語学的な仕事をするプログラムになります。これらの中心に知識ベースがあります。これは常識であるとか、辞書や文法という言語学的な知識を含む相当大きなものです。それを取り囲むように、推論エンジンがあるのではないかと考えています。

私の話は、解析と生成用のプログラムの大部分は作らなくてすむというものです。システムに組み込みになっている機能を使えばよいということです。

図1で示した例は、中間言語方式になっていますが、これは一つの理想の姿で、それなりによいところもあるし悪いところもあります。

そして、中間言語は果たして存在するのかどうか、私自身も理想的な中間言語がいつできるか見当もつきません。しかし、機械翻訳システムを考えるときに、こういうものを解析して、

どんどん解析の深さを上げる方向は、やはり中間言語を求める方向になるでしょう。

これとは別な方式として、知識ベースを三つに分けるトランスファー方式と呼ばれるものがあります。アナリシスに使われるものとジェネレーションに使われるものと、両者に依存するものの三つです。そのときには推論エンジンが間にあって、いろいろな訳語の選択をするということになると思います。いずれにしても、自然言語処理システムでは知識ベースの占める役割は非常に大きくなります。最近、その重要性が多くのところで認識され始めています。

全体の流れですが、Prologという言葉をそのまま使って話をしますが、少しわかりにくいことがあるかもしれません。いちおうホーン節形式という、ある種の述語論理の知識の形式をとります。そのなかの一つの枠組として、私がDCKR（Definite Clause Knowledge Representation）と呼ぶ形式が、どのようなものであるかを説明します。

意味ネットワークによる知識表現の問題点

推論と知識継承が、人工知能において非常に大きな問題になっています。まず、それとこのDCKRの関係、あるいは意味処理とDCKRの関係に触れることにします。

DCKRで書かれた知識は、かならずしも人間にとって書きやすい形式になっていません。もう少し高水準の知識の表現形式を作ってみたいわけです。これは一種のシンタックスシュガー（人間にわかりやすいように構文を変えること）のようなもので、SRL/Oというものを考えてみました。実際にはトランスレーターがあり、このSRL/Oの表現形式がDCKRの表現形式に変換され、Prologのインタープリタにより実行可能な知識になるわけです。

これまでの知識表現形式をながめてみると、代表的なものは意味ネットワークという形式であったように思われます。これは非常に直観的でわかりやすいために、多くのところで参考にされたり、使われたりしています。簡単にいうと、いくつか対象を表すノード（節点）があります。ノードとノードの間は関係を表すリンクでつながれているのです。

具体的に、図2の場合を考えてみます。この図をみた瞬間、人間は頭のなかに推論機構がありますから、次のように推論できます。

すなわち、clyde#1がいて、そのage（年齢）は6である。

clyde#1はelephant（象）である。

elephant#1はelephantである。

elephantのcolor（色）はgray（灰色）である。

elephantはmammal（哺乳類）である。

mammalのblood Temp（体温）はwarm（暖かい）である。

以上のことが、図をみただけでわかります。

それはまさに、私たちの頭のなかに、この形式を解釈するインタープリタをもっているからです。こういった知識を計算機に入力することはできますが、例えばclyde#1はblood Tempがwarmかどうかなどをみるためには、たぶんこのなかのネットワークをずっとたどっていかなくてはなりません。

したがって、たどるためのプログラムを作る

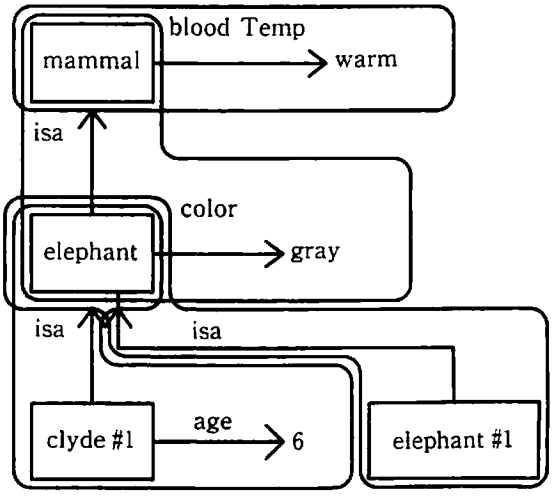


図2　意味ネットワークによる知識表現例

必要があります。それが、前述の「推論をするインタープリタが必要になる」ということです。

つまり、図のような形式の知識を計算機のなかに入れておいても、それを解釈するインタープリタを別途に作らなければなりません。そのインタープリタのつくりによって、意味ネットワークがいろいろな意味をもつことになります。それもまた、あまり都合のよいことではありません。

それに対して、DCKRと呼ばれる知識表現形式で書いておくと、その形式で書かれた知識はPrologのインタープリタが解釈してくれます。そのために、極端なことをいえば、書くだけでよいということになります。そこで次は、どのような形式で書くかが問題になります。

DCKRによる知識表現

次の例は、図2の知識をDCKRで書いたものです。

例　1

```
①sem1(clyde#1, age: 6 . _).
②sem1(clyde#1, P, S):-
    isa(elephant, P, [clyde#1 | S]).
③sem1(elephant#1, P, S):-
    isa(elephant, P, [elephant#1 | S]).
④sem (elephant, color: gray, _).
⑤sem (elephant, P, S):-
    isa(mammal, P, [elephant | S]).
⑥sem (mammal, blood Temp: warm, _).
⑦isa (Upper, P, S):-
    P=isa: Upper;
    sem(Upper, P, S).
```

例えば、clyde#1についての知識はsem1という述語を使い㋑のように「clyde#1のageが6である」と書けます。sem1という述語は、三つの引数があり、最初の引数はclyde#1で、2番めがageが6、というように性質を表すものです。最後の引数はいろいろな用途に使われますが、ここでは説明しません。ですから、意味ネットワークが与えられると、これらをDCKR形式で書くのは容易です。

また、clyde#1がelephantであるという知識は、sem1という述語をヘッドにして②のような形式で書きます。これは典型的なPrologの形式です。②をどのように読むかというと、：－はifと読み、次のようになります。

sem1 (clyde#, P, S) if isa (elephant, P,…)

普通、意味ネットワークでclyde#1の上位にelephantがいるという知識は、このように書きます。

これを解釈すると、先ほど第2引数はプロパティー（性質）のようなもので、第1引数はノードに相当するものでしたが、「elephantがPという性質をもつならば、clyde#1も同じPという性質をもつ」と、読みます。

ややこしいのは、第3引数をつけ加えなければならないことです。これはいわば機械語であるPrologそのもので書いていますから、それが必要になるわけです。

また、elephant#1がelephantであるという知識は、③のように書きます。ここでsem1という述語の第1引数には個体の名前がきます。そのように考えておきます。そしてもう一つ、semという述語を使ったのが、④です。④は「elephantのcolorはgrayである」と、読みます。

⑤は、mammalがPをもてば、elephantも同じPをもつということです。⑥は「mammalのbloodTempはwarmである」と、読みます。

ここで強調したいのは、意味ネットワークで書かれた知識は、Prologを知っていれば、比較的簡単に書くことができるということです。

isaの定義は、⑦のように書けます。これは非常に簡単で、Upper（上位）のコンセプト（概念）がPという性質をもつということです。isaが呼ばれたらそのボディでsemを呼びます。

コロンマイナス（：－）の意味は、「コロンマイナスの左が呼ばれて、中身（右側）が実行さ

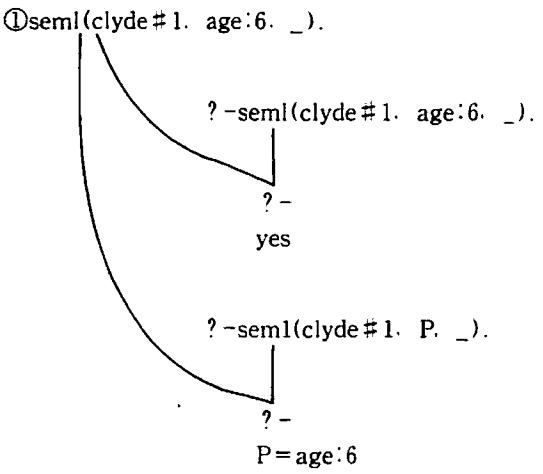


図3　Prologプログラムの実行
文に沿って変化を表すとよりよくわかる

れる」と、読みます。

：－の左辺をヘッドと呼びます。：－の右辺がボディです。ヘッドとボディでは、働きが違います。先ほどのいい方でいうと、ヘッドは呼びだし口になっていて、ボディはその中身になっていると考えてよいかと思います。同様にisaという述語にはヘッドがあり、ボディが二つあります。ボディ中の；(セミコロン）の記号はor（または）です。「；の前がうまくいかなければ、；の後を実行しなさい」ということです。

Prologプログラムの基本的な動き

ここで、Prologプログラムの実行を説明します。次のステートメントを入力します。

```
?-seml (clyde#1, age: 6, _).
```

を入力すると、Prologのプログラムが動きだします。ここで、上のステートメントと、先ほどみせたPrologのステートメントの集合がありますが、これらをPrologプログラムと呼ぶことにします。そうすると、図3のパターンヘッドがあうものをさがします。今の場合パターンseml (clyde#1, age: 6, _) を与えましたから、図3の①がすぐみつかります。

そこで、Prologは「このパターンにあうものがヘッドのところにあった」ということで消します。消すと、?-だけが残ります。

これはPrologプログラムの実行が成功したということです。以上のように「このパターンとこのパターンは同じだから消す」ということで、Prologプログラムの実行が進みます。

以上のことからわかるようにPrologは、パターンマッチ駆動（パターンマッチによって起動される）の計算をすることになります。大文字で始まる引数があると、それは変数を表します。次のパターンを与えてみます。

```
?-seml (clyde#1, P, _).
```

すると、このパターンとマッチするために、変数にいろいろなものを代入します。代入した結果、それとマッチするヘッドの部分が、プログラム中にあれば、やはり消します。今の場合、プログラム中の

```
seml (clyde#1, age: 6, _).
```

とパターン

```
seml (clyde#1, P, _)
```

を比較すると、第1引数は同じです。第2引数をみると、変数Pにage: 6を代入するとあいますから消すわけです。消した結果?-が残り成功です。そのときに代入したものはなんであるかを検索すると、それが答えになります。これがPrologプログラムの基本的な動きです。

知識継承とPrologのユニフィケーション機構

もう少し複雑なものをみてみます（図4）。

```
?-seml (clyde#1, color: gray, _).
```

これは、clyde#1のcolorはgrayかどうかを聞いているわけです。このパターンとあうヘッドをもつプログラムをさがしにいくと、最初は、

```
seml (clyde#1, age: 6, _)
```

ですが、パターンがあいません。そこで、

```
seml (clyde#1, P, S):-
    isa (elephant, P, [clyde#1 | S]).
```

をみると、これはあいます。どうあうかという

と、こんどはヘッドにPという変数があり、このPがcolor: grayであればよいわけです。パターンがあいますから消します。消すと、

isa (elephant, color: gray, ･･･).

が残り、これが新たなゴール（パターン）になります。ゴールとして新たな実行ステートメントになるということです。

今度は、isaという述語のヘッドとパターンがあいます。このプログラムがisa (elephant, color: gray,･･･)というパターンとなぜあうかというと、⑦のヘッドのなかのUpperは大文字で始まっているので変数で、Upperはelephantであればよく、また、Pはcolor: grayであればよいわけです。パターンがあうと、残りはボディを実行します。

このとき注意してほしいのはPにcolor: grayを代入していますから、ボディのなかのPは、color: grayに直しておきます。その部分をもう1回書くと、

?-color: gray=isa: elephant;

sem (elephant, color: gray,･･･).

になります。

color: gray=isa: elephant

は成立しませんから；の次の

sem (elephant, color: gray,･･･).

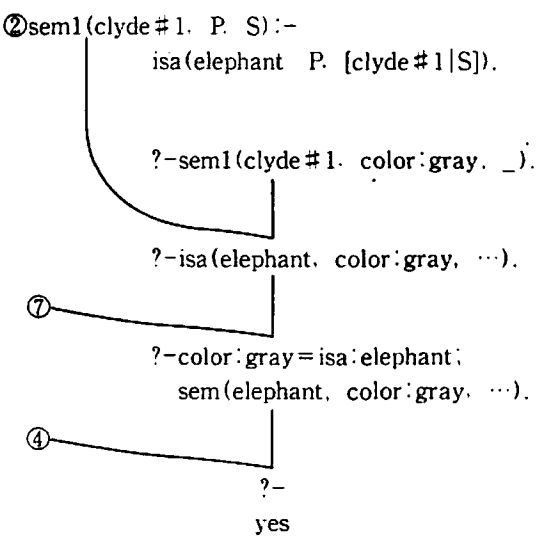


図4　Prologのユニフィケーション

を実行します。それにより④のヘッドとあい、めでたく実行が成功し「yes」という答えがえられます。

この動きをみていくと、Prologのプログラムを実行するだけでclyde#1のcolorがgrayかということに対して、yesという答えをえていることになります。このことはとりもなおさず、clyde#1の上位にあるelephantの性質を、clyde#1という個体からアクセスしたことになります。

これが、知識の継承というものです。「親のもっている性質を下からアクセスする」、別のいい方をすると「親のもっている性質が上から下に遺伝しておりてくる」ということです。このように、知識の継承に関しては、Prologのユニフィケーション（単一化）の機能が（今の例ではパターンマッチとして行っていましたが）、パターンマッチの機構そのものでも代用できてしまうことがわかります。

別の例を一つ紹介します。

?-sem1 (clyde#1, P, _).を実行すると、

P=age: 6;

P=isa: elephant;

P=color: gray;

P=isa: mammal;

P=blood Temp: warm;

とclyde#1のすべての知識がえられます。

普通ですと、ネットワークをたどるプログラムを作らなければなりませんが、その必要がありません。さらにおもしろいことに、

?-sem1 (X, Y, _).

を実行すれば、個体Xの性質Yを対にして、それぞれのもっているすべての知識を出力し始めます。またPrologの特徴を生かして、

sem1 (X, isa: mammal, _).

を実行すれば、

X=clyde#1;

X=elephant#1;

のように、mammalの下位に位置する個体を知

ることができます。一方、

?-sem(X, isa: mammal, _).

を実行すれば、

X=elephant

のように、mammalの下位に位置するものを答えてくれます。

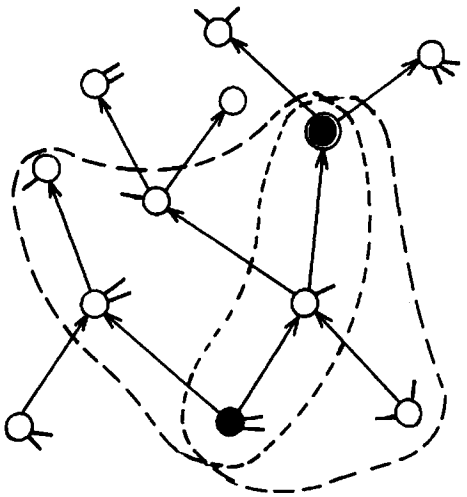
つまり、ネットワークを逆向きにたどることを、Prologが自動的に行います。ネットワークを上下自在にたどることが、プログラムを作らなくてできます。Prologがもっている基本計算機能で、それができてしまうということです。

継承範囲の限定

実はいろいろ知識を書いていると、知識継承に関する親のもっている性質を下からアクセスするとき、いろいろな問題があります。一つは継承範囲の限定をしたいことがときどきあります。あまり上の性質は欲しくない、上位のかなり上にある性質は欲しくないことがあります。詳しく述べられませんが、並立の格助詞「と」を含む名詞句の意味解析で、並立する名詞をきめるとき、意味的な近さを計算する必要があります。このときにも知識継承のコントロールが必要になります（後述）。

次に、継承経路にループがあるときの検出があります。それから継承知識の首尾一貫性も問題です。いろいろな親をもっている場合、各親から知識がおりてきますが、その間にコンフリクト（競合、矛盾）が起きてはなりません。これについては、まだ研究中です。少し見通しがえられてきましたが、もう少し先になると何か結果がでてくると期待しています。

まず、継承範囲の限定には、二つあると思います。図5にあるノード間の関係を、isaリンクと称するものと考えてください。そのようなネットワークが与えられたときに、一つは、あるノードから2段上までの知識（2段上のノード）にアクセスしたい、つまり、その範囲の知識がほしいとするときです。もう一つは、あるノードの2段下までの知識（2段下のノード）を、上から下に向かってアクセスしたいときです。



上方／下方　⇨isa述語

「日本の動物と植物の分布」

「東洋の仏教と西洋のキリスト教」

図5　継承範囲の限定

これらは、sem1, sem, isa述語の第3引数を使って簡単にできます。isaという述語については先ほど話しましたが、階層を上がったり下がったりするときは、かならずisaを経由します。そこで、isaという述語に少し細工をします。例えば、カウンターなどを入れておき、カウントダウンして0になったら、それ以上は上にいかないようにするわけです。これがある意味で、階層のコントローラーになります。そのコントローラーの部分を少し書き換えるだけで、継承範囲を限定することが簡単にできます。

私のところでは、並立する助詞のようなもので、例えば「日本の動物と植物の分布」の場合、動物と植物が並立します。「東洋の仏教と西洋のキリスト教」というと、仏教とキリスト教が並立します。シンタックスではこのあたりはうまく処理できなくて、どうしても知識を使わないと、並立名詞の固定はうまくいきません。

そのとき、動物というノードと植物というノードがあって、それが共通のノードをもつかど

うか、何段めぐらいで共通のノードをもつかといったことが類似性の計算に役立つとすると、継承範囲を限定するメカニズムは、その解析に役に立ちます。私のところで今その問題を、一人の学生が行っています。継承範囲の限定がうまくいって、この仕事がやりやすくなりました。並立する名詞句をきめないと、たぶん機械翻訳の結果はめちゃくちゃなものになるでしょうから、機械翻訳の立場からも重要であるといえましょう。そういうことで、継承範囲の限定を一つの応用例として紹介しました。

継承経路のループの検出

次は、継承経路のループを検出する方法についてです。

継承経路のループでは、例えば図6のように、「音楽の父はバッハ」という文で考えてみます。isaリンクを使って「バッハは音楽の父」、「音楽の父はバッハ」ということにすると、図のようなループが起きます。このように1段上がったところでのループならよいのですが、もっと複雑なループが起きたときにどう検出するかが問題になります。

一度たどった道をまたたどった、無駄な道をたどったということで、そのような知識を継承する経路を覚えておくメカニズムがあれば、ループの検出にも役立ちますし、同じことを二度行わないですみます。それもisa述語の第3引数、isa述語を少し拡張するだけで簡単にできます。

Prologに馴染みのない方にはわかりにくかったと思います。ここまでの結論は、知識表現は意味ネットワークのような形式でもたぶんよいのですが、推論することを考えると、もっとよい知識表現の方法があるということです。

確かに、意味ネットワークは、人間にとってわかりやすい形式です。なぜかというと、人間はその形式を推論するメカニズムを頭にもっているからです。しかし、それを計算機に入力するためには、計算機が人間の頭で行っているような、ネットワークをたどっていくプログラムを開発しなくてはなりません。それを、先ほどのような形式で知識を書きます。書くこと自体は非常に簡単です。書ければ、あとはそれを動かすだけで、いろいろな推論をすることが、Prologのインタープリタの力を借りてできます。

意味処理の基本的な考え方

次に、意味処理の基本的な考え方について、長尾先生が格フレームの話として触れましたので、その説明をします。これまで多くのフレーム処理のシステムでは、次のような意味処理を行っていました。

```
[open, frame,
[subj,[[human,=agent],
      [[$or, event, thingOpen],=object],
      [instrument,=instrument],
      [wind,=reason]]],
[obj, [$or, event, thingOpen],=object],
[with, instrument,=instrument],
[self, act]]
```

fig. 1 example of dictionary 'open' in SRL

これがどうして意味処理だといわれると困り

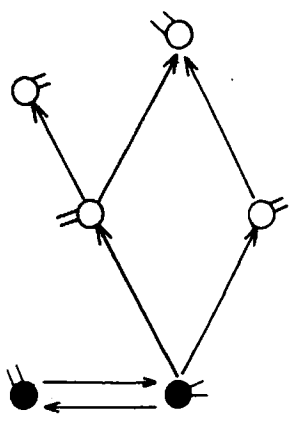


図6　継承経路のループの検出

ますが、「この程度の意味処理しかまだやっていない」といういい方が正確かもしれません。例えば「open」という辞書の記述に、いくつかサブパーツを用意しておきます。これはスロットと呼ばれています。このスロットには先頭に名前がついています。例えば、「I open the door.」といったら、openのフレーム中のスロットのどれかにIならIがフィラーとしてはまります。基本的な考え方はこうです。

ですから、意味処理はまず動詞から格フレームの知識（fig.1）をとりだしておいて、主語や目的語、前置詞句などが、どのスロットにはまりこむかを解析します。例えば「I open the door.」の場合だと、Iは主語です。これはシンタックス的なアナリシスによりわかります。そこでsubjのスロットを選んでみます。

すると、Iはたまたま人間ですから、人間であるかどうかを調べて、人間であればそれはagentという格にしましょう、ということをするわけです。そして、深層格と称されるagent格を抽出します。

したがって、意味処理の一番の基本は、まずスロットを選択することです。選択したら、フィラー（はめるもの）といいますが、例えば主語、目的語あるいは前置詞句といったものが、どのスロットにはまるか調べます。スロットの名前をシンタックス的な機能名にしておくと、それで選んで、さらにIならIというものの意味がどういうものであるか、人間であるのかどうなのかを調べます。

このとき、実は知識が必要になります。「Iは人間である」「花子は人間である」などの知識が必要になってきます。それは、isaリンクをたどることに相当します。普通は、これを、フィラーに対するセマンティックな制約といっています。

さらに、フィラーの意味を調べて、その意味チェックをして、合格したものをはめ込むといった手順で意味処理をします。

今の処理手順をまとめてみると、次のようになります。一つは選択したスロットの統語的ならびに意味的制約条件（今の場合は「統語的な制約条件」がスロットの名前になっていました）と、フィラーがもっている統語的、意味的な性質をつきあわせてみます。このときに、常識が必要になります。もしこうした条件をフィラーが満たせば、意味を抽出します。

もし、選択したスロットをフィラーが満たすことができなければ、次のスロットを選びます。それで同じことを繰り返します。ところが選択すべきスロットがないと、今度は上位概念のもつスロットにフィラーがはまるかもしれません。これを知識の継承といいましたが、上位がいくつかのスロットをもっているので、そこを調べるようにします。さんざん調べて、どれもうまくはまらなかったら、意味処理は失敗します。これが基本的な考えです。

DCKRによる辞書項目の記述

この考え方をよくよくながめてみると、Prologの組み込みの機構をそのまま実行できることがわかります。意味的制約条件は、ただ「人間だ」というラベルを貼りつけるだけではまずくて、人間かまたはなんとかであるとか、もっと複雑な意味的制約条件をスロットに書きたいこともあります。その場合、普通は論理式「and, or, not」の組み合わせで書くのがよいわけです。DCKRでorやandの記号がでてきましたが、このあたりの意味的制約条件を論理式で書くことは簡単です。

また、意味的制約条件の検査結果がうまくいかなかったら次のスロットを選ぶという場合ですが、Prologではパターンマッチしたとき、うまくいかなければ次のプログラムをとりだすことを、自動的にプログラムが行ってくれました。スロットを次々にとりだすことは、実はPrologのバックトラック機構といいますが、この機構を利用することができます。

さらに、上位のところに共通な性質が書いてあって、共通のスロットが書いてあるものにアクセスすることは、isaのメカニズムで簡単に実現できます。以上のことから、「open」の場合、次のように知識を書けばよいのです。

例　2

```
⑧sem(open, subj: Filler^In^Out, _):-
        sem1 (Filler, isa: human, _),
        addProp (agent: Filler^In^Out);
        (sem1(Filler, isa:event Open, _);
              sem1(Filler,isa:thing Open, _)),
        addProp ( object: Filler^In^Out);
        sem1 (Filler, isa: instrument, _),
        addProp(instrument:Filler^In^Out);
        sem1 (Filler, isa: wind, _),
        addProp(reason: Filler^In^Out).
⑨sem(open, obj: Filler^In^Out, _):-
        (sem1(Filler, isa:event Open, _);
            sem1
              (Filler,isa:thing Open, _)),
        addProp(object: Filler^In^Out).
⑩sem(open, with: Filler^In^Out, _):-
        sem1 (Filler, isa: instrument, _),
        addProp
              (instrument:Filler^In^Out).
⑪sem(open, P, S):-
        T=[open | S],
        isa (action, P, T);
        isa (event, P, T).
```

ヘッドの第2引数の先頭部分にスロットの名前が書いてあります。各スロットは、一つのホーン節で表します。そして、フィラーと称されるものの意味的制約条件のチェックは、：－の右にあるボディ部分にいろいろ書くことができます。

例えば⑧を訳すと次のようになります。Fillerがsubjという文法機能をもっているときは、以下を行います。Fillerが人間なら（カンマ(,)はandを表します)、Fillerをagentにしなさい。それがうまくいかなかったら、(orの記号がありますから) Fillerがevent Openなのか、もしくはFillerがthing Openなのかを調べなさい。それを満たせば、Fillerをobjectにしなさい。

このようにして、ボディの部分に意味的なチェックをさせます。スロットの選択は、一つがうまくいかなかった場合、次々に別のものを選ぶことをPrologが行いますから、このような形式で知識を書くと、意味処理用のプログラムはほとんど書かなくてすみます。

高水準知識表現言語SRL/O

私は昔、通産省の電子技術総合研究所にいて、そこで自然言語処理の研究を行っていました。そこでは、別のプログラム言語でプログラムを書いていました。そのときに比べてプログラムの量が驚くほど減りました。2桁ぐらい意味処理のプログラム量が減りました。

問題は、知識を書くことで、知識を先ほどの形式で書くことは、普通の人には少したいへんです。それで、それよりももう少し高レベルの書き方がないものかと研究しました。先ほどの意味ネットワークの例文だと、次のように書くことができます。

```
clyde#1::
        [age: 6]
        [isa: elephant].
elephant#1::
        [isa: elephant].
elephant::
        [color: gray]
        [isa: mammal].
mammal::
        [bloodTemp: warm].
X    ::
        [worksIn: Y
            where Y isa: department.
                  Y manager: X].
```

このように、SRL/0の書き方を導入すると、少しみやすくなります。第3引数をみないですむし、書かなくてもすみます。もう少しわかりやすくしようと、現在これよりさらに高レベルの知識の書き方を研究しています。

具体的にはユーザーがこういった形式で書いて、それを先ほどのDCKR形式に、計算機が自動的に翻訳します。動かすときには、そちらを使って動かすことになろうかと思います。

意味処理用の辞書の記述例を、SRL/0で書いてみると、次のようになります。

```
open::
        [subj $ isa: human⇨agent]
        [obj $ isa: eventOpen;
                isa: thingOpen⇨object]
        (with $ isa: instrument
                when object! instance
                isa: thing Open⇨instrument)
        (with $ isa: animal⇨coagent)
        ::
        [subj $ isa: eventOpen; isa: thingOpen
        ⇨object]
        ::
        [subj $ isa: instrument⇨instrument;
                isa: wind⇨reason]
        [obj $ isa: thingOpen⇨object]
        ::
        ((at; in) $ isa: place⇨location).
```

これでわかるように、意味を抽出するときに、一つのスロットがフィラーだけをみていればよいわけではなくて、他にどういうスロットがうまっているかというところまで、つまり2項関係だけではなくて多項の共起関係をみたいということも、おうおうにしてあります。これはwhenを使った書き方です。わざわざむずかしい例を書きましたが、そういったことも書けるようになっています。

おわりに

自然言語処理システムの基本的なアーキテクチャの説明のところで、Prologを利用しますと、アナリシスをするプログラムのかなりの部分（従来行っていたかなりの部分）が、Prologに組み込みの推論エンジンで置き換えることができます。そうすると、あとは知識を書けばよいだけです。

この知識も、生のかたちでPrologのプログラムを書くのはたいへんですから、少し高レベルの記述形式SRL/0で書きます。それを変換して、Prologのコードをだして実行すると、アナリシスの部分の意味処理も、Prologの組み込み機能を利用して行うことができる、ということを示してきました。

従来、言語学者が、自然言語処理をしてみようかというときに、プログラムを作るのがたいへんで手控える傾向があったと思います。たしかに、むずかしいプログラムももちろん残っています。つまり、文脈解析のあたりは、まだよくわかっていません。しかし、これまで述べてきた方法によると、従来多くの人が行っていたシンタックティック、セマンティックなアナリシスのプログラムは、ほとんど書かないですみますから、言語に少し興味をもっている方が、自然言語処理の分野にすんなりとはいり込める時代になってきたと思います。

問題は、将来実用ということを考えて、知識が膨大になったときに、高速にインファレンスを行えるものでなければなりません。

「Prologマシーン＋知識ベースマシーン」ができれば、それが自然言語処理マシーンではないかと思います。

なかなかうまく説明できなくて苦労しましたが、これで終わらせていただきます。

Q&A

■Q■

今のお話のなかで、辞書の言語学的な枠組として、格文法が有望であるというお話があったと思います。格文法をとった場合、言語のどういう側面を処理するうえで有効であるか、その点を教えていただけませんか。

プレディケート（述語）がとるアーギュメントには、その深層格の指定の他に、シンタックティックリストリクションやセマンティックリストリクションがかせられています。先ほど、長尾先生のお話で「あげる」のような同音多義語の例がでていました。それを区別するうえでセマンティックリストリクションは有効であるが、その深層格の指定に関する限り、提示された三つの用例は、いずれも同一です。深層格が同一である限り、同音多義語の弁別には深層格は役立っていません。そうであるとすると、どういう点で深層格の指定が寄与するか、具体的な例を教えていただけませんか。

●A●

それについては、私は格文法だけで行わなければいけないと主張するつもりはありませんが、多くの人がなぜ格文法を使うかがわかれば、少し質問にお答えできるのではないかと思います。確かに、深層格としてどういうセットを用意したらよいかについては、いろいろ議論があります。深層格そのものがあるのかという議論もあります。

おそらく機械翻訳を行っている人たちは、言語学をやられている方でも、多くの格を設定しています。深層格は最初7個ぐらいで出発したと思いますが、実際の言語現象にあたってみると、格の役割を細かくしたほうが都合のよい場合があります。例えば、今の質問で深層格をまったく使ってないといわれますが、そうではありません。深層格とそれがもっている意味的な性質を複合して、訳語の選択などに使うことができます。深層格を抽出しても、それをまったく使っていないわけではありません。

問題は、いくつ深層格があるかということです。この点については言語学者は議論を諦めているようなところがあります。私の知っている深層格の数だと、約30個、長尾先生のシステムでも30個近いわけで、従来、計算機サイドの人も、多数の格を設定しても、それらをすべてマニピュレイトするようなソフトウエアが作れるかどうかが、問題でした。

それに関しても、やってみるとけっこうできます。その意味では、今の機械翻訳システムの作成者たちがとっている態度は、とにかくふやしてみて、それから減らすことができれば減らすという、たぶんに試行錯誤的な方法です。

例えば、私は以前takeについて調べました。I take a planeの場合の「take」は「飛行機に乗る」ですがI take a plane to Moscow.のようにto Moscowという格があると、単に前置詞句のなかでも、to Moscowですからゴールです。ゴールがあるとそれは、「乗る」だとまずくて、「乗っていく」と訳さなければいけません。そういうことがあって、格を抽出することは、翻訳に必要な情報を与えていることになります。●