資　料

知識情報処理言語PROLOG（その3）

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

「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、事実、ルール、パターン・マッチング、バックトラック、リストなどの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。いずれも、プログラムはセンターで使用できる ShapeUp という Prolog で記述してきたが、具体的な使い方については触れなかった。そこで、今回は「その1」と「その2」の例を用いて、ShapeUp の具体的な使い方について述べよう。必要に応じて「その1」と「その2」を参照されたい。なお、Shape Up のマニュアルとしては文献[1]がある。

2．ShapeUp の使い方

2.1．システムの起動

ShapeUp の起動は、端末の種類により少し異なり、

○SHP1 － 英小文字を使用可能な ASCII 端末(N6950、パソコン端末など)
○SHP2 － 英大文字のみ使用可能な ASCII 端末
○SHP3 － JIS 端末(N6300 シリーズなど)

となる。本稿では、最も多くのユーザが使用していると思われる英小文字を使用可能な ASCII 端末の場合について説明する。そこで、

```
SYSTEM ?SHP1↵
*** SHAPEUP V 1.12.A6S started (please do ':-versionup.' to confirm) ***
>
```

とすると、ShapeUp システムが起動される(下線を引いた部分はユーザの入力を表わす。以下も同じ。また、↵ は「return」、「書き込み」、「送信」などのキィーを表わす。以下では、特に強調する場合以外は、入力の終わりの ↵ を省略する)。ShapeUp でのプロンプトは > である。

システムからのメッセージに従って、

```
>:-VERSIONUP.
```

とすると、

```
*** Version-Up information *** Welcome to ShapeUp V1.12.A6S ***

   This version is for the terminals which are able to use small and
 capital characters, such as, N6950, PC's, VT's , and so on.

 [News for this version.]
  1  Automatic "?-" is supported. Try to input ".".
  2  History is also supported. Try to input ":".
  3  Changed the signal for stpm continue from ";" to "Cr".

 Version      Date      Information
            3/27'85   changed the IBSIZE from 800 to 1200.
 1.12.A6S    2/28'85  updated for official release.
 1.11.A6S   10/21'83
 1.10.A6S    7/20'83
 1.00.A6S    2/ 1'83  ShapeUp 1st version.

>
```

という情報が得られる。3-4 行目の内容は、SHP1、SHP2、SHP3 で異なる。 News の 1、2、3 については、以下で説明する。

2.2. 基本的な使い方

さて、「その１」の例をもとにして(必ずしも「その１」のものと同じではない)、ShapeUp の実際の使い方について述べていこう。

```
>LOVES(TOM,SUSAN).

>
```

で、事実を登録できる。最後に . を入れるのを忘れないように注意しよう。

すると、yes/no の問い合わせは、

```
>?-LOVES(TOM,SUSAN).

>?-LOVES(TOM,NANCY).
NO!

>
```

とすればよい。yes のときには、何も表示されない。また、変数を含む問い合わせは、

```
>?-LOVES(TOM,*X).
*X = SUSAN
?⏎

>?-LOVES(*X,SUSAN).
*X = TOM
?⏎

>?-LOVES(*X,*Y).
*X = TOM
*Y = SUSAN
?⏎

>
```

となる。このとき、? のプロンプトが出るが、これについては後で説明するので、今は ⏎ だけを入力しておくことにする。

さて、事実とルールを追加しよう。 /* */ でかこまれた部分は ShapeUp の注釈である。

```
>LOVES(TOM,MARY).

>TOGETHER(TOM,MARY).

>KISS(*MAN,*WOMAN) :- LOVES(*MAN,*WOMAN),          /* 2 行に渡る節の入力 */
*                     TOGETHER(*MAN,*WOMAN).

>
```

節の終わりは . で判定しているので、 . を入れ忘れる(慣れるまではよく忘れる)と、* のプロンプトが出てしまう。そのときは、* のプロンプトに対して . のみを入力すればよい。

これに対して、いくつかの問い合わせを行なおう。いくつかの問い合わせを続けて行なうときには、?- モードを使うと便利である。

```
>.⏎                                  /* ?- モードに入る */
> ?- KISS(TOM,MARY).                 /* プロンプトが > ?- になる */

> ?- KISS(TOM,SUSAN).
NO!

> ?- KISS(TOM,*X).
*X = MARY
?⏎

> ?-
```

ここで、事実を1つ追加しよう。

```
> ?- .⏎                                    /* ?- モードから出る */
>LOVES(TOM,SISTER(JOHN)).

>
```

さて、現在の節の登録状態はどうなっているだろうか。 これをみるために LIST という述語(関数子)がある。

```
>:-LIST.
LOVES(TOM,SUSAN).
LOVES(TOM,MARY).
LOVES(TOM,SISTER(JOHN)).
TOGETHER(TOM,MARY).
KISS(*MAN,*WOMAN) :- LOVES(*MAN,*WOMAN),
                     TOGETHER(*MAN,*WOMAN).

>
```

:- で始まるものも、問い合わせの一種である。 述語 LIST では、節を左辺の述語ごとに集めた形(同じ述語の中では入力した順)で表示される。

いままでの例で分かるように、ShapeUp には、2 通りの問い合わせのしかたがある。それらは、解を 1 つだけ見つける方法(:- で節を始める)と複数個の解を見つける方法(?- で節を始める)である。また、:- では変数の値を表示しないが、?- では変数の値を表示する。

```
>:-LOVES(TOM,*X).                /* 解を 1 つだけ見つけ、変数の値を表示しない */

>?-LOVES(TOM,*X).                /* 解を次々と見つけ、変数の値を表示する */
*X = SUSAN
?;⏎                              /*  ;⏎ は「次の解を見つけよ」を意味する */
*X = MARY
?;⏎
*X = SISTER(JOHN)
?;⏎
NO!

>?-LOVES(TOM,*X).
*X = SUSAN
?;⏎
*X = MARY
?⏎                        /*  ⏎ のみは「これで解をさがすのをやめろ」を意味する  */

>
```

さらに、述語 LIST は引数をつけると、次のように特定の述語に関する節のみを表示することもできる。

```
>:-LIST(LOVES).                          /* 述語 LOVES に関する節のみ表示する */
LOVES(TOM,SUSAN).
LOVES(TOM,MARY).
LOVES(TOM,SISTER(JOHN)).

>
```

また、不必要な節は述語 RETRACTN で削除できる。

```
>:-RETRACTN(LOVES,2).                /* 述語 LOVES の 2 番目の節を削除 */

>:-RETRACTN(KISS,1).                 /* 述語 KISS の 1 番目の節を削除 */

>:-LIST.
LOVES(TOM,SUSAN).
LOVES(TOM,SISTER(JOHN)).
TOGETHER(TOM,MARY).

>
```

述語 RETRACTN の 2 番目の引数は、LIST の表示中の順番を表わす。

さらに、新しいプログラムを入力するために、今、登録されている節をすべて消去するには、

```
>:-CLEAR.                                /* すべての節を消去する */

>:-LIST.                                 /* 確認する */

>
```

とすればよい。

2.3. 階乗のプログラムと実行のトレース

さて、「その２」の階乗のプログラムを実行させてみよう。まず、プログラムを入力する。

```
>FACTORIAL(1,0).

>FACTORIAL(*NF, *N) :- -(*M, *N, 1),
*                      FACTORIAL(*MF, *M),
*                      *(NF, *N, *MF).          /* 誤りあり */

>
```

そして、実行させてみると、

```
>?-FACTORIAL(*A,3).
```

```
!! argument 0 must be integer. : *
!! LOCAL STACK   overflow.
please respond 'Cr' (don't modify) or integer (new max size).
        SIZE (100) => ⏎

>
```

のようにエラーになってしまう(エラー・メッセージの一覧とその説明は文献[1]の pp.47-53 にある)。 このエラー・メッセージをみると、どこかの述語の第 0 引数(引数は 0 から数える)が整数でなければならないのに、整数になっていないことが分かる。そして、それが原因で LOCAL STACK (実行のための作業用スタックと考えればよい)がオーバフローしているらしい。

プログラムを見ても誤りが分からない(分かりますか？)ので、実行過程をトレースしてみよう。このために、ShapeUp にはステップ・モードと呼ばれるものがある。

```
>:-STPMON.                          /* step mode on：ステップ・モードに入る */
<SUCCESS> STPMON

>?-FACTORIAL(*A,3).
[COMMAND] ?-FACTORIAL(*A,3).
<CALL>    FACTORIAL(undefine,3)
! [1] FACTORIAL(1,0).?⏎            /* 中断１。⏎ は「そのまま続きを実行せよ」*/
!       <UF-FAIL>
! [1] FACTORIAL(*NF, *N) :- -(*M, *N, 1),                    FACTORIAL(*MF
!       , *M),                      *(NF, *N, *MF).?⏎          /* 中断２ */
! <CALL>    -(undefine,3,1)
! <SUCCESS> -(2,3,1)
! <CALL>    FACTORIAL(undefine,2)
! ! [2] FACTORIAL(1,0).?⏎                                     /* 中断３ */
! !       <UF-FAIL>
! ! [2] FACTORIAL(*NF, *N) :- -(*M, *N, 1),                    FACTORIAL(
! !       *MF, *M),                   *(NF, *N, *MF).?⏎        /* 中断４ */
! ! <CALL>    -(undefine,2,1)
! ! <SUCCESS> -(1,2,1)
! ! <CALL>    FACTORIAL(undefine,1)
! ! ! [3] FACTORIAL(1,0).?⏎                                   /* 中断５ */
! ! !       <UF-FAIL>
! ! ! [3] FACTORIAL(*NF, *N) :- -(*M, *N, 1),                     FACTORI
! ! !       AL(*MF, *M),                    *(NF, *N, *MF).?⏎ /* 中断６ */
! ! ! <CALL>    -(undefine,1,1)
! ! ! <SUCCESS> -(0,1,1)
! ! ! <CALL>    FACTORIAL(undefine,0)
! ! ! ! [4] FACTORIAL(1,0).?⏎                                 /* 中断７ */
! ! ! <SUCCESS> FACTORIAL(1,0)
! ! ! <CALL>    *(NF,1,1)
!! argument 0 must be integer. : *
! ! ! <FAIL>
! ! ! <RETRY>   FACTORIAL(undefine,0)
! ! ! ! [4] FACTORIAL(*NF, *N) :- -(*M, *N, 1),                      FACT
! ! ! !       ORIAL(*MF, *M),                    *(NF, *N, *MF).?⏎  /*8*/
```

```
!  !  !  !  <CALL>    -(undefine,0,1)
!  !  !  !  <SUCCESS> -(-1,0,1)
!  !  !  !  <CALL>    FACTORIAL(undefine,-1)
!  !  !  !  !  [5]  FACTORIAL(1,0).?⏎                          /* 中断9 */
!  !  !  !  !        <UF-FAIL>
!  !  !  !  !  [5]  FACTORIAL(*NF, *N) :- -(*M, *N, 1),                  F
!  !  !  !  !        ACTORIAL(*MF, *M),                    *(NF, *N, *MF).?E⏎
!! LOCAL STACK   overflow.
please respond 'Cr' (don't modify) or integer (new max size).
        SIZE (100) => ⏎

>
```

トレースにおいて、 <CALL> は実行すべき述語を表わし、[1]や[2]などは実行のレベルを表わす。レベル[n]の右辺の述語を実行するためにパターン・マッチング(ユニフィケーション)を行なうときにレベルが[n+1]になる。 そして、このときに実行を中断し、ユーザにトレースに関する指示をあおぐ。 これに対して、⏎ のみを入力すると、ステップ・モードのまま続きを実行するが、E⏎ を入力すると、ステップ・モードから抜け出す(したがって、トレース情報を表示せずに、実行を続ける)。さて、上のトレース情報をみると、エラー・メッセージが *(NF,1,1) の実行中に出ているのが分かる。 そこで、この述語をよく見ると、*NF であるべきところが、NF になっているのが分かる。NF では整数とマッチできないので、このようなエラー・メッセージが出ていた訳である。

ShapeUp の中で、これを修正するには、この節を削除してから、新たに入れ直すしか方法がない。

```
>:-LIST.
FACTORIAL(1,0).
FACTORIAL(*NF, *N) :- -(*M, *N, 1),
                      FACTORIAL(*MF, *M),
                      *(NF, *N, *MF).

>:-RETRACTN(FACTORIAL,2).                /* FACTORIAL の 2 番目の節を削除する */

>FACTORIAL(*NF, *N) :- -(*M, *N, 1),
*                      FACTORIAL(*MF, *M),
*                      *(*NF, *N, *MF).

>
```

修正できたので、もう一度、実行させると、

```
>?-FACTORIAL(*A,3).
*A = 6
?;⏎
!! LOCAL STACK   overflow.
please respond 'Cr' (don't modify) or integer (new max size).
        SIZE (100) => ⏎
```

```
>
```

となる。 この場合には、次の解を求めようとすると、LOCAL STACK がオーバフローしてしまう(なぜか？)。確認のために、トレースさせてみると、

```
>?-STPMON,FACTORIAL(*A,3),STPMOFF.                    /* STPMOFF：step mode off */
<SUCCESS> STPMON
<CALL>    FACTORIAL(undefine,3)
!  [1]  FACTORIAL(1,0).?*⏎          /* 中断。*⏎ は「中断せずに続きを実行せよ」*/
!        <UF-FAIL>
!  [1]  FACTORIAL(*NF, *N) :- -(*M, *N, 1),                    FACTORIAL(*MF
!         , *M),                          *(*NF, *N, *MF).
!  <CALL>    -(undefine,3,1)
!  <SUCCESS> -(2,3,1)
!  <CALL>    FACTORIAL(undefine,2)
!  !  [2]  FACTORIAL(1,0).
!  !        <UF-FAIL>
!  !  [2]  FACTORIAL(*NF, *N) :- -(*M, *N, 1),                    FACTORIAL(
!  !        *MF, *M),                          *(*NF, *N, *MF).
!  !  <CALL>    -(undefine,2,1)
!  !  <SUCCESS> -(1,2,1)
!  !  <CALL>    FACTORIAL(undefine,1)
!  !  !  [3]  FACTORIAL(1,0).
!  !  !        <UF-FAIL>
!  !  !  [3]  FACTORIAL(*NF, *N) :- -(*M, *N, 1),                    FACTORI
!  !  !        AL(*MF, *M),                          *(*NF, *N, *MF).
!  !  !  <CALL>    -(undefine,1,1)
!  !  !  <SUCCESS> -(0,1,1)
!  !  !  <CALL>    FACTORIAL(undefine,0)
!  !  !  !  [4]  FACTORIAL(1,0).
!  !  !  <SUCCESS> FACTORIAL(1,0)
!  !  !  <CALL>    *(undefine,1,1)
!  !  !  <SUCCESS> *(1,1,1)
!  !  <SUCCESS> FACTORIAL(1,1)
!  !  <CALL>    *(undefine,2,1)
!  !  <SUCCESS> *(2,2,1)
!  <SUCCESS> FACTORIAL(2,2)
!  <CALL>    *(undefine,3,2)
!  <SUCCESS> *(6,3,2)
<SUCCESS> FACTORIAL(6,3)
<CALL>    STPMOFF
*A = 6
?⏎

>
```

となって、正しく動作しているのが分かる。中断時に、*⏎ を入れると、トレース情報は表示するが、中断せずに最後まで実行してくれる。 これを見ると、<CALL> と <SUCCESS> が ! の線を介して、きちんと対応しているのが分かる。

さて、このプログラムをファイルに蓄えよう。そのためには、

```
>:-TELL(FACT),RESAVE,CLOSE(FACT).   /* ファイル FACT にプログラムをセーブする */

>
```

とすればよい。ここで、TELL(FACT) は FACT というファイルを出力用に開くことを、RESAVE は現在のプログラムを LIST と同じ形でファイルに出力することを、CLOSE(FACT) は FACT というファイルを閉じることを指定する組み込み述語である。ただし、このときに作られるファイルはテンポラリ・ファイルである。これを、ShapeUp の中から確認するには、

```
>:-SYSTEM(FLIST).                          /* TSS の FLIST コマンドを実行する */
FILE.NAME (FC) MODE  SIZE(LLINK) CATALOG.STRING OR FILE TYPE
 SY**          RAND           2  TEMPORARY
 FACT          LINK          12  TEMPORARY

FLIST   normally terminated

>
```

とすることにより可能である。述語 SYSTEM の引数に、特殊文字(* ( ) , など)を含む場合は、引数全体を " " でくくればよい(この規則は、述語 SYSTEM に対してだけではなく、一般の述語に対しても有効である)。なお、テンポラリ・ファイルをパーマネント・ファイルにする方法については、文献[2]の p.35 を参照されたい。

これで、ひと区切りになるので、一度、ShapeUp を終了しよう。

```
>:-END.                                    /* ShapeUp を終了する */


*** ShapeUp version 1.12.A6S normally terminated ***
        (C&C Systems Research Lab's NEC)

SYSTEM ?
```

2.4. ファイルへの書き出しとファイルからの読み込み ー append プログラムを例にして

前節までの説明では、プログラムの誤りを見つけても、ShapeUp の中では、節を削除し、入れ直すしか方法がなかった。このままでは、プログラムのデバッグなど、とてもできないが、エディタと組合わせて使用する方法がある。「その２」の append プログラムを例にして、その方法について説明しよう。なお、最近ではスクリーン・エディタを用いるのが普通であるが、使用できる端末に制限があるので、本稿では、無手順端末からでも使用できる行エディタを用いることにする。

```
SYSTEM ?EDIT NEW                          /* 行エディタで新たにプログラムを作る */
ENTER
*APPEND([], *X, *Y).                      /* 誤りあり */
*APPEND([*H | *X], *Y, [*H | *Z]) :- APPEND(*X, *Y, *Z).
*↵
-SAVE AP                                  /* AP というファイルに書き出す */
DATA SAVED-AP

APPEND([], *X, *Y).
```

これで、APPEND のプログラムを AP というファイルに書き出すことができた。SHP3 により起動している場合(すなわち、JIS 端末を使用している場合)には、| に対応するキィーがないので、代わりに $ を使うことに注意しよう。 なお、エディタのサブコマンド中には注釈を書けないが、便宜上、ShapeUp と同じ形式で注釈を書くことにする。

そして、エディタの中から ShapeUp を起動する。

```
-SHP1                                     /* エディタの中から ShapeUp を起動する */

*** SHAPEUP V 1.12.A6S started (please do ':-versionup.' to confirm) ***

>:-SEE(AP),CONSULT,CLOSE(AP).             /* ファイル AP からプログラムを読み込む */

>:-LIST.
APPEND([], *X, *Y).
APPEND([*H | *X], *Y, [*H | *Z]) :- APPEND(*X, *Y, *Z).

>
```

ここで、SEE(AP) は AP というファイルを入力用に開くことを、CONSULT はファイルからプログラムを読み込む(現在のプログラムに追加する)ことを、CLOSE(AP) は前にも述べたように AP というファイルを閉じることを指定する組み込み述語である。

このプログラムを実行してみよう。

```
>?-APPEND([A,B], [C,D,E], *A).
*A = [A,B|undefine]
?;↵                                       /* 実行結果は誤っている */
NO!

>
```

となって、思いどおりの結果が出ない。しかし、このエラーはプログラムを見ると、すぐに分かり

(分かりますね？)、1 番目の節で、*X となるべきところが *Y となっている。このとき、ShapeUp の中では修正を行なわずに、いったん、エディタに返って修正し、ファイルに書き出してから、もう一度 ShapeUp を起動し、プログラム全体をファイルから読み込むようにする。

```
>:-END.                          /* いったん ShapeUp を終わる */


*** ShapeUp version 1.12.A6S normally terminated ***
        (C&C Systems Research Lab's NEC)


APPEND([], *X, *Y).

-LIST                        /* エディタに戻ると前の内容がそのまま残っている */
APPEND([], *X, *Y).
APPEND([*H | *X], *Y, [*H | *Z]) :- APPEND(*X, *Y, *Z).


APPEND([], *X, *Y).

-RVS:/Y/:/X/                     /* 文字列 Y を X に置き換える */
APPEND([], *X, *X).

-LIST
APPEND([], *X, *X).
APPEND([*H | *X], *Y, [*H | *Z]) :- APPEND(*X, *Y, *Z).


APPEND([], *X, *X).

-RESAVE AP                       /* ファイル AP にもう一度書き出す */
DATA SAVED-AP

APPEND([], *X, *X).

-SHP1                            /* エディタの中から ShapeUp を再起動する */

*** SHAPEUP V 1.12.A6S started (please do ':-versionup.' to confirm) ***

>:-SEE(AP),CONSULT,CLOSE(AP).    /* ファイル AP からプログラムを読み込む */

>:-LIST.
APPEND([], *X, *X).
APPEND([*H | *X], *Y, [*H | *Z]) :- APPEND(*X, *Y, *Z).

>
```

エディタの使って、プログラムを修正する方法については、文献[2]の pp.47-59 も参照されたい。

このプログラムを実行してみよう。

```
>?-APPEND([A,B], [C,D,E], *A).
*A = [A,B,C,D,E]
?↵

>?-APPEND([A,B,C], *A, [A,B,C,D,E]).
*A = [D,E]
?↵

>?-APPEND(*A, [D,E], [A,B,C,D,E]).
*A = [A,B,C]
?↵

>.↵                                        /* ?- モードに入る */
> ?- APPEND([A,X],*A,[A,B,C,D,E]).
NO!

> ?- APPEND([A,C,B], *A, [A,B,C,D,E]).
NO!
> ?- APPEND(*A, [X], [A,B,C,D,E]).
NO!

> ?- APPEND(*A, [E,D], [A,B,C,D,E]).
NO!

> ?- APPEND(*A, *B, [A,B,C,D,E]).
*A = []
*B = [A,B,C,D,E]
?;↵
*A = [A]
*B = [B,C,D,E]
?;↵
*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 = []
?;↵
NO!

> ?- .↵                                    /* ?- モードから出る */
>
```

これで、プログラムが正しく動作していることが確認できた（トレースをとって動作を確認するこ

とは読者にまかせる)。

このような使い方をすると、ファイルからプログラムを読み込む命令を頻繁に使うことになる。そこで、

```
>LOAD(*X) :- SEE(*X),CONSULT,CLOSE(*X).

>
```

のように定義しておいて、 :-LOAD(AP). のように利用することも考えられる。しかし、エディタと組み合わせた使い方では、 ShapeUp を呼び出してから最初に 1 回だけ LOAD を使うだけである上に、ShapeUp を新たに呼び出したときには以前に登録した節はすべて消えてしまっているので、LOAD を使うにはもう一度定義しなければならない。このようなときに便利なのが、ライブラリ・ファイルの機能である。これは、ShapeUp を起動するときに、LIBFL という名前のファイルがクイック・アクセス・ファイルとして存在するか AFT に登録されていると、自動的にこのファイルの中身を実行するという機能である。 したがって、ファイル LIBFL の中に LOAD の定義を入れておけばよい。このファイルはエディタで作成してもよいが、動作を確認してから入れた方がよいので、ShapeUp の中で作成できるようになっている。

```
>:-CLEAR.                                          /* 以下で、動作の確認を行なう */

>LOAD(*X) :- SEE(*X),CONSULT,CLOSE(*X),LIST.       /* LOAD を定義する */

>:-LIST.
LOAD(*X) :- SEE(*X),CONSULT,CLOSE(*X),LIST.

>:-LOAD(AP).                                       /* 動作を確認する */
LOAD(*X) :- SEE(*X),CONSULT,CLOSE(*X),LIST.
APPEND([], *X, *X).
APPEND([*H | *X], *Y, [*H | *Z]) :- APPEND(*X, *Y, *Z).

>:-LIBRARY(LOAD).                                  /* LOAD をライブラリに入れる */

>:-LIST.                                          /* ライブラリ内の節は表示されない */
APPEND([], *X, *X).
APPEND([*H | *X], *Y, [*H | *Z]) :- APPEND(*X, *Y, *Z).

>
```

ここで定義した LOAD には、プログラムを読み込んでから、現在のプログラムの内容を表示するように、最後に LIST を追加してある。 また、述語 LIBRARY で節を指定すると、指定された節は述語 LIST では表示されなくなる。しかし、実行に関しては定義されているのと同じである。そして、LIBRARY で指定された節は、ShapeUp の 終了時に LIBFL というテンポラリ・ファイルに書き出される。

さて、 LOAD をライブラリに追加したついでに、 現在のプログラムをファイルに書き出す SAVE もライブラリに追加しておこう。これは ShapeUp の中で簡単な修正や追加を行なったときに使う。

```
>SAVE(*X) :- TELL(*X),RESAVE,CLOSE(*X).          /* SAVE を定義する */

>:-SAVE(AP1).

>:-SYSTEM(FLIST).                /* TSS の FLIST コマンド により AP を確認する */
FILE.NAME (FC) MODE  SIZE(LLINK) CATALOG.STRING OR FILE TYPE
 SY**          RAND           2  TEMPORARY
 FACT          LINK          12  TEMPORARY
 AP            LINK           1  A60000/AP
 *SRC          LINK          12  TEMPORARY
 AP1           LINK          12  TEMPORARY

FLIST   normally terminated

>:-LIBRARY(SAVE).                          /* SAVE をライブラリに入れる */

>:-LIST.
APPEND([], *X, *X).
APPEND([*H | *X], *Y, [*H | *Z]) :- APPEND(*X, *Y, *Z).

>:-END.


*** ShapeUp version 1.12.A6S normally terminated ***
          (C&C Systems Research Lab's NEC)


APPEND([], *X, *X).

-FLIST                 /* TSS の FLIST コマンド により LIBFL を確認する */
FILE.NAME (FC) MODE  SIZE(LLINK) CATALOG.STRING OR FILE TYPE
 SY**          RAND           2  TEMPORARY
 FACT          LINK          12  TEMPORARY
 AP            LINK           1  A60000/AP
 *SRC          LINK          12  TEMPORARY
 AP1           LINK          12  TEMPORARY
 LIBFL         LINK          12  TEMPORARY

APPEND([], *X, *X).
```

ライブラリ・ファイルの確認のため、もう一度、ShapeUp を起動すると、

```
-SHP1

*** SHAPEUP V 1.12.A6S started (please do ':-versionup.' to confirm) ***
```

```
!! Restoring library functions from "LIBFL".
>
```

のようにライブラリ・ファイルを読み込んだというメッセージが表示される。ライブラリ・ファイルの中に定義してある節は、述語 LIBRARY で登録したのと同じ状態になっているので、述語 LIST では表示できない(述語 DLIBRARY を用いると通常の節に戻せる)。

差リストを用いた append のプログラムの実行については省略するので、読者自身で試みていただきたい。

2.5. その他

ここでは、プログラムの入力、デバッグ、実行のときに便利な機能をいくつか紹介しよう。

(1) 入力の拡張： Prolog では、同じ左辺をもつ節をいくつか定義するのが普通である。 ShapeUp では、このようなときに入力の手間を減らすことができるように、次のような形の入力が可能になっている。

```
>A :- B; C, D; E, F, G.
```

これは、

```
>A :- B.
>A :- C, D.
>A :- E, F, G.
```

のような３つの節を定義するのと同じである。

(2) ヒストリ機能：デバッグのときには、同じ問い合わせを何度も行ないたいことが多い。そこで、ShapeUp はユーザの問い合わせ（すなわち、:- と ?- で始まる節)に番号を付けて、最近の 10 個を覚えている。そして、

```
>:⏎
```

とすると、覚えている 10 個の問い合わせを番号と共に表示してくれ、

```
>:番号⏎
```

により、その番号の問い合わせを実行できる。

(3) 述語 HELP：ShapeUp の組み込み述語の説明のために、HELP という述語がある。

```
>:-HELP.

 * Category ( I, A, C, M, T, D, O, N, Cr) * ?
```

ここで、入力できるのは ( ) の中の文字であるが、その意味は次の通りである。

I：入出力用の組み込み述語

A：算術演算および比較演算の組み込み述語

C：実行制御用の組み込み述語

M：メタロジカルな(プログラムを変更する)組み込み述語

T：型チェックおよび型変換用の組み込み述語

D：デバッグを支援する組み込み述語

O：問い合わせに関する組み込み述語

N：組み込み述語の名前のみを表示

Cr：HELP を終了

そして、内容が多過ぎて、1 画面に入りきらないときは、

```
 * continued (";" or Cr) * ?
```

というメッセージが出る。 ここで、;⏎ を入れると、続きを表示し、⏎ のみを入れると、この項の表示を終わり、また、I から N までをきいてくる。

以上、本節では ShapeUp のプログラムを 作成・デバッグ・保守していくための方法について説明した。 本稿で説明した以外の組み込み述語や組み込み述語の詳しい説明などは、文献[1]を参照されたい。

3．おわりに

以上、センターで利用できる Prolog である ShapeUp について、 基本的な使い方、トレース機能、プログラムのファイルへの書き出しとファイルからの読み込み、ライブラリ機能について具体的な例を用いて説明した。 Prolog のような言語は、実際に使ってみるのが一番である。いろいろと試みて頂きたい。

次回は、Prolog の応用として、簡単な自然言語の処理について述べる予定である。

[参考文献]

1. ShapeUp 仕様書(V1)、58p.、日本電気株式会社 C&C 研究所 (1984)。
2. TSS の手引(昭和 60 年 3 月)、190p.、大阪大学 大型計算機センター (1985)。