【Prolog(JunPrologInterpreter)マニュアル】　　2008/06/03現在

Dec-10のPrologの記法に準拠したPrologの処理系(インタプリタ)です。すべてが前置記法で，op述語による前置／中置／後置などは定義できませんが，ユニフィケーションやバックトラックなどの機構を備えた立派なPrologの処理系です。

定数

Prologの定数には以下のようなものがあります。

記号
数
文字列
リスト
オブジェクト

記号は，小文字で始まる英数字です。もし，記号の中に大文字や特殊文字を含ませたい場合には，「'」と「'」で囲って記述します。「'」を文字列の中に書きたい場合には「''」というように重ねて書きます。

```
aoki
atsushi
symbol12345
'AOKI Atsushi'
'You are ''cool''.'
```

数は，Smalltalkの数オブジェクトと同様な書き方をします。

```
123
-123
123.456
-123.456
123.456e7
-123.456e-7
16r123
```

文字列は，「"」と「"」で囲って記述します。「"」を文字列の中に書きたい場合には「""」というように重ねて書きます。

```
"AOKI Atushi"
"You are ""cool""."
```

リストは，「[」「|」「,」「]」を使用して表わします。空リストは「[]」のように角カッコを二つ合わせて書きます。ドットペアは「[AOKI|Atsushi]」のように表わします。「AOKI」が頭部(head)であり，「Atsushi」が尾部(tail)です。「|」の代わりに「,」を用いて「[AOKI,Atsushi]」と表わすと，「AOKI」が頭部になり，「[Atsushi]」が尾部になります。「[AOKI,Atsushi]」と「[AOKI|[Atsushi]]」は同義です。また，「[]」はSmalltalkオブジェクトの「nil」と同じです。

```
[]
[AOKI|Atsushi]
[AOKI,Atsushi]
[AOKI|[Atsushi]]
```

オブジェクトは，「{」と「}」で囲ってSmalltalkのメッセージ式を記述します。「{」と「}」で囲まれた部分は，まずSmalltalk内部で実行され，その実行結果がPrologの定数として扱われます。なお，「{}」は「{nil}」と同義です。

```
{100@100}
{Image fromUser}
{('aaa') at: 2 put: $z; yourself}
```


変数

変数は，大文字で始まる英数字です。また，「~」も変数で匿名変数と呼ばれます。

```
AOKI
Atsushi
Variable1234
~
```

匿名変数は，変数が必要だけれども，その変数の値がどうでもよい場合に使います。たとえば，誰がsmalltalkを好きであるのかを調べたいのだけれど，それが誰であるかを知る必要のない場合です。

```
likes(aoki,smalltalk).
likes(aoki,prolog).
likes(watanabe,smalltalk).
likes(watanabe,lisp).
likes(sakoh,lisp).
likes(sahara,prolog).
?- likes(~,smalltalk).
yes
?- likes(~,prolog).
yes
?- likes(~,lisp).
yes
?- likes(~,c).
no
```

同様に，誰でもいいけどprologやlispを好きな人がいるかという質問「?- likes(~,prolog).」「?- likes(~,lisp).」は成功するでしょうが，一方，誰でもいいけどcが好きな人はいるかという質問「?- likes(~,c).」は失敗し，誰もC言語が好きでないことが直ちに分かります。

なぜ，匿名変数に「~(チルダ)」を使い，通常の「_(アンダーバー)」を使用しないのかというと，昔のSmalltalkの処理系は「_」を「←」とみなし，代入を表わす特殊な記号として扱っていました。そして，最新のSmalltalkは，「_」をメッセージ名の中に含めることができるようになっています。そのため，SmalltalkとPrologのインタフェースに配慮して，匿名変数を「~」にしています。

システム述語

Prologのシステム内にあらかじめ組み込まれている述語をシステム述語と呼び，それらは以下の10個のグループに大別されます。

組み込みシステム述語(systemPredicatesNo0)
基本的なシステム述語(systemPredicatesNo1)
比較演算に関するシステム述語(systemPredicatesNo2)
算術演算に関するシステム述語(systemPredicatesNo3)
節や項に関するシステム述語(systemPredicatesNo4)
高階のシステム述語(systemPredicatesNo5)
入出力に関するシステム述語(systemPredicatesNo6)
デバッグのためのシステム述語(systemPredicatesNo7)
その他のシステム述語(systemPredicatesNo8)
ユーザ定義のシステム述語(systemPredicatesNo9)

上記の10個のグループに大別されて組み込まれているシステム述語をそれぞれ順番に説明していきます。

組み込みシステム述語(systemPredicatesNo0)

!
　カット(述語!)は常に成功する述語です。しかし，大きな副作用があります。目標を充足させるにあたって，後戻り(バックトラック)が生じるような場合，その後戻りのための選択枝を再考察の必要のないものとして捨ててしまいます。そのため，解に無関係な再充足目標をあらかじめ除去しますので，実行の効率を上げることができます。また，後戻りのために使用されている記憶領域を解放しますから，メモリの節約にもなります。しかし，Prologの後戻りを故意に無効にする述語ですから，適切に使用することが肝要です。
　カットを適切に使用するケースは三つあると思われます。その第一は，目標に対する正しい方向が発見されたことをPrologに伝える場合です。第二は，ある目標を直ちに失敗させたい場合です。第三は，後戻りによる別の解の生成を停止したい場合です。

true
　常に成功する述語です。実際には必要のない述語ですが，便宜上存在しています。

fail
　常に失敗する述語です。故意に後戻り(バックトラック)を起こすことができるので，使用する場面が多かろうと思います。たとえば，append(X,Y,[1,2,3])の目標に対するすべての解を求めたい場合には，次のようになります。

```
output(X,Y) :- write(X), write(", "), write(Y), nl.
?- append(X,Y,[1,2,3]), output(X,Y), fail.
```

　このプログラムは，充足したXとYをoutput(X,Y)で出力させ，その後のfailよって後戻りを起こさせ，XとYに充足可能なすべての解を求めています。
　また，カット(!)との組み合わせも有効です。「…, !, fail.」は「実行がここまで進んだらもうよい(もう結構)」ということを表わしています。たとえば，ある目標Gが充足したことを否定する述語not(G)は，カット(!)と失敗(fail)を組み合わせて，以下のように記述することができます。

```
not(G) :- call(G), !, fail.
not(G).
```

　第一の節を実行している間に目標Gが失敗すれば，「…, !, fail.」に届く前に後戻りが起こり，第二の節の「not(G).」が充足され，目標Gが成功したことになります。反対に，第一の節を実行している間に目標Gが成功すれば，それまでの選択枝が!によって除去され，その後のfailによって第一の節が失敗し，結局のところ目標Gが失敗したことになります。

var(X)
　Xが代入されていない変数の時に成功します。

```
?- is(X,100), var(X).
no
?- is(X,Y), var(X).
X = X3 ,
Y = X3
yes
```

　ある変数が代入されていない時に，ユーザに入力をうながすには，「…, var(X), read(X), …」というイディオムが便利です。

send(X,Y,Z)

send(X,Y,Z,A)
　Smalltalkのメッセージを送信する述語です。Xをレシーバにして，Yというメッセージを，Zという引数リストを付けて送信し，その結果をAにユニファイします。レシーバに「self」の記号を指定すると，Prologのインタプリタのインスタンス(a PrologInterpreter)がレシーバになります。

```
image({Image fromUser}).
point({50@50}).
point({100@100}).
point({150@150}).
gc({ScheduledControllers activeController view graphicsContext}).
?- image(Image),
   point(Point),
   gc(GraphicsContext),
   send(Image,displayOn:at:,[GraphicsContext,Point],Image).
```

基本的なシステム述語(systemPredicatesNo1)

repeat
　繰り返しを行なう述語です。
nonvar(X)
　Xが未代入の変数であれば失敗し，代入済みの変数もしくは定数ならば成功します。

integer(X)
　Xが整数ならば成功し，それ以外ならば失敗します。

float(X)
　Xが単精度浮動小数点数ならば成功し，それ以外ならば失敗します。

double(X)
　Xが倍精度浮動小数点数ならば成功し，それ以外ならば失敗します。

fraction(X)
　Xが分数ならば成功し，それ以外ならば失敗します。

number(X) :- send(self,number:,[X]).
　Xが数ならば成功し，それ以外ならば失敗します。

symbol(X) :- send(self,symbol:,[X]).
　Xが記号ならば成功し，それ以外ならば失敗します。

string(X) :- send(self,string:,[X]).
　Xが文字列ならば成功し，それ以外ならば失敗します。

list(X) :- send(self,list:,[X]).
　Xがリストならば成功し，それ以外ならば失敗します。空リストは成功します。

dotp(X)
　Xがセル(ドットペア)ならば成功し，それ以外ならば失敗します。空リストは失敗します。

atom(X)
　Xがアトムならば成功し，それ以外ならば失敗します。アトムとは，記号，文字列，空リストのことです。

atomic(X)
  Xがアトムまたは数ならば成功し，それ以外ならば失敗します。

structure(X)
  Xがアトムまたは数ならば失敗し，それ以外ならば成功します。

比較演算に関するシステム述語(systemPredicatesNo2)

==(X,Y) :- send(X,=,[Y]).
  SmalltalkでXとYが等しいとき成功し，それ以外ならば失敗します。

¥==(X,Y) :- ==(X,Y), !, fail.
  SmalltalkでXとYが等しいとき失敗し，それ以外ならば成功します。

=(X,X)
  XとYが等しいとき成功し，それ以外ならば失敗します。

¥=(X,Y)
  XとYが等しいとき失敗し，それ以外ならば成功します。

>(X,Y)
  XがYより大きいならば成功し，それ以外ならば失敗します。

>=(X,Y)
  XがYより大きいか等しいならば成功し，それ以外ならば失敗します。

<(X,Y)
  XがYより小さいならば成功し，それ以外ならば失敗します。

=<(X,Y)
  XがYより小さいか等しいならば成功し，それ以外ならば失敗します。

算術演算に関するシステム述語(systemPredicatesNo3)

+(X,Y,Z)
  XとYを加算したものがZと等しいならば成功し，それ以外ならば失敗します。

-(X,Y,Z)
  XからYを減算したものがZと等しいならば成功し，それ以外ならば失敗します。

*(X,Y,Z)
  XとYを乗算したものがZと等しいならば成功し，それ以外ならば失敗します。

//(X,Y,Z)
  XをYで除算して余りを切り捨てたものがZと等しいならば成功し，それ以外ならば失敗します。

/(X,Y,Z)
  XをYで除算したものがZと等しいならば成功し，それ以外ならば失敗します。

¥¥(X,Y,Z)
  XをでYをモジュロ演算したものがZと等しいならば成功し，それ以外ならば失敗します。

is(X,Y)
  XとYが等しいとき成功し，それ以外ならば失敗します。

is(Z,F(X,Y))
  加減乗除において，XとYを加減乗除したものがZと等しいならば成功し，それ以外ならば失敗します。

節や項に関するシステム述語(systemPredicatesNo4)

listing
  登録されているすべての述語の節を出力し，常に成功します。

listing(X)
  Xで指定される記号を述語としてもつ節を出力し，成功します。対応する述語がないならば失敗します。

systemListing
  登録されているすべてのシステム述語の節を出力し，常に成功します。

systemListing(X)
  Xで指定される記号をシステム述語としてもつ節を出力し，成功します。対応するシステム述語がないならば失敗します。

editing
  登録されているすべての述語を編集するエディタを開き，常に成功します。

consult
  Prologのプログラムを読み込むためのウィンドウが開き成功します。このウィンドウ内でPrologのプログラムを書き，acceptすると，書かれた節を最後に追加します。

consult(X)
  Xで指定されるファイルからPrologのプログラムを読み込み，読み込んだ節を最後に追加し，成功します。対応するファイルが存在しないならば失敗します。

reconsult
  Prologのプログラムを読み込むためのウィンドウが開き成功します。このウィンドウ内でPrologのプログラムを書き，acceptすると，書かれた節をこれと同じ述語の節に置き換えます。

reconsult(X)
  Xで指定されるファイルからPrologのプログラムを読み込み，読み込んだ節をこれと同じ述語の節に置き換え，成功します。対応するファイルが存在しないならば失敗します。

saving
  登録されているすべての述語の節を入力ダイアログで指定されたファイルへ出力し，常に成功します。

saving(X)
  Xで指定される記号を述語としてもつ節を入力ダイアログで指定されたファイルへ出力し，成功します。対応する述語がないならば失敗します。

userPredicates(X)
  登録されているすべての述語をリストにしてXとユニファイします。

systemPredicates(X)
  登録されているすべてのシステム述語をリストにしてXとユニファイします。

predicates([X|Y])
  登録されているすべての述語をリストにしてXとユニファイし，登録されているすべてのシステム述語をリストにしてYとユニファイします。

functor(T,F,A)
  この述語は，Tが関数子FとN個の引数をもつ構造を意味します。

```
?- functor(likes(aoki,smalltalk),F,N).
F = likes ,
N = 2
yes
?- functor(aoki,F,N).
F = aoki ,
N = 0
yes
```

arg(N,S,T)
  この述語は，構造SのN番目の引数がTであることを意味します。常にNとSが代入されていなければなりません。

```
?- arg(2,likes(aoki,smalltalk),A).
A = smalltalk
yes
```

=..(X,L)
  この述語は「ユニブ」と呼ばれ，LはXの関数子と引数からなるリストを意味します。

```
?- =..(likes(aoki,smalltalk),L).
L = [likes,aoki,smalltalk]
yes
?- =..(X,[likes,aoki,smalltalk]).
X = likes(aoki,smalltalk)
yes
```

name(A,L)
  記号Aの文字列はリストLであることを意味します。

```
?- name(apple,L).
L = "apple"
yes
?- name(A,[97,112,112,108,101]).
A = apple
yes
?- name(A,"apple").
A = apple
yes
```

remove
  登録されているすべての述語の節を削除します。この述語は常に成功します。

remove(X)
  Xで指定される記号を述語としてもつ節を削除し，成功します。対応

する述語がないならば失敗します。

clause(X)
  Xで指定される節を登録されている節にユニファイします。

```
concat([],X,X).
concat([A|X],Y,[A|Z]) :-
        concat(X,Y,Z).
?- clause([[concat|Arguments]|Body]).
Arguments = ([],X6,X6) ,
Body = [] ;
Arguments = ([A11|X11],Y11,[A11|Z11]) ,
Body = concat(X11,Y11,Z11) ;
no
?- clause([concat(|Arguments)|Body]).
Arguments = ([],X6,X6) ,
Body = [] ;
Arguments = ([A11|X11],Y11,[A11|Z11]) ,
Body = concat(X11,Y11,Z11) ;
no
```

asserta(X)
  Xで指定される節をその述語で示される節の先頭に加えます。

```
?- remove,
   asserta([likes(aoki,smalltalk)]),
   asserta([likes(watanabe,smalltalk)]),
   listing.
likes(watanabe,smalltalk).
likes(aoki,smalltalk).
yes
```

assert(X)
assertz(X)
  Xで指定される節をその述語で示される節の末尾に加えます。

```
?- remove,
   assertz([on(cat,dog)]),
   asserta([on(hen,cat)]),
   assertz([on(dog,donkey)]),
   assertz([above(X,Y) :- on(X,Y)]),
   assertz([above(X,Y) :- on(X,Z), above(Z,Y)]),
   listing.
?- above(X,dog).
above(X,Y) :-
        on(X,Y).
above(X,Y) :-
        on(X,Z),
        above(Z,Y).
on(hen,cat).
on(cat,dog).
on(dog,donkey).
X = X1 ,
Y = Y1 ,
Z = Z1
yes
?- above(X,dog).
X = cat ;
X = hen ;
no
```

retract(X)
  Xで指定される節を登録されている節にユニファイし，成功すると，その節は削除されます。

```
        concat([],X,X).
        concat([A|X],Y,[A|Z]) :-
                concat(X,Y,Z).
        ?- listing(concat).  % ちゃんと登録されているかな
        concat([],X,X).
        concat([A|X],Y,[A|Z]) :-
                concat(X,Y,Z).
        yes
        ?- retract([concat(|Arguments)|Body]).  % 消すぞ
        Arguments = ([],X6,X6) ,
        Body = [] ;
        Arguments = ([A11|X11],Y11,[A11|Z11]) ,
        Body = concat(X11,Y11,Z11) ;
        no
        ?- listing(concat).  % もうなくなっているはず
        no
```

高階のシステム述語(systemPredicatesNo5)

call(G)
  Gを目標として充足できるならば成功となり，充足できないならば失敗となります。

not(G)
  Gを目標として充足できるならば失敗となり，充足できないならば成功となります。

or(X,Y)
  XとYの選言を表わします。Xが成功するかまたはYが成功すれば，選言は成功となります。Xが失敗の時に，Yの充足が試みられます。この時，Yも失敗すると選言が失敗をします。

and(X,Y)
  XとYの連言を表わします。Xが成功しかつYも成功するならば，連言は成功となります。Xが成功したのに，Yが失敗すると，Xの再充足が試みられます。

入出力に関するシステム述語(systemPredicatesNo6)

read(X)
  入力ダイアログから入力をXとユニファイします。

read(X,M)
  入力ダイアログから入力をXとユニファイします。その際に，入力ダイアログのメッセージとしてMを使用します。

write(X)
  Xを出力します。常に成功する述語です。

nl
  復改を出力します。常に成功する述語です。

tab(X)

  タブをXだけ出力します。Xが代入されていないならば失敗します。

clear
  出力をきれいにします。常に成功する述語です。

デバッグのためのシステム述語(systemPredicatesNo7)

clock(X)
  現在の時刻をミリ秒単位でXにユニファイします。

verbose(X)
  Xがtrueという記号の場合に，充足にかかった時間と試みたゴールの数を出力します。Xがtrueという記号以外ならば，時間とゴール数の出力を停止します。

```
        ?- remove.
        above(X,Y) :-
                on(X,Y).
        above(X,Y) :-
                on(X,Z),
                above(Z,Y).
        on(hen,cat).
        on(cat,dog).
        on(dog,donkey).
        ?- verbose(true).
        ?- above(X,donkey).
        ?- verbose(false).
        ?- remove.
        yes
        ?- verbose(true).
        <0 milliseconds, 1 goals>
        yes
        ?- above(X,donkey).
        <0 milliseconds, 4 goals>
        X = dog ;
        <0 milliseconds, 17 goals>
        X = hen ;
        <0 milliseconds, 35 goals>
        X = cat ;
        <16 milliseconds, 56 goals>
        no
        ?- verbose(false).
        yes
```

gc
  ガベジコレクションを行ないます。この述語は常に成功します。

inspect(X)
  Xをインスペクションします。この述語は常に成功します。

spy(X)
  Xで指定される記号を述語としてもつ節のトレースを開始します。

nospy(X)
  Xで指定される記号を述語としてもつ節のトレースを終了します。

trace
  登録されているすべての節のトレースを開始します。

notrace
  登録されているすべての節のトレースを終了します。

その他のシステム述語(systemPredicatesNo8)

append(X,Y,Z)
  XとYを接続したリストがZであることを意味します。

member(X,Y)
  XがYのリストの要素であることを意味します。

reverse(X,Y)
  Xのリストのリバースがyのリストであることを意味します。

length(X,Y)
  Xのリストの長さがYであることを意味します。

nth(X,Y,Z)
  XのリストのY番目の要素がZであることを意味します。

printlist(L)
  Lのリストの要素を順番に出力します。

lispAppend(X,Y,Z)
lispReverse(X,Y)
lispMember(X,Y)
lispMember(X,Y,Z)
lispAssoc(X,Y)
lispAssoc(X,Y,Z)
lispNconc(X,Y,Z)
  これらはLispの関数を呼び出す述語で，高速ですが，後戻り(バックトラック)できません。

ユーザ定義のシステム述語(systemPredicatesNo9)

  現在，ユーザ定義のシステム述語はありませんが，皆さんで作成して，ここに，その機能をお書き下さい。

システム述語一覧

```
!   (systemPredicatesNo0)
*   (systemPredicatesNo3)
+   (systemPredicatesNo3)
-   (systemPredicatesNo3)
/   (systemPredicatesNo3)
//   (systemPredicatesNo3)
<   (systemPredicatesNo2)
=   (systemPredicatesNo2)
=..   (systemPredicatesNo4)
=<   (systemPredicatesNo2)
==   (systemPredicatesNo2)
>   (systemPredicatesNo2)
>=   (systemPredicatesNo2)
and   (systemPredicatesNo5)
append   (systemPredicatesNo8)
arg   (systemPredicatesNo4)
```



```
assert   (systemPredicatesNo4)
asserta   (systemPredicatesNo4)
assertz   (systemPredicatesNo4)
atom   (systemPredicatesNo1)
atomic   (systemPredicatesNo1)
call   (systemPredicatesNo5)
clause   (systemPredicatesNo4)
clear   (systemPredicatesNo6)
clock   (systemPredicatesNo7)
consult   (systemPredicatesNo4)
dotp   (systemPredicatesNo1)
double   (systemPredicatesNo1)
editing   (systemPredicatesNo4)
fail   (systemPredicatesNo0)
float   (systemPredicatesNo1)
fraction   (systemPredicatesNo1)
functor   (systemPredicatesNo4)
gc   (systemPredicatesNo7)
inspect   (systemPredicatesNo7)
integer   (systemPredicatesNo1)
is   (systemPredicatesNo3)
length   (systemPredicatesNo8)
lispAppend   (systemPredicatesNo8)
lispAssoc   (systemPredicatesNo8)
lispMember   (systemPredicatesNo8)
lispNconc   (systemPredicatesNo8)
lispReverse   (systemPredicatesNo8)
list   (systemPredicatesNo1)
listing   (systemPredicatesNo4)
member   (systemPredicatesNo8)
name   (systemPredicatesNo4)
nl   (systemPredicatesNo6)
nonvar   (systemPredicatesNo1)
nospy   (systemPredicatesNo7)
not   (systemPredicatesNo5)
notrace   (systemPredicatesNo7)
nth   (systemPredicatesNo8)
number   (systemPredicatesNo1)
or   (systemPredicatesNo5)
predicates   (systemPredicatesNo4)
printlist   (systemPredicatesNo8)
read   (systemPredicatesNo6)
reconsult   (systemPredicatesNo4)
remove   (systemPredicatesNo4)
repeat   (systemPredicatesNo1)
retract   (systemPredicatesNo4)
reverse   (systemPredicatesNo8)
saving   (systemPredicatesNo4)
send   (systemPredicatesNo0)
spy   (systemPredicatesNo7)
string   (systemPredicatesNo1)
structure   (systemPredicatesNo1)
symbol   (systemPredicatesNo1)
systemListing   (systemPredicatesNo4)
systemPredicates   (systemPredicatesNo4)
tab   (systemPredicatesNo6)
trace   (systemPredicatesNo7)
```

```
        true   (systemPredicatesNo0)
        userPredicates   (systemPredicatesNo4)
        var   (systemPredicatesNo1)
        verbose   (systemPredicatesNo7)
        write   (systemPredicatesNo6)
        ¥=   (systemPredicatesNo2)
        ¥==   (systemPredicatesNo2)
        ¥¥   (systemPredicatesNo3)

Copyright (C) 1985-2008 AOKI Atsushi
```