プログラミング言語論第15回
論理型プログラミング (2)およびまとめ

情報工学科 山本修身

論理型プログラミング 再掲

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．

プログラムの構造 再掲

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 `:-` P_1`,` P_2`, ... ,` P_n`.`　右辺の条件はカンマ（,）で繋ぐ．

ただし，Q, P_1, P_2,..., 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,...,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．

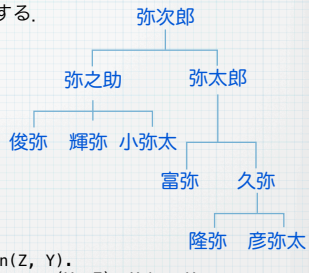
ホーン節のうち，`:-` の右が存在しないものを**事実 (fact)** と呼び，無条件で正しい事柄を表す．

簡単なプログラム (1) 再掲

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.
```

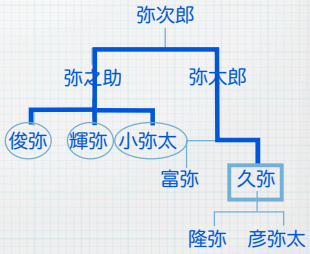
変数は大文字でアトムは小文字

簡単なプログラム (2) 再掲

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
| ?-
```

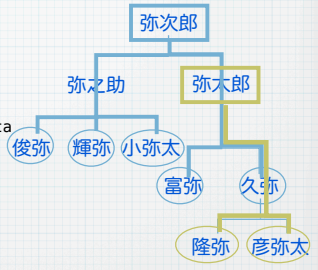


簡単なプログラム (3) 再掲

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 再掲

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
| ?-
```

集合からその要素を順に取り出す

Prologでは与えられた集合から要素を順に取り出す動作を記述する必要がある．そのとき利用されるのがselectという述語である．selectは標準で定義されているが，ここでは説明のために，my_selectを定義してみる．

```
my_select(X, [X | Xs], Xs).
my_select(X, [Y | Ys], [Y | Zs]) :-
    my_select(X, Ys, Zs).

test2 :-
    my_select(X, [a, b, c, d], Xs),
    write(X),
    write(' '),
    fail.
```

```
| ?- test2.
a b c d

no
| ?-
```

Prolog処理系は与えられた述語が正しいことを証明できるか証明不能であることが判明するまで，あらゆる可能性を繰り返す．この場合，最後に人工的にfailを入れることによってすべての可能性で失敗させて，すべての答を導出させる．

順列をつくる

順列を作り出す述語は簡単に定義することができる．

```
perm([], []).
perm(Xs, [Z | Zs]) :- select(Z, Xs, Ys), perm(Ys, Zs).
```

実際，これを用いて4つの要素の順列を生成すると以下のようになる．

```
test :-
    perm([a, b, c, d], X),
    write(X),
    nl,
    fail.
```

```
| ?- test.   [c,a,b,d]
[a,b,c,d]    [c,a,d,b]
[a,b,d,c]    [c,b,a,d]
[a,c,b,d]    [c,b,d,a]
[a,c,d,b]    [c,d,a,b]
[a,d,b,c]    [c,d,b,a]
[a,d,c,b]    [d,a,b,c]
[b,a,c,d]    [d,a,c,b]
[b,a,d,c]    [d,b,a,c]
[b,c,a,d]    [d,b,c,a]
[b,c,d,a]    [d,c,a,b]
[b,d,a,c]    [d,c,b,a]
[b,d,c,a]
```

組み合わせを列挙する

組み合わせはselectを用いなくても簡単に作ることができる．ある集合の要素の組み合わせは，その集合の要素 X が組み合わせに**含まれるか含まれないか**のどちらかである．それぞれの可能性について述語のルールを定義してやれば良い．

```
comb(_, 0, []).
comb([X | Xs], N, [X | Ys]) :-
    N > 0,
    N1 is N - 1,
    comb(Xs, N1, Ys).
comb([_ | Xs], N, Ys) :-
    N > 0,
    comb(Xs, N, Ys).
```

←Xが組み合わせに含まれる場合

←Xが組み合わせに含まれない場合

```
job :-
    comb([1, 2, 3, 4, 5, 6], 3, X),
    write(X),
    nl,
    fail.
```

```
| ?- job.
[1,2,3]
[1,2,4]
[1,2,5]
[1,2,6]
[1,3,4]
[1,3,5]
[1,3,6]
[1,4,5]
[1,4,6]
[1,5,6]
[2,3,4]
[2,3,5]
[2,3,6]
[2,4,5]
[2,4,6]
[2,5,6]
[3,4,5]
[3,4,6]
[3,5,6]
[4,5,6]

no
```

３次の魔方陣を求める

先ほど定義した述語permを用いて３次の魔方陣をすべて列挙する．sumx(X, Idx, N)は配列YのIdxの指す要素の和がNとなる述語である．

```
perm([], []).
perm(Xs, [N | Ys]) :- select(N, Xs, Rest), perm(Rest, Ys).
sumx(_, [], 0).
sumx(Y, [I| Idx], N) :- nth(I, Y, V), sumx(Y, Idx, N1), N is V + N1.

magic3(X) :- perm([1, 2, 3, 4, 5, 6, 7, 8, 9], X),
             sumx(X, [1, 2, 3], 15),
             sumx(X, [4, 5, 6], 15),
             sumx(X, [7, 8, 9], 15),
             sumx(X, [1, 4, 7], 15),
             sumx(X, [2, 5, 8], 15),
             sumx(X, [3, 6, 9], 15),
             sumx(X, [1, 5, 9], 15),
             sumx(X, [3, 5, 7], 15).
```

2	7	6
9	5	1
4	3	8

この場合，すべての順列を列挙してから，それが魔方陣になっているか調べている．

```
job :- magic3(X),
       write(X),
       nl,
       fail.
```

```
[2,7,6,9,5,1,4,3,8]
[2,9,4,7,5,3,6,1,8]
[4,3,8,9,5,1,2,7,6]
[4,9,2,3,5,7,8,1,6]
[6,1,8,7,5,3,2,9,4]
[6,7,2,1,5,9,8,3,4]
[8,1,6,3,5,7,4,9,2]
[8,3,4,1,5,9,6,7,2]
```

繰り返しを表現する

繰り返しを表現するために，述語betweenを定義する．

```
mybetween(X, Y, X) :-  X =< Y.
mybetween(X, Y, Z) :-  X < Y, X1 is X + 1, mybetween(X1, Y, Z).
```

betweenはシステムであらかじめ定義されているので，ここではmybetweenとして定義した．上記のように定義すると，Cにおける繰り返しのような作業が可能となる．

```
| ?- mybetween(1, 10, _), write('a'), fail.
aaaaaaaaaa

no
| ?- mybetween(1, 10, I), write(I), write(' '), fail.
1 2 3 4 5 6 7 8 9 10

no
```

九九の表を書いてみる

betweenを使えば，ループによる繰り返しを表現することができる．このしくみを用いて九九の表を書いてみる．

```
write_if_less_than_ten(K) :- K >= 10.
write_if_less_than_ten(K) :- K < 10, write(' ').

kuku :-
    between(1, 9, I),
    nl,
    between(1, 9, J),
    K is I * J,
    write('  '),
    write_if_less_than_ten(K),
    write(K),
    fail.
kuku :- nl.
```

```
| ?- kuku.

   1   2   3   4   5   6   7   8   9
   2   4   6   8  10  12  14  16  18
   3   6   9  12  15  18  21  24  27
   4   8  12  16  20  24  28  32  36
   5  10  15  20  25  30  35  40  45
   6  12  18  24  30  36  42  48  54
   7  14  21  28  35  42  49  56  63
   8  16  24  32  40  48  56  64  72
   9  18  27  36  45  54  63  72  81

yes
| ?-
```

カットについて

これまで見てきたようにPrologでは，与えられた述語が正しいということを証明するために，与えられたホーン節をトラックしながら計算を進める．可能性がなくなるか，証明ができれば停止する．その際自動的にバックトラックが行われる．

カット (!) はそれ自体常に成功する述語であるが，副作用として現在調べている述語が呼ばれてからカットが呼ばれる以前までの部分でバックトラックを止めさせる述語である．

```
a :- between(1, 3, X), between(1, 3, Y),
     write(X), write(Y), nl, fail.
```

```
| ?- a.
11
12
13
21
22
23
31
32
33
```

```
a :- between(1, 3, X), !, between(1, 3, Y),
     write(X), write(Y), nl, fail.
```

```
| ?- a.
11
12
13
```

カットを用いた否定

カットを用いることによって，否定を実現することができる．すなわち，ある述語が失敗したとき成功し，成功したとき失敗するような述語を作ることができる．

```
not(X) :- X, !, fail.
not(_).
```

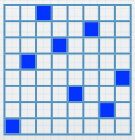
not(X)を証明するためにまず，Xが正しいことを証明することを試みる．Xが正しいと！へ進みその後failへ進み失敗する．本来はバックトラックして他の解を探しにいくが，!を通過しているのでバックトラックせず，失敗して終わる．一方，Xが正しくないと，!の前で失敗するので，バックトラックして，2行目の定義にしたがって，成功する．以上より，Xの論理値の否定が結果となる．

```
| ?- not(a == b).

yes
| ?- not(a == a).

no
```

８クイーンパズルを計算する

前述の否定があると簡単に8クイーンパズルを解くプログラムを作ることができる．attack(X, N, Xs)はXsの先頭の要素とXがNずれているという述語である．

```
attack(X, N, [M | _]) :- M is X + N.
attack(X, N, [M | _]) :- M is X - N.
attack(X, N, [_ | Rest]) :- N1 is N + 1, attack(X, N1, Rest).
attack(_, _, []) :- fail.

eightqueen([], []).
eightqueen([N | Xs], Y) :- select(N, Y, Ys),
                           eightqueen(Xs, Ys),
                           not(attack(N, 1, Xs)).

job :- eightqueen(X, [1, 2, 3, 4, 5, 6, 7, 8]),
       write(X), nl, fail.
```



```
[1,5,8,6,3,7,2,4]
[1,6,8,3,7,4,2,5]
[1,7,4,6,8,2,5,3]
[1,7,5,8,2,4,6,3]
.......
[7,5,3,1,6,8,2,4]
[8,2,4,1,7,5,3,6]
[8,2,5,3,1,7,4,6]
[8,3,1,6,2,5,7,4]
[8,4,1,3,6,2,7,5]
```

92個の解がすべて表示される．

フィボナッチ数列の計算を効率化する

フィボナッチ数列を計算するプログラムは以下のように書くことで，コンパイラで直接実行形式を生成することができる．この場合F(30)を出力する．このとき，カットがないとメモリが溢れてうまく動かない

```
fib(0, 1).
fib(1, 1).
fib(N, X) :- N > 1,
             N1 is N - 1, fib(N1, X1), !,
             N2 is N - 2, fib(N2, X2), !,
             X is X1 + X2.

q :- fib(30, X), write(X), nl, halt.

:- initialization(q).
```

$$F(0) = F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

```
OMacBook:yama564> gplc fib.pl
OMacBook:yama565> ./fib

Fatal Error: local stack
overflow (size: 16384 Kb,
reached: 16384 Kb, environme\
nt variable used: LOCALSZ)
OMacBook:yama566>
```

カットがない場合

```
OMacBook:yama567> ./fib
1346269
OMacBook:yama568>
```

カットがある場合

期末試験に向けて (1)

半年間の講義を振り返る

回	表題	
1	変数と関数の定義と評価のモデル	関数や変数の定義
2	変数と関数の定義と評価のモデル	プログラムの構造
3	手続きとそれが生成するプロセス	手続きと変数，引数
4	手続きとそれが生成するプロセス	
5	データによる抽象化（1）	consセル，リスト
6	例題による解説（1）	8クイーンパズルなど
7	データによる抽象化（2）	集合の表現，スタック，キュー
8	中間テストと解説	
9	高階手続きによる抽象化	データとしてのlambda式
10	局所状態変数と環境モデルー クロージャによる抽象ー	環境モデル
11	遅延評価とストリーム	遅延評価
12	例題による解説（2）	evalとapplyの構造，call/cc
13	マクロ	マクロ
14	論理型プログラミング（1）	Prologの初歩
15	論理型プログラミング（2）とまとめ	cut, Prologの応用

プログラミング言語の基礎

プログラミング言語の仕組み

期末試験に向けて (2)

チェックポイント

- **Schemeにおける関数や変数の使い方が理解出来ている**
- Schemeにおける末尾再帰が理解できて，通常の再帰と区別できる．また末尾再帰のプログラムが書ける．
- S式とconsセルの関係を理解しており，リストによるデータの管理ができる．
- リストによるキューやスタックなどの実現について理解している．
- lambda式の意味を理解して，高階手続きを使うことができる．
- 環境モデルが理解でき，それぞれの状況で適切に利用できる
- 遅延評価について理解して，プログラムが書ける．
- Schemeの評価子の仕組みの概略を理解している．
- 継続の処理について理解している．
- Schemeにおけるマクロの意味を理解して，簡単なマクロが書ける．
- Prologの基本的な仕組みについて理解していて，簡単なプログラムが書ける．

まとめ

プログラミング言語は色々な試行錯誤の結果，現在のような状況となっている．これからもプログラミング言語は変化していくと考えられ，一定の規則や様式を覚えているだけでは対応することが難しくなる可能性がある．

しかし，プログラミング言語の設計思想やその基礎となる原理はそれほど変わるものではないので，そこを理解すればある程度追従していくことができると考えられる．

この講義ではSchemeをベースにしていくつかのプログラミング言語のしくみについて解説した．自分の身近なプログラミング言語でこのようなしくみがどのように実現されているのか良く考えることが必要である．