プログラミング言語論第14回
論理型プログラミング (1)

情報工学科 山本修身

論理型プログラミング

ここでは**述語 (predicate)** とは「XはYよりも大きい」とか「XはYの親である」などの文のことを表す．述語を扱うことができるプログラミング言語として，Prologが良く知られている．このような言語を**論理型または論理プログラミング言語 (logic programming language)** と呼ぶ．Prolog は Programming in Logic の意味である．論理型に分類されるプログラミング言語としては，他に GHC (Guarded Horn Clause)やKL1の並列論理型プログラミング言語がある．

ここでは，Prologの基本的なプログラミングの考え方やしくみについて説明する．

本講義で用いるProlog処理系

PrologはC言語などと異なり，単一のプログラムでアルゴリズム自体が記述されるというよりも，ルールと質問からなる．ルールはファイルに記述され，質問はプロンプトからユーザが問い合わせる形式をとることも可能である．

この講義では，GNUソフトウェアである gprolog を用いて説明する．gprologは以下のURLから手に入れることができる．

http://www.gprolog.org/

このProlog処理系にはインタプリタとコンパイラが用意されている．インタプリタを立ち上げると以下のように表示してプロンプトを出す．

```
GNU Prolog 1.4.4 (64 bits)
Compiled Dec  1 2015, 00:45:10 with gcc
By Daniel Diaz
Copyright (C) 1999-2013 Daniel Diaz
| ?-
```

prologのプログラムファイルの構造

Prologのプログラムは知識の集まりである．知識とは，単純な事実またはルールのことである．これらの知識を構成するものは述語(predicate)と呼ばれる．「明日は天気が良い」とか「久弥の父は弥太郎である」などの事実や，変項を伴うもの「Xの父はY」であるなどが考えられる．それぞれの述語は，

```
father(X, Y)
```

のように記述される．カッコ内に書かれるものは変項（変数）であり変項のない述語には括弧は付けない．変項はいつもアルファベットの大文字からスタートする．小文字からスタートするものは，アトムとよばれる．Schemeのシンボルと同じである．

プログラムの構造

Prologのプログラム（コンサルト: consult) は以下に示すような構造をもつ**Horn節 (Horn clause)** から構成される.

$$Q\ \texttt{:-}\ P_1,\ P_2,\ \ldots\ ,\ P_n.$$

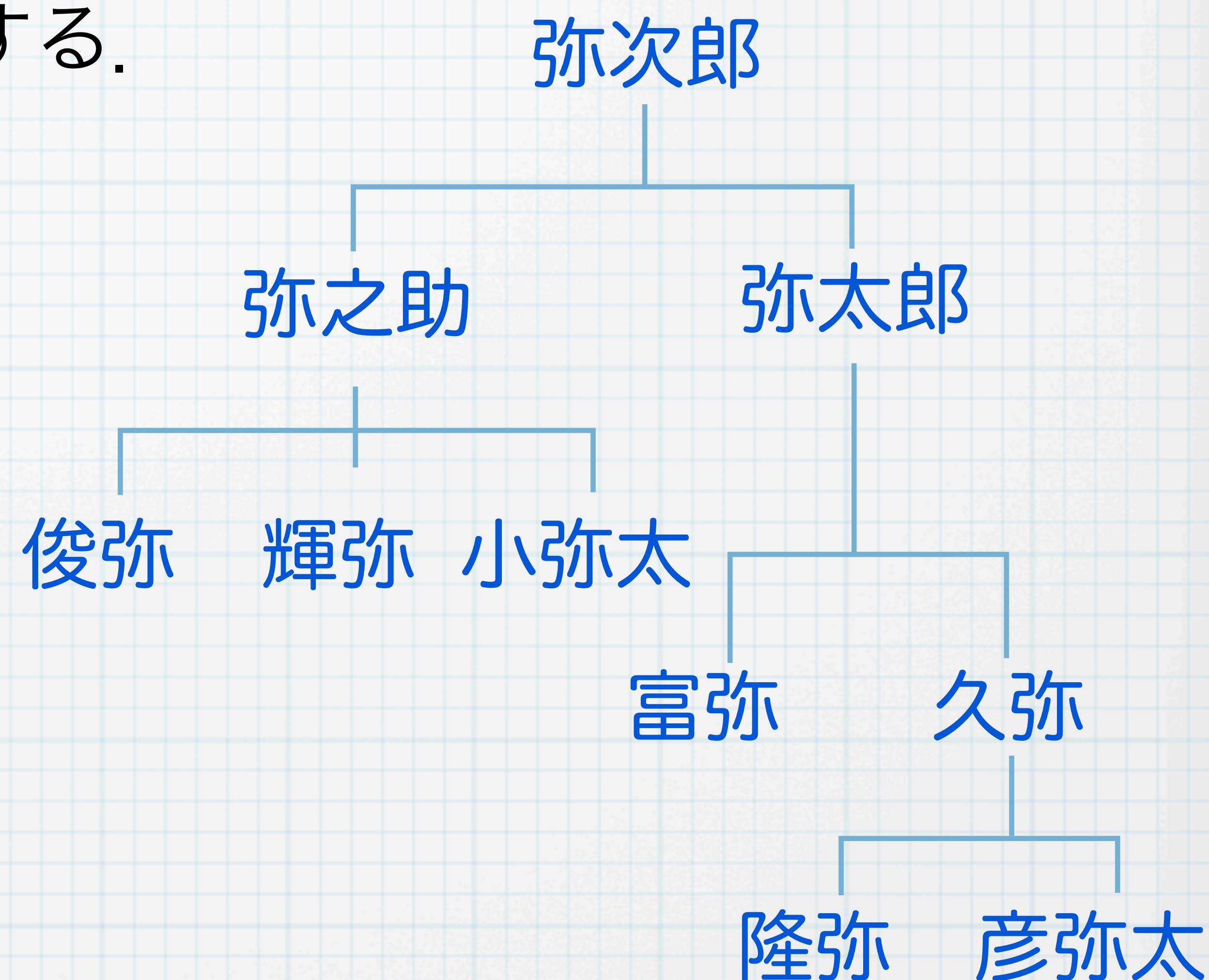
右辺の条件はカンマ（, ）で繋ぐ.

ただし，Q，P_1，$P_2, \ldots,$ P_n はそれぞれ述語である．この節が意味することは，P_1，$P_2, \ldots, P_n$がそれぞれ正しければ，Qが正しいことが証明できるということである.

ホーン節のうち，　:-　の右が存在しないものを**事実（fact)** と呼び,無条件で正しい事柄を表す.

簡単なプログラム (1)

親子関係から系図を計算するプログラムを示す．Xの親がYであることをparent(X, Y)と表現することにする．



事実

```
parent(yataro, yajiro).
parent(yanosuke, yajiro).
parent(hisaya, yataro).
parent(toshiya, yanosuke).
parent(teruya, yanosuke).
parent(koyata, yanosuke).
parent(tomiya, yataro).
parent(hikoyata, hisaya).
parent(takaya, hisaya).
```

ルール

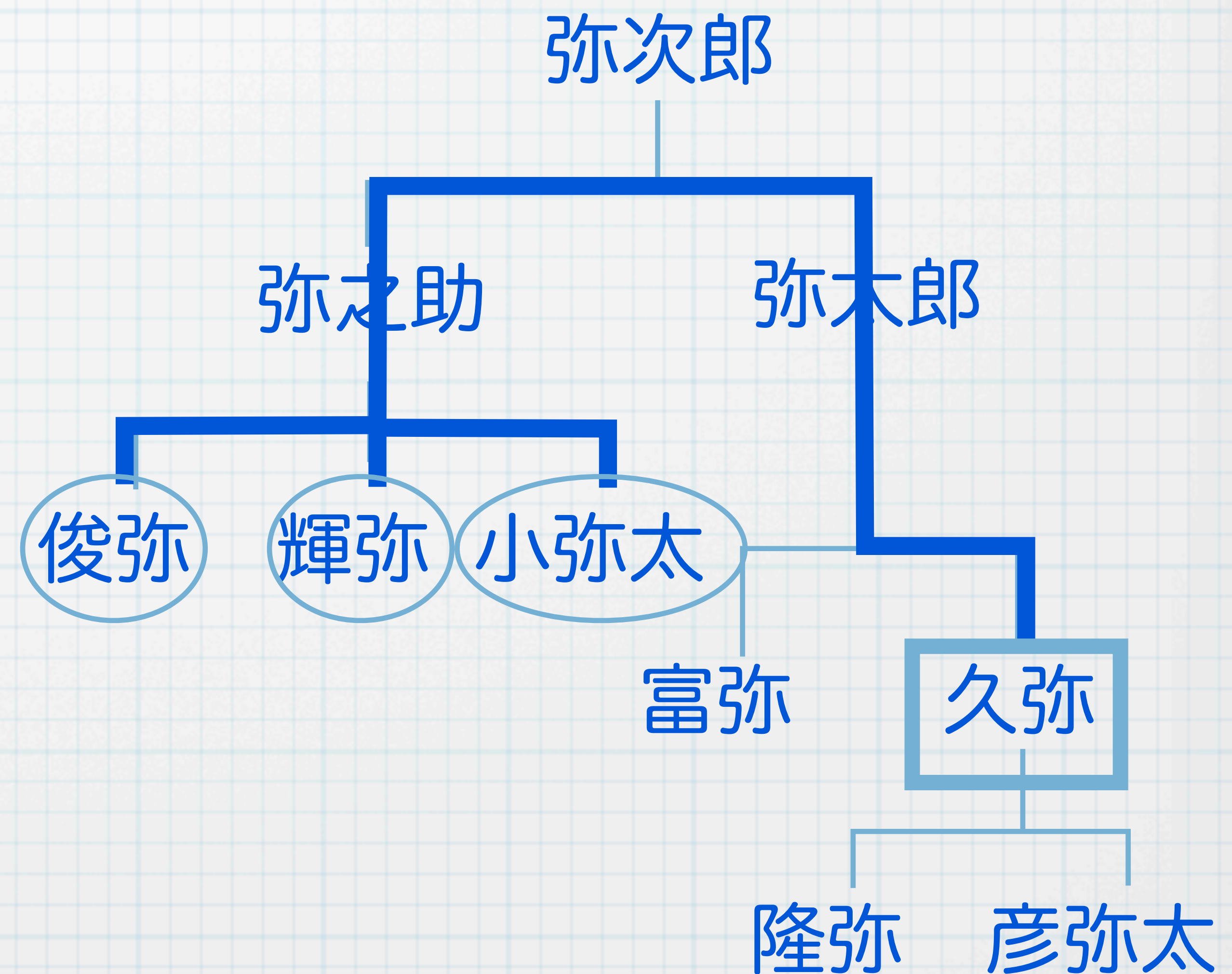
```
son(X, Y) :- parent(Y, X).
grandson(X, Y) :- son(X, Z), son(Z, Y).
brother(X, Y) :- parent(X, Z), parent(Y, Z), X \== Y.
cousin(X, Y) :- parent(X, Z1), parent(Y, Z2),
                brother(Z1, Z2), X \== Y.
```

簡単なプログラム (2)

このプログラムを実行する．まず，gprologを立ち上げて，ファイル（コンサルトファイル）を読み込む．質問に対応する述語を入力する．この述語が真になるような解を1つ見つけるとプログラムは停止する．その後，「a」を入力するとすべての可能な解を列挙する．

久弥のいとこを見つける

```
| ?- cousin(hisaya, X).

X = toshiya ? a
X = teruya
X = koyata

no
| ?-
```



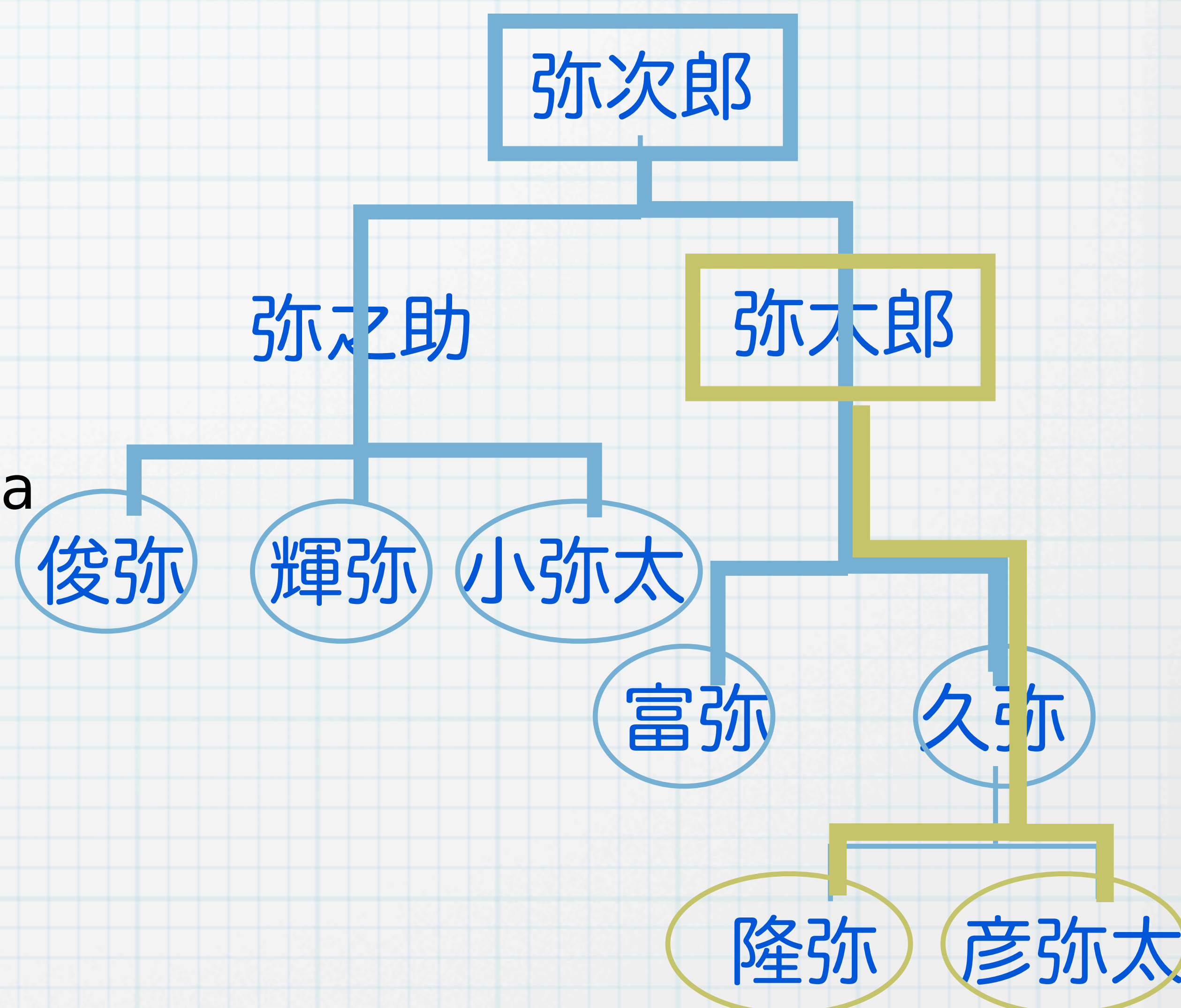
簡単なプログラム (3)

つぎに，孫とおじさんの関係をすべて列挙してみる．

すべての祖父と孫の関係

```
| ?- grandson(X, Y).

X = yajiro
Y = hisaya ? a
a

X = yajiro
Y = tomiya

X = yajiro
Y = toshiya

X = yajiro
Y = teruya
```

```
X = yajiro
Y = koyata

X = yataro
Y = hikoyata

X = yataro
Y = takaya

(1 ms) no
| ?-
```



すべての解を見つけ出してくれる．

与えられた数の和差積商を求める

2つの与えられた数の和，差，積，商を求めるプログラムを書いてみる．ここで **is** は与えられた式を評価して値を求める述語である．

```
myarith(X, Y, Sum, Diff, Prod, Quot) :-
    Sum is X + Y,
    Diff is X - Y,
    Prod is X * Y,
    Quot is X / Y.
```

```
| ?- myarith(23, 45, A, B, C, D).

A = 68
B = -22
C = 1035
D = 0.51111111111111107

yes
| ?-
```

Newton法で方程式を解いてみる

Newton法は右下のような反復公式による方程式の求解アルゴリズムである．述語 **newton** を順々に証明しようとするが，最後には値が収束してDdに関する不等式が成り立たなくなり止まる．

```
myfunc(X, Fx) :- Fx is X ** 2 - 2.
myfuncd(X, Fx) :- Fx is 2 * X.
newton(X) :-
    myfunc(X, V),
    Dd is abs(V),
    Dd > 1.0e-10,
    myfuncd(X, Vd),
    Xx is X - V / Vd,
    write(Xx), nl,
    newton(Xx).
```

$$f(x) = x^2 - 2$$

$$f'(x) = 2x$$

$$x' \leftarrow x - \frac{f(x)}{f'(x)}$$

```
| ?- newton(2.0).
1.5
1.4166666666666667
1.4142156862745099
1.4142135623746899
```

writeは与えられた引数をコンソールに出力して，真を返す述語である．また，nlは改行を出力してから真を返す述語である．

Prologにおけるリストの扱い

PrologではSchemeと同様にリストを扱うことができる．Prologのリストはスキームとほぼ同じ構造を持ち，基本的にはconsセルによって構成される.

2つの要素を繋げたconsセルは [a | b] と表現される．また，[a | [b]] のことを [a, b] と書く．[a, b, c] = [a | [b | [c | []]]]となる.

Prologのリスト	Schemeのリスト
[]	()
[a, b]	(a b)
[a | b]	(a . b)
[a, b | c]	(a b . c)

リストの長さを測る，リストの要素を取り出す

リストの長さを測るには，Schemeでやったのと同様にconsセルを一つずつはがしていく．はがせなくなったら，終了となる．同時にカウンタを引数に入れることによって，長さが測れる．

```
mylength([], 0).
mylength([_ | X], N) :-
    length(X, N1),
    N is N1 + 1.
```

```
| ?- mylength([a, b, c, d, e], N).

N = 5

(1 ms) yes
```

また，リストのn番目の要素を求めるには，以下のようなmy_nthという述語を定義すれば良い．

```
my_nth([X | _], 0, X).
my_nth([_ | Y], N, Z) :-
    N > 0,
    N1 is N - 1,
    my_nth(Y, N1, Z).
```

```
| ?- my_nth([a, b, c, d, e, f, g], 4, X).

X = e ? ;

no
| ?-
```

ここで，直接利用しないが，何らかのオブジェクト（リストやアトムを含めて）が存在することを 「_」で表現する．

与えられた要素をリストの中から探す述語

リストとある要素が与えられたとき，リスト中にその要素が出現するか否かを判定する述語を示す．Nは出現する位置を表す．ここでfailは常に失敗する述語で，これが実行されるということは見つからなかったことを表す．

```
my_search([], _, _) :- fail.
my_search([X | _], X, 0).
my_search([X | Y], Z, N) :-
    X \== Z,
    my_search(Y, Z, N1),
    N is N1 + 1.
```

```
| ?- my_search([a, b, c, d, e], d, N).

N = 3 ?

yes
| ?- my_search([a, b, c, d, e], x, N).

no
| ?-
```

fail は常に失敗して偽を返す述語である.

集合の共通部分を計算する

リストを集合と見たとき，共通部分を求める述語の定義は以下のとおり．

```
is_member(_, []) :- fail.
is_member(X, [X | _]).
is_member(X, [Y | Z]) :-
    X \== Y,
    is_member(X, Z).

not_member(_, []).
not_member(X, [X | _]) :- fail.
not_member(X, [Y | Z]) :-
    X \==Y,
    not_member(X, Z).

my_intersection([], _, []).
my_intersection([X | Y], Z, [X | M]) :-
    is_member(X, Z),
    my_intersection(Y, Z, M).
my_intersection([X | Y], Z, M) :-
    not_member(X, Z),
    my_intersection(Y, Z, M).
```

```
| ?- my_intersection([a, b, c, d], [c, d, e, f], X).

X = [c,d] ? a
a

no
```

リストを連結する述語

リストを連結するプログラムはとても簡単に書ける.

```
my_append([], X, X).
my_append([A | X], Y, [A | Z]) :- my_append(X, Y, Z).
```

```
| ?- my_append([a,b,c], [d, e, f], Z).

Z = [a,b,c,d,e,f]

yes
| ?-
```

Prologが扱っているのは述語であって関数ではないので，連結したときの結果を指定して，元のリストを答えさせることが可能である.

```
| ?- my_append(X, Y, [a, b, c]).

X = []
Y = [a,b,c] ? a
a

X = [a]
Y = [b,c]

X = [a,b]
Y = [c]

X = [a,b,c]
Y = []

no
| ?-
```

Prologはどのように計算しているか (1)

前述のmy_appendを例にして，Prologがどのように計算しているのかを調べてみる.

```
(I)   my_append([], XX, XX).
(II)  my_append([AA | XX], YY, [AA | ZZ]) :- my_append(XX, YY, ZZ).
```

Prologは問い合わせられた述語が真であることを証明しようとする. そのためにうまくマッチできるHorn節を探す. たとえば,

```
my_append(X, Y, [a, b, c]).
```

を入力すると，まず，`(I)`, `(II)`のどちらかに合わせようとする.

`(I)`を採用して `my_append([], XX, XX) = my_append(X, Y, [a, b, c])`

だとすれば,

```
X = [], XX= Y, XX = [a, b, c]
```

となり，これより， **`X = [], Y = [a, b, c], XX = [a, b, c]`**

と推論して，`(I)`のHorn節の右辺に何もないことから，これがひとつの解となる．よって，これを出力する.

Prologはどのように計算しているか (2)

つぎに (II) の可能性を調べる．この場合，

```
my_append([AA | XX], YY, [A | ZZ]) = my_append(X, Y, [a, b, c])
```

となり，

```
[AA | XX] = X, YY = Y, [AA | ZZ] = [a, b, c]
```

これより，

```
AA = a, ZZ = [b, c], X = [a| XX], YY = Y
```

となるが，この対応が正しいためには，(II)のHorn節の右辺が成り立つ必要がある．したがって，

```
my_append(XX, Y, [b, c]).
```

を証明しようとする．これが証明できれば，元の述語が正しいことが証明できる．

```
(I)  my_append([], XX, XX).
(II) my_append([AA | XX], YY, [AA | ZZ]) :- my_append(XX, YY, ZZ).
```

Prologはどのように計算しているか (3)

my_append(XX, Y, [b, c]). を証明してみる．まず，(I)にマッチする可能性を調べると，

```
my_append([], XX’, XX’) = my_append(XX, YY, [b, c])
```

これより，

```
[] = XX, XX’ = YY, XX’ = [b, c]
```

これを解いて，

XX = [], XX’ = [b, c], YY = [b, c]

(I)の右辺はないので，この対応で正しいことになる．したがって，前のスライドの対応と合わせれば，

AA = a, ZZ = [b, c], X = [a, XX], YY = Y

→ **AA = a, ZZ = [b, c], X = [a], YY = Y= [b, c]**

X = [a], Y = [b, c]を出力する.

```
(I)  my_append([], XX, XX).
(II) my_append([AA | XX], YY, [AA | ZZ]) :- my_append(XX, YY, ZZ).
```

Prologはどのように計算しているか (4)

さらに可能性として，(II)にマッチさせてみる.

```
my_append([AA’ | XX’], YY’, [AA’ | ZZ’]) = my_append(XX, YY, [b, c])
```

これより，以下が成り立たないといけない.

```
[AA’ | XX’] = XX, YY’ = YY, [AA’ | ZZ’] = [b, c]
```

これを解いて，

AA’ = b, ZZ’ = [c], XX = [b | XX’], YY’ = YY

この可能性が実際に成り立つためには，Horn節の右辺が成りたたなければならない．したがって，

```
my_append(XX’, YY’, [c]).
```

を証明する必要がある．この述語が(I)にマッチするとすれば，どうようにして，**XX’ = []**, **YY’ = [c]**となり，結局 **XX = [b]**, **YY = [c]**となり，元に戻すと，**X = [a, b], Y = [c]**となる．これを出力する

```
(I)  my_append([], XX, XX).
(II) my_append([AA | XX], YY, [AA | ZZ]) :- my_append(XX, YY, ZZ).
```

Prologはどのように計算しているか (5)

さらに計算は続くが，以降は省略する.

```
my_append([], XX, XX) = my_append(X, Y, [a, b, c])
```

を仮定したとき,

```
X = [], XX= Y, XX = [a, b, c]
```

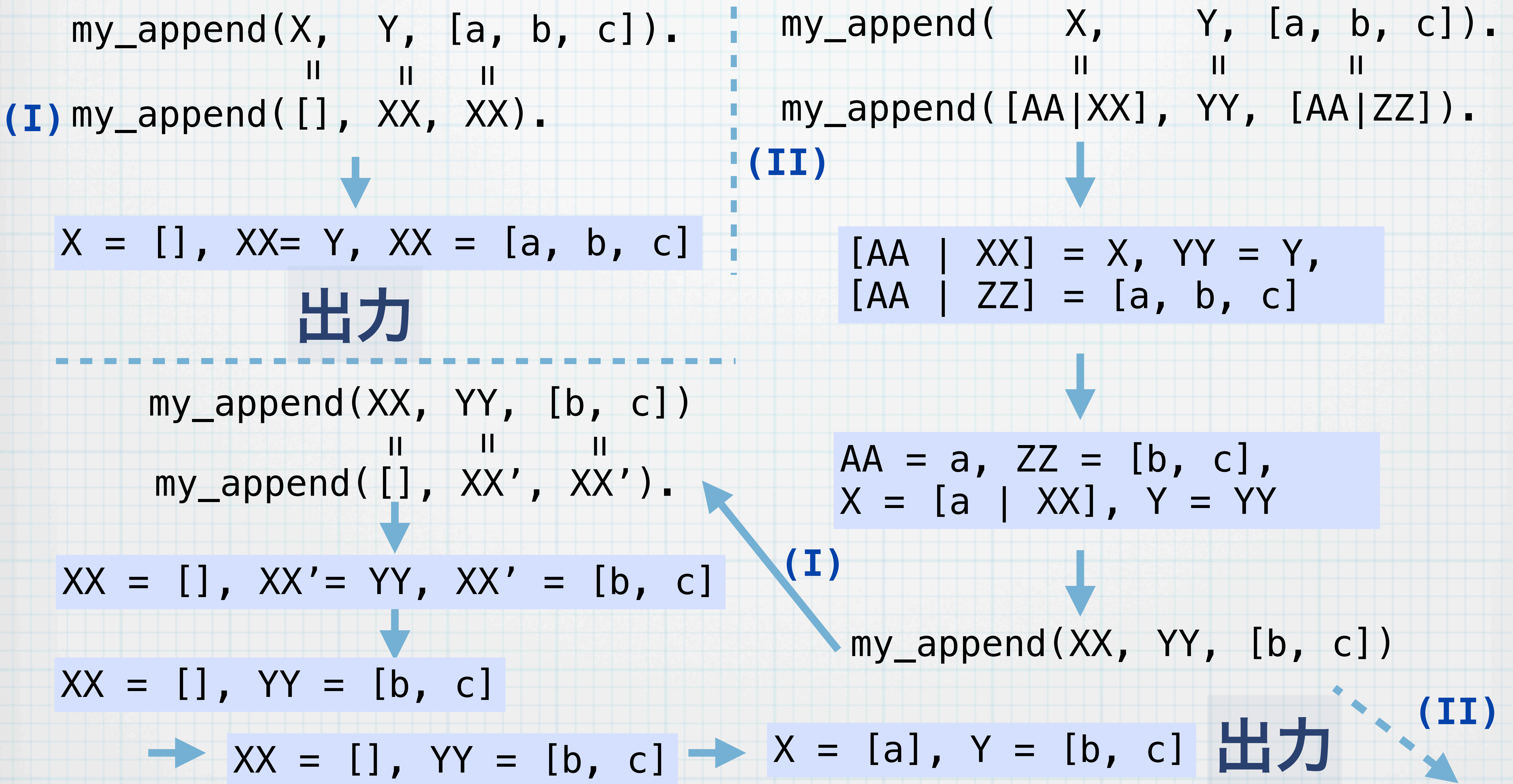
のように対応させて，さらにここから,

```
X = [], Y = [a, b, c], XX = [a, b, c]
```

を得る操作を**ユニフィケーション（unification, 単一化）**と呼ぶ．この操作がPrologの動作の基本となる．PrologはHorn節から得られるすべての可能性を試みて，その後停止する.

Prologはどのように計算しているか (6)

結局，Prologの動きをまとめると以下のようになる．




```
(I)  my_append([], XX, XX).
(II) my_append([AA | XX], YY, [AA | ZZ]) :- my_append(XX, YY, ZZ).
```

素直なリストの連結の動作については？

単純に2つのリストを結ぶ場合にはバックトラックがないので，もっと動きは単純になる．

```
      my_append([a, b], [x, y, z], X).
                    ||       ||          ||
(II) my_append([AA|XX],  YY,    [AA|ZZ]).
                    ↓
AA = a, XX = [b], YY = [x, y, z], X = [a | ZZ]
                    ↓
      my_append([b],   [x, y, z],   ZZ).
                 ||        ||           ||
(II) my_append([AA'|XX'],  YY',   [AA'|ZZ']).
                    ↓
AA' = b, XX' = [], YY' = [x, y, z], ZZ = [b | ZZ']
                    ↓
                                            (I)
      my_append([], [x, y, z], ZZ').  →  ZZ' = [x, y, z]
→ ZZ = [b, x, y, z]  →  X = [a, b, x, y, z]
```

```
(I)  my_append([], XX, XX).
(II) my_append([AA | XX], YY, [AA | ZZ]) :- my_append(XX, YY, ZZ).
```

ベキ集合の計算

ベキ集合の述語 power_set を以下のように定義する.

```
append_one(_, [], []).
append_one(A, [X | Y], [[A | X] | AY]) :- append_one(A, Y, AY).

power_set([], [[]]).
power_set([A | X], PX_APX) :-
    power_set(X, PX),
    append_one(A, PX, APX),
    append(PX, APX, PX_APX).
```

```
| ?- power_set([a, b, c], X).

X = [[],[c],[b],[b,c],[a],[a,c],[a,b],[a,b,c]] ? a

no
| ?-
```

実際にどのような計算が行われているのか, 考えてみよう

ここまでのまとめ

- Prologは与えられたHorn節の集合に基づいて，与えられたゴールが正しいことを証明するためのシステムである．ゴールの中に変数については，その変数に何を設定すれば正しくなるかを最後に返す．正しいことが証明できれば，システムは計算をやめるが，すべての解が得られるまで計算を継続させることもできる．

- Prologの計算は単一化を行いながら進む．Prologの計算の本質は単一化であると言っても良い．