The University of Osaka
Institutional Knowledge Archive

Title	知識情報処理言語PROROG(その2)
Author(s)	馬野, 元秀; 溝口, 理一郎
Citation	大阪大学大型計算機センターニュース. 1986, 60, p. 185-194
Version Type	VoR
URL	https://hdl.handle.net/11094/65685
rights	
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

資　料

知識情報処理言語PROLOG（その2）

大阪大学·大型計算機センター　馬野　元秀
大阪大学 産業科学研究所　溝口　理一郎

1．はじめに

前回（第54号、Vol.14、No.2、pp.37-44、1984年8月号）は、Prolog の入門として、事実、ルール、パターン・マッチング、バックトラック、リストなどについて一通り説明し、最後に member と append のプログラムを示した。そして、今回は Prolog による自然言語の構文解析に話を進める予定であったが、再帰的定義になじみのない読者も多いと思われるので、もう少し回り道をして今回は階乗を求めるプログラムと append のプログラムを少し詳しく調べてみよう。今回もプログラムはセンターで使用できる ShapeUp という Prolog で記述する。なお、ShapeUp の具体的な使い方については次号で説明する予定であるが、現在のところは文献[1]を参照されたい。

2．階乗を求めるプログラム

負でない整数 n の階乗は、よく知られているように、

$$n! = 1 \cdot 2 \cdot \cdots \cdot (n-1) \cdot n$$

で定義され、例えば、Fortran 77 では繰り返しを用いて、

```
      INTEGER FUNCTION FACT(N)
      INTEGER N, J
      FACT = 1
      DO 10 J = 1, N
         FACT = J * FACT
   10 CONTINUE
      END
```

で計算することができる（ただし、n<0 のときのことは考えていない）。

Prolog でも、これと同じような形で計算することもできるが、普通は、n! が

$$n! = \begin{cases} 1 & n=0 \text{ のとき} \\ n \cdot (n-1)! & n>1 \text{ のとき} \end{cases}$$

と定義できることを利用して、

```
factorial(1, 0).                                          (2.1)
factorial(*nf, *n) :- -(*m, *n, 1),
                      factorial(*mf, *m),                 (2.2)
                      *(*nf, *n, *mf).
```

のように書くのが普通である（見易さのために、アルファベットは主に小文字を用いる）。ここで、-(*m, *n, 1) と *(*nf, *n, *mf) は組み込み述語で、前者は「*m は *n から 1 を引いたものである」ことを表わし、後者は「*nf は *n と *mf をかけたものである」ことを表わす。そして、節(2.1) は「0! が 1 である」ことを意味しており、節(2.2) は「*n の階乗 *nf を求めるには、*n-1 である *m の階乗 *mf を求めて、*n と *mf の積を求めればよい」ことを意味している。節(2.2) は factorial を定義するのに factorial を使用しているので、このような定義を<u>再帰的定義</u>という。これに対して、

```
?- factorial(*a,3).                                       (2.3)
```

という問いかけをすると、

```
*a = 6
```

となって成功する。

この実行過程を追跡してみよう。まず、(2.3)を入力すると、節(2.1)とのパターン・マッチングには失敗するが、節(2.2)とは、

```
{*nf = *a, *n = 3}
```

という代入が行なわれて成功する。したがって、次に実行すべきゴール列は節(2.2)の右辺にこの代入を行なった

```
-(*m, 3, 1), factorial(*mf, *m), *(*a, 3, *mf).
```

となる。まず、-(*m, 3, 1) はすぐに計算できて、*m=2 となる。したがって、新しいゴール列は、

factorial(*mf, 2), *(*a, 3, *mf). (2.4)

となり、次に factorial(*mf, 2) を実行することになる。これは、(2.3)の場合と同様にして

{*nf = *mf, *n = 2} (2.5)

という代入が行なわれて、節(2.2)が成功する。そして、節(2.2)の右辺にこの代入を行なったもの、

-(*m, 2, 1), factorial(*mf, *m), *(*mf, 2, *mf).

でゴール列(2.4)の factorial(2, *mf) を置き換えたものになり、新しいゴール列は

-(*m, 2, 1), factorial(*mf, *m), *(*mf, 2, *mf), *(*a, 3, *mf).

となる。 しかし、これを見ているとおかしなことに気がつく。それは3番目の述語 * の変数 *mf である。 この述語には、変数 *mf が2つあるが、後の *mf は1つ前の述語 factorial から得られた結果を利用するものであるが、前の *mf は代入により伝わって来たもので、もとは (2.4) の *mf であった。このように異なる意味をもつ変数が同じ *mf で表わされている。

Prolog では変数の有効範囲は1つの節の中だけであった。今のようにパターン・マッチングが同じ節と2回以上成功したときでも、1回目と2回目は異なる変数となる。 これを同じ変数であると考えたために、おかしなことが起こってしまった訳である。計算機の中ではパターン・マッチングのたびごとに変数は異なるものとし処理をしてくれるが、我々がプログラムを追跡するときには注意しなければならない。普通は、同じ節が2回目以降に使われたときには、変数に ' を付けて区別する。すると、節(2.2)は、

```
factorial(*nf', *n') :- -(*m', *n', 1),
                        factorial(*mf', *m'),
                        *(*nf', *n', *mf').
```
(2.6)

となり、 (2.5)の代入は

{*nf' = *mf, *n' = 2} (2.7)

となり、節(2.2)の右辺にこの代入を行なったものは、

-(*m', 2, 1), factorial(*mf', *m'), *(*mf, 2, *mf').

となり、新しいゴール列は

```
-(*m', 2, 1), factorial(*mf', *m'), *(*mf, 2, *mf'), *(*a, 3, *mf).
```

となる。これを見ると、先ほどの問題点が解消しているのが分かる。

以下同様に、処理を続けると、

```
-(*m', 2, 1), factorial(*mf', *m'), *(*mf, 2, *mf'), *(*a, 3, *mf).
    (組み込み述語とマッチ、代入：{*m' = 1} )
→ factorial(*mf', 1), *(*mf, 2, *mf'), *(*a, 3, *mf)
    (節(2.2)とマッチ、代入：{*nf'' = *mf', *n'' = 1} )
→ -(*m'', 1, 1), factorial(*mf'', *m''), *(*mf', 1, *mf''),
                                        *(*mf, 2, *mf'), *(*a, 3, *mf).
    (組み込み述語とマッチ、代入：{*m'' = 0} )
→ factorial(*mf'', 0), *(*mf', 1, *mf''), *(*mf, 2, *mf'), *(*a, 3, *mf).
    (節(2.1)とマッチ、代入：{*mf'' = 1} )
→ *(*mf', 1, 1), *(*mf, 2, *mf'), *(*a, 3, *mf).
    (組み込み述語とマッチ、代入：{*mf' = 1} )
→ *(*mf, 2, 1), *(*a, 3, *mf).
    (組み込み述語とマッチ、代入：{*mf = 2} )
→ *(*a, 3, 2).
    (組み込み述語とマッチ、代入：{*a = 6} )
```

となり、成功して、変数 *a に 6 を得る。

以上、階乗を計算するプログラムを例として、再帰的に定義されたプログラムの実行過程について述べた。このようなことは異なる節で同じ変数名を使っている場合の追跡においてもあてはまる（普通は、そのような形のプログラムになってしまう）。

本節では階乗のプログラムを Prolog で記述したが、実はこれはあまり Prolog のプログラムらしくない。 再帰的定義による階乗のプログラムならば Pascal や C で書いた方が、分かり易く、すっきりと書けると思われる。そこで、より Prolog のプログラムらしい append のプログラムに話を移そう。

3．append プログラム

append は２つのリストを連結して１つのリストにするもので、例えば、

```
?- append([a, b], [c, d, e], *a).                    (3.1)
```

を実行させると、

```
*x = [a, b, c, d, e]
```

となる。念のために、復習しておくと、[a, b] や [c, d, e] や [a, b, c, d, e] はリストで、任意個の要素を並べたものであった。そして、要素がまたリストであってもよく、要素が 0個のリストは空リストと呼ばれ、[] と書かれる。リストの最初の要素（ヘッド）と残りのリスト（テール）に分けるには、パターンとして、[*h | *x] と書けばよかった。また、パターン [*h | *x] は要素 *h をリスト *x の先頭に結合したリストをも意味する。

さて、append のプログラムは、前回の最後にも出したように、非常に簡単で、

```
append([], *x, *x).                                   (3.2)
append([*h | *x], *y, [*h | *z]) :- append(*x, *y, *z).   (3.3)
```

となる。節(3.2) は「空リストにどんなリスト *x を連結しても、そのリスト *x である」ということを表わし、節(3.3)は「1番目のリストから先頭の要素を取り除いたリスト *x と 2番目のリスト *y とを連結したリストが *z であるので、これと 1番目のリストの最初の要素 *h を結合したリストが結果のリストとなる」ということを表わしている。これはまた「リスト *x と *y の連結したものがリスト *z であるならば、*x に *h を付け加えたリストと *y というリストとの連結は *z に *h を付け加えたものに等しい」と考えてもよい。

さて、(3.1)の実行を追跡してみよう。

```
 ?- append([a, b], [c, d, e], *a).
        (節(3.3)とマッチ、代入：{*h = a, *x = [b], *y = [c, d, e], [a | *z] = *a} )
 →  append([b], [c, d, e], *z).
        (節(3.3)とマッチ、代入：{*h' = b, *x' = [], *y' = [c, d, e], [b | *z'] = *z} )
 →  append([], [c, d, e], *z').
        (節(3.2)とマッチ、代入：{*x'' = [c, d, e], *x'' = *z'} )
```

となって成功する。このとき、リスト [b] とパターン [*h | *x] とのマッチングは成功し、変数 *h が b に、変数 *x が空リスト [] になることは注意を要する（上の場合には、変数に ' が付

いていたが）。

この場合、成功したことは節(3.2) の右辺がないことから分かるが、問い合わせ中の変数 *a の値はどうなるのだろうか。それを知るためには、代入を逆にたどっていけばよい。まず、最後の代入 {*x'' = [c, d, e], *x'' = *z'} から、 *z' = *x'' = [c, d, e] が分かり、その1つ前の代入から *z = [b | *z'] が分かり、*z = [b | [c, d, e]] = [b, c, d, e] となる。そして、これを1つ上の *a = [a | *z] に代入して、 *a = [a | [b, c, d, e]] = [a, b, c, d, e] となる。これで，変数 *a の値が得られた。 これは、確かにリスト [a, b] と [c, d, e] とを連結したものになっている。 [a | *z] = *a のような代入が可能であるというのも、Prolog の特徴の1つである。

append は、 もちろん、2つのリストを連結するのに使用するが、リストを分割するのに使うこともできる。例えば、

```
?- append([a, b, c], *a, [a, b, c, d, e]).          (3.4)
```

を実行させると、

```
*a = [d, e]
```

となって成功するし、

```
?- append(*a, [d, e], [a, b, c, d, e]).             (3.5)
```

を実行させると、

```
*a = [a, b, c]
```

となって成功する。(3.4)と(3.5)に対する実行過程を付録に示しておくので、参照されたい。もちろん、

```
?- append([a, x], *a, [a, b, c, d, e]).
?- append([a, c, b], *a, [a, b, c, d, e]).
?- append(*a, [x], [a, b, c, d, e]).
?- append(*a, [e, d], [a, b, c, d, e]).
```

などは、失敗する。

さらに、

```
?- append(*a, *b, [a, b, c, d, e]).                    (3.6)
```

のように、2ヶ所に変数があるものも実行できる。これは、まず(3.2)とマッチし、

```
*a = [],    *b  = [a, b, c, d, e]
```

が得られる。さらに処理を続けさせる（その方法は処理系により異なる）と、節(3.3)とマッチし、その右辺が節(3.2)とマッチして、

```
*a = [a],    *b  = [b, c, d, e]
```

が得られる。さらに、処理を続けていくと、順次、

```
*a = [a, b],    *b  = [c, d, e]
*a = [a, b, c],    *b  = [d, e]
*a = [a, b, c, d],    *b = [e]
*a = [a, b, c, d, e],    *b = []
```

が得られる。

いままでは、普通のリストに対する連結のプログラムについて述べてきたが、データ構造を工夫することによりさらに高速にリストの連結を行なうことができる。このためのデータ構造が、差リスト（difference list）と呼ばれるものである。これは、言語の構文解析を行なう際にも重要な働きをする。

これは、名前の通り２つのリストの差で１つのリストを表わすものである。例えば、

```
[[a, b, c, d, e], [d, e]]
```

で、[a, b, c] というリストを表わすことになる。さらに、後の部分は何でもよいので変数にして、

```
[[a, b, c | *x], *x]
```

のようにしてもよい。

すると、後の部分が変数の差リストの連結プログラムは、

```
append([*x, *y], [*y, *z], [*x, *z]).                  (3.7)
```

のような 1行だけの簡単なものになる。

これの意味を調べる前に、簡単な例を実行してみよう。

```
?- append([[a, b, c | *t], *t], [[d, e | *u], *u], [*a, *b]).
```

を実行すると、代入

```
{*x = [a, b, c | *t], *y = *t, *y = [d, e | *u], *z = *u, *x = *a, *z= *b}
```

で成功する。変数がたくさんあるが、結果を求めるのに必要なものは *a と *b であるので、これに関する代入を調べると、

```
*a = *x = [a, b, c | *t] = [a, b, c | *y] = [a, b, c | [d, e | *u]]
   = [a, b, c, d, e | *u]

*b = *z = *u
```

となり、結局のところ、結果は

```
[*a, *b] = [[a, b, c, d, e | *u], *u]
```

となる。 これは、意味的には、リスト [a, b, c, d, e] を表わしており、思い通りの結果になっている。

何やら騙されたような感じを持つかもしれないが、 これは、リスト [a, b, c | *t] の変数 *t を [d, e | *u] で置き換えると、 [a, b, c | [d, e | *u]] = [a, b, c, d, e | *u] となることが分かれば、すぐに理解できる。(3.2)と(3.3)の普通のリストの append は、後のリストに前から1つずつ要素を結合して、リストとリストを連結したのに対して、差リストの場合は、前の差リストの最後の要素が変数であることを利用して、その変数に後の差リストを代入して、直接連結してしまった訳である。 したがって、(3.7) の差リストの append を使うときには、1番目の差リストの後の部分は変数でなければならない（2番目は定数でもよい）。

以上、 append のプログラムについて述べた。普通のリストに対するものも、差リストに対するものも、 append のプログラムは Prolog の特徴をうまく利用した、非常に Prolog らしいプログラムである。

4．おわりに

今回は、階乗を求めるプログラムと append のプログラムを詳しく調べた。 append のプログラ

ムは、Prolog の特徴をうまく利用した、非常に Prolog らしいプログラムである。 このようなプログラムは「知識」情報処理という感じがあまりしないかもしれないが、再帰的定義とリスト処理を理解することは知識情報処理においては必須である。次回はセンターで使用できる Prolog である ShapeUp の具体的な使い方について説明しよう。

なお、最近、日本語の Prolog 関係の本がたくさん出版されるようになった。いくつかを参考文献[2-11]にまとめておくので、参照されたい。

[参考文献]

1. 日本電気株式会社（1984）：「 ShapeUp 仕様書（V1）」、58p.。
2. W.F. Clocksin and C.S. Mellish（1981）："Programming in Prolog", Springer-Verlag.
   訳：中村（1983）：「Prolog プログラミング」、マイクロソフトウェア。
3. 中島（1983）：「Prolog」、165p.、コンピュータ・サイエンス・ライブラリー、産業図書。
4. 後藤（1984）：「PROLOG 入門 － 知識情報処理の序曲」、186p.、ソフトウェア・ライブラリ１、サイエンス社。
5. 太細、鈴木、伊草、佐藤（1984）：「Prolog入門」、228p.、啓学出版。
6. 中島（1985）：「知識表現と Prolog/KR」、179p.、コンピュータ・サイエンス・ライブラリー、産業図書。
7. 安部（1985）：「Prolog プログラミング入門」、195p.、共立出版。
8. 溝口 監修（1985）：「Prolog とその応用２」、318p.、総研出版。
9. D. Li（1984）："A PROLOG Database System", 207p., Research Studies Press, Letchworth, UK.
   訳：安部（1985）：「Prolog データベース・システム」、231p.、近代科学社。
10. 黒川（1985）：「Prolog のソフトウェア作法」、256p.、岩波コンピュータ・サイエンス・シリーズ、岩波書店。
11. 中村（1985）：「Prolog と論理プログラミング」、144p.、オーム社。

[付録]

<(3.4)の実行過程>

```
?- append([a, b, c], *a, [a, b, c, d, e]).
      (節(3.3)とマッチ、代入： {*h = a, *x = [b, c], *y = *a, *z = [b, c, d, e]} )
```

```
→ append([b, c], *a, [b, c, d, e]).
    (節(3.3)とマッチ、代入： {*h' = b, *x' = [c], *y' = *a, *z' = [c, d, e]} )
→ append([c], *a, [c, d, e]).
    (節(3.3)とマッチ、代入： {*h'' = c, *x'' = [], *y'' = *a, *z'' = [d, e]} )
→ append([], *a, [d, e])
    (節(3.2)とマッチ、代入： {*x''' = *a, *x''' = [d, e]} )

    *a = *x''' = [d, e]
```

<(3.5)の実行過程>

```
?- append(*a, [d, e], [a, b, c, d, e]).
    (節(3.3)とマッチ、
     代入： {[*h | *x] = *a, *y = [d, e], *h = a, *z = [b, c, d, e]}
         = {[a | *x] = *a, *y = [d, e], *h = a, *z = [b, c, d, e]} )
→ append(*x, [d, e], [b, c, d, e]).
    (節(3.3)とマッチ、
     代入： {[b | *x'] = *x, *y' = [d, e], *h' = b, *z' = [c, d, e]} )
→ append(*x', [d, e], [c, d, e]).
    (節(3.3)とマッチ、
     代入： {[c | *x''] = *x', *y'' = [d, e], *h'' = c, *z'' = [d, e]} )
→ append(*x'', [d, e], [d, e]).
    (節(3.2)とマッチ、代入： {[] = *x'', *x''' = [d, e]} )

    *a = [a | *x]
       = [a | [b | *x'] ]
       = [a | [b | [c | *x'']] ]
       = [a | [b | [c | []]]]
       = [a | [b | [c]]]
       = [a | [b, c]]
       = [a, b, c]
```