知識情報処理言語PROLOG（その4）

大阪大学 大型計算機センター　馬野　元秀

1．はじめに

「その1（大阪大学 大型計算機センター ニュース、Vol.14、No.2（1984 年 8 月号、第 54 号）、pp.37-44）」では、Prolog の入門として、基本的な事項について説明し、「その2（同、Vol.15、No.4（1986 年 2 月号、第 60 号）、pp.185-194）」では、階乗を求めるプログラムと append のプログラムについて述べた。そして、「その3（同、Vol.16、No.2（1986 年 8 月号、第 62 号）、pp.185-194）」では、センターで使用できる ShapeUp という Prolog の具体的な使い方について述べた。

今回は、Prolog の応用として、簡単な自然言語の処理について述べよう。

2．Prolog による自然言語の処理

自然言語（普通の日本語や英語のこと）を計算機で取り扱うのは、そう簡単ではない。特に、自然言語の意味を計算機で取り扱うのは非常に難しい。最近、各種の計算機上で実現されている機械翻訳システムは、意味処理の試みを行なっているが、まだまだ十分なものとは言えない。

本稿では、簡単な英語の構文解析について考える。構文解析では、単語の並び方がどのようになっているかだけを調べ、文全体で何を表しているかという意味については考えない（これを行なうのが意味解析である）。構文解析は、自然言語処理のための必須の技術であり、普通は、構文解析を行なった後に、意味解析を行なう。

2.1. 構文規則

構文の規則は、生成規則（または書き換え規則）で記述される。これは、例えば、

(a) 文 → 名詞句 動詞句
(b) 動詞句 → 自動詞
(c) 動詞句 → 他動詞 目的語
(d) 目的語 → 名詞句
(e) 名詞句 → 固有名詞
(f) 名詞句 → 所有格代名詞 普通名詞
(g) 自動詞 → runs
(h) 他動詞 → likes
(i) 固有名詞 → John
(j) 所有格代名詞 → my
(k) 普通名詞 → dog

のようなもの[1]で、例えば、(a)で「文は、名詞句と動詞句を並べたものである」を表わし、(b)と(c)で「動詞句は、自動詞だけか他動詞と目的語を並べたものである」を表わす。これをなぜ生成規則や書き換え規則と呼ぶかというと、これらの規則を使って、次々と書き換えていくことにより文を生成することができるからである。

書き換えは、

文

から始める。「文」を書き換えることができるのは、→ の左側が「文」である規則(a)である。したがって、「文」は

名詞句 動詞句

に書き換えられる。これを

文 −(a)→ 名詞句 動詞句 (1)

と書くことにする。次に、名詞句と動詞句を書き換えることができる規則を探すと、名詞句に対しては規則(e)と(f)がみつかり、動詞句に対しては(b)と(c)がみつかる。名詞句と動詞句のどちらを先に書き換えるか、また、それぞれに対して、2つのうちのいずれの規則を用いるかという問題があるが、今は、いずれも前の方を用いよう。(1)の名詞句に規則(e)を適用すると、

名詞句 動詞句 −(e)→ 固有名詞 動詞句 (2)

となる。これに対して、さらに規則を適用していくと、

固有名詞 動詞句 −(i)→ John 動詞句 (3)
John 動詞句 −(b)→ John 自動詞 (4)
John 自動詞 −(g)→ John runs (5)

となり、これ以上書き換えることができなくなる。最後を見ると、“John runs”という文が生成されている。

書き換えのときに、2つ以上の規則から1つを選択する場合が何度かあった。今は、すべて 1 番目の規則を選んだが、それぞれの場合に、別の規則を選ぶことにより異なる文を生成することができる。例えば、(4)で(b)の代わりに(c)を選ぶと、

John 動詞句 −(c)→ John 他動詞 目的語 (4')
John 他動詞 目的語 −(h)→ John likes 目的語 (5')
John likes 目的語 −(d)→ John likes 名詞句 (6')
John likes 名詞句 −(e)→ John likes 固有名詞 (7')
John likes 固有名詞 −(i)→ John likes John (8')

となる。また、(7')で、(e)の代わりに(f)を選択すると、

```
John likes 名詞句 −(f)→ John likes 所有格代名詞 普通名詞                (7'')
John likes 所有格代名詞 普通名詞 −(j)→ John likes my 普通名詞           (8'')
John likes my 普通名詞 −(k)→ John likes my dog                          (9'')
```

が生成される。

さらに、(2) で規則(f) を使うと、"my dog runs" と "my dog likes John" と "my dog likes my dog" が得られる(実際の生成は読者自身で試みていただきたい)。

2.2. Prolog による構文解析

生成規則による文の生成の様子をみていると、Prolog の動作とよく似ているのに気が付く。もともと Prolog は、フランス語を効率よく処理するために考えだされた[2]という歴史があり、自然言語の構文解析は Prolog の最も得意とするものの１つである。上の生成規則を Prolog に直すと、

```
sentence([*np,*vp]) :- noun_phrase(*np), verb_phrase(*vp).   /* sentence：文 */
verb_phrase(*vi) :- intransitive_verb(*vi).          /* verb-phrase：動詞句 */
verb_phrase([*vt,*obj]) :- transitive_verb(*vt), object(*obj).
object(*np) :- noun_phrase(*np).                          /* object：目的語 */
noun_phrase(*pn) :- proper_noun(*pn).                  /* noun-phrase：名詞句 */
noun_phrase([*pp,*n]) :- possessive_pronoun(*pp), noun(*n).
intransitive_verb(runs).                  /* intransitive-verb：自動詞 */
transitive_verb(likes).                   /* transitive-verb：他動詞 */
proper_noun(John).                        /* proper-noun：所有格代名詞 */
possessive_pronoun(my).                   /* possessive-pronoun：普通名詞 */
noun(dog).                                /* noun：文 */
```

となる。ここでは、文をリストにより表わしている。このプログラムをみると、生成規則とほとんど同じであることが分かる。これを実行させると、

```
>?-sentence(*s).
*s = [John,runs]
?;↵
*s = [John,[likes,John]]
?;↵
*s = [John,[likes,[my,dog]]]
?;↵
*s = [[my,dog],runs]
?;↵
*s = [[my,dog],[likes,John]]
?;↵
*s = [[my,dog],[likes,[my,dog]]]
?;↵
NO!

>
```

となる(下線を引いた部分はユーザの入力を表わす。なお、ShapeUp を起動し、プログラムを入力・修正していく方法については、「その３」を参照されたい)。 このとき、構造を反映した形で文が生成されていることに注意しよう。しかし、このままでは、語彙があまりにも貧弱なので、単語を適当に追加して、いろいろと試みられたい。

さて、上のプログラムは、文を生成するだけではなく、与えられた文を解析して、文法にあっているかどうかを判定するのにも使うことができる。

```
>?-sentence([john,runs]).                    /* 何も出力されないのは、yes */

>?-sentence([john,walks]).
NO!

>?-sentence([john,likes,my,dog]).
NO!

>
```

これを見ると、[john,likes,my,dog] が no になってしまっている。これは yes にしたい。そのためには、プログラムの各節で、右辺の結果を左辺でリストとしてまとめているのを、append を使ってまとめてから左辺に渡せばよい。すなわち、

```
sentence(*s) :- noun_phrase(*np), verb_phrase(*vp), append(*np,*vp,*s).
verb_phrase(*vi) :- intransitive_verb(*vi).
verb_phrase(*vp) :- transitive_verb(*vt), object(*obj), append(*vt,*obj,*vp).
object(*np) :- noun_phrase(*np).
noun_phrase(*pn) :- proper_noun(*pn).
noun_phrase(*np) :- possessive_pronoun(*pp), noun(*n), append(*pp,*n,*np).
intransitive_verb([runs]).
transitive_verb([likes]).
proper_noun([John]).
possessive_pronoun([my]).
noun([dog]).
```

とすればよい。このとき、単語もリストにしておく必要があることに注意しよう。さらに、リストの append を追加しておく必要もある（プログラムは「その２」を参照のこと。ただし、英小文字を使うために shp1 で起動しているときは、大文字と小文字は異なるものとみなすので、注意する必要がある)。

しかし、append するなら、差リストを使えば、高速に処理できる。差リストの append は、

```
append([*1, *2], [*2, *3], [*1, *3]).
```

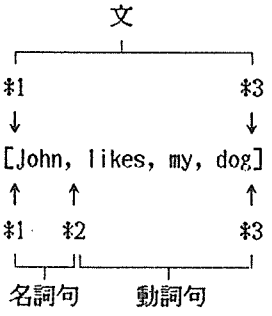
であった([その２]の差リストの append の変数名をかえてある)。これは [*1,*2] と [*2,*3] から [*1,*3] を作るだけであるから、append としてわざわざ定義しなくても、例えば、

```
sentence([*1,*3]) :- noun_phrase([*1,*2]), verb_phrase([*2,*3]).
```

とすればよい。さらに、すべての述語に [ ] が付いているので、これを省略して、

```
sentence(*1,*3) :- noun_phrase(*1,*2), verb_phrase(*2,*3).
```

とすることができる。これは、各変数がリストの次のような場所を指しているポインタと考えれば理解しやすい。



すなわち、*1 から *3 までが 文(sentence)であるためには、*2 を考えて、*1 から *2 までが名詞句(noun_phrase)であり、*2 から *3 までが 動詞句(verb_phrase)であればよい、という訳である。これによるプログラムは、

```
sentence(*1,*3) :- noun_phrase(*1,*2), verb_phrase(*2,*3).
verb_phrase(*1,*2) :- intransitive_verb(*1,*2).
verb_phrase(*1,*3) :- transitive_verb(*1,*2), object(*2,*3).
object(*1,*2) :- noun_phrase(*1,*2).
noun_phrase(*1,*2) :- proper_noun(*1,*2).
noun_phrase(*1,*3) :- possessive_pronoun(*1,*2), noun(*2,*3).
intransitive_verb([runs|*1],*1).
transitive_verb([likes|*1],*1).
proper_noun([John|*1],*1).
possessive_pronoun([my|*1],*1).
noun([dog|*1],*1).
```

となる。このとき、単語も差リストのような形にしておく必要がある。これを実行させると、

```
>?-sentence(*s,[]).
*s = [John,runs]
?;↵
*s = [John,likes,John]
?;↵
*s = [John,likes,my,dog]
?;↵
```

```
$s = [my,dog,runs]
?;↵
$s = [my,dog,likes,John]
?;↵
$s = [my,dog,likes,my,dog]
?;↵
NO!

>
```

となる。問い合わせも、差リストのような形にする必要がある。また、与えられた文の解析は、

```
>?-sentence([John,likes,my,dog],[]).

>?-sentence([John,loves,my,dog],[]).
NO!

>
```

とすればよい。さらに、名詞句や動詞句を生成させることもできる。

```
>?-noun-phrase($np,[]).
$np = [John]
?;↵
$np = [my,dog]
?;↵
NO!

>?-verb-phrase($vp,[]).
$vp = [runs]
?;↵
$vp = [likes,John]
?;↵
$vp = [likes,my,dog]
?;↵
NO!

>
```

これは、プログラムのデバッグに便利な機能である。

さて、上のプログラムを見ていると、生成規則からプログラムを機械的に作成できそうである。実際、生成規則に近い形のものを入力し、それを自動的に Prolog のプログラムに変換する機能を持っているシステムもある。これは、確定節文法 DCG (definite clause grammar) と呼ばれ、

```
sentence --> noun_phrase, verb_phrase.
verb_phrase --> intransitive_verb.
verb_phrase --> transitive_verb, object.
object --> noun_phrase.
```

```
noun_phrase --> proper_noun.
noun_phrase --> possessive_pronoun, noun.
intransitive_verb --> [runs].
transitive_verb --> [likes].
proper_noun --> [John].
possessive_pronoun --> [my].
noun --> [dog].
```

のように書かれるのが普通である。単語は他のものと区別するために [ ] でくくられる。 DCG から Prolog への変換を行なうプログラムは、Prolog でも書くことができる[3]。 ただし、ShapeUp には演算子を定義する機能がないので、このままの形の DCG を ShapeUp に変換するプログラムを ShapeUp で作成するのは、少し面倒である。

さて、いままで、入力にはリストを用いてきたが、できれば

```
John likes my dog.
```

のような自然な形で入力したい。そのためには、1 文字ずつ読み込み、リストにまとめあげればよい。これについては、ここでは省略するが、英文はスペースにより単語に分けられているので、それほど難しくはない(日本語に比べて)。

また、このままでは、主部の名詞句と述部の動詞句は 独立に調べているので、3単現の s のチェックもできない。 これは、名詞句の解析結果を動詞句に渡すための変数 *ai を追加して、例えば、

```
sentence(*1,*3) :- noun_phrase(*1,*2,*ai), verb_phrase(*2,*3,*ai).
```

とすればよい。3単現の実際の情報は、単語のところに書かれることになる。

さらに、今のプログラムでは、文法にあった文を生成したり、与えられた文が文法にあっているかのチェックを行なうことはできるが、意味解析のための情報はまったく得られていない。しかし、このためには、どのような情報を収集すればよいか、それをどのような形で表現すればよいか等の問題がある。 DCG では、{ } でくくって、Prolog の文を書くことができ、各種の追加的な処理を直接記述できるようになっている。

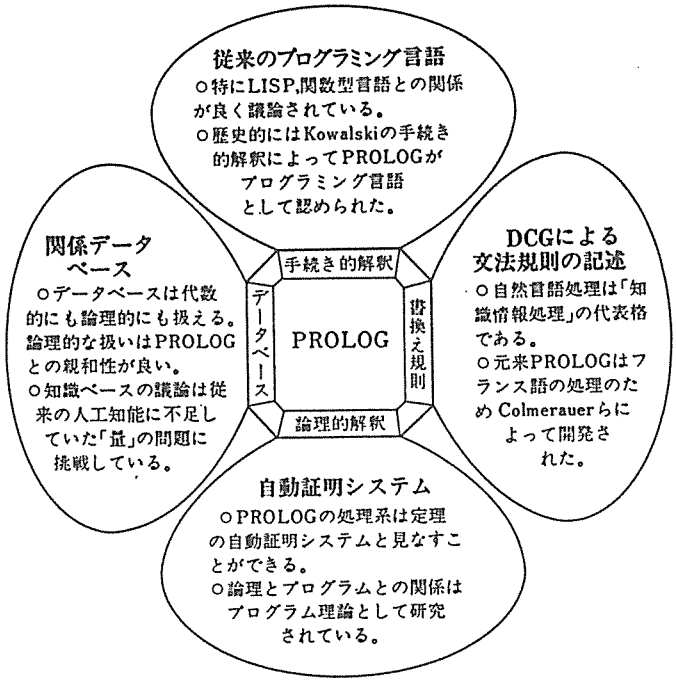
自然言語処理のうち、生成規則としてきちんと書ける部分は、Prolog を用いると、容易に処理できそうである。実際、かなりの部分が処理可能であるが、構文のあいまいさ（1つの文が数通りに構文解析できてしまう）があったり、省略や特殊な使い方などがあったりして、そう簡単には処理できない部分も多い。また、意味解析については、自然言語の意味をどのように表現すればよいかについて、まだ十分に分かっていないということもあり、まだまだ研究すべきことは多い（もちろん、いろいろな試みがなされている)。

3．おわりに

以上、Prolog による簡単な自然言語の構文解析について述べた。Prolog の強力な機能の一端が伺えたのではないかと思われる。

今回で、この Prolog の解説を終わることにする。知識情報処理の匂いを少しでも臭ぎ取って頂ければ幸いである。もっと早いペースで掲載するつもりであったが、回数の割には長期に渡ってしまった。この間に、多くの解説書も出版された（「その２」の参考文献にそのうちのいくつかをのせた）ので、Prolog に興味を持たれた方はそれらを参照されたい。

最後に、Prolog の多面性を表わす図[1]を示して、本稿を終わりたい。



図．Prolog の多面性[1]

[参考文献]

1．後藤(1984)：PROLOG 入門　知識情報処理の序曲、186 ページ、ソフトウェア ライブラリ 1、サイエンス社。

2．A. Colmerauer, H. Kanoui, R. Pasero and P. Roussel (1973) : "Un System de Communication Homme-Machine en Francais, Rapport de Recherche, Groupe d'Intelligence Artificielle, UER de Luminy, Universite d'Aix-Marseille.

3．溝口 監修 (1985)： Prolog とその応用２、318 ページ、総研出版。