Part 1 The Prolog Language

Chapter 5
Controlling Backtracking

5.1 Preventing backtracking

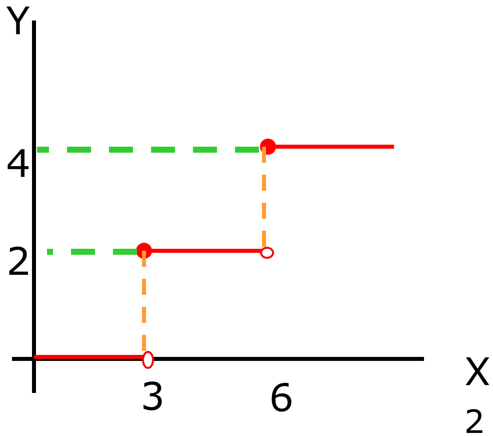
- Prolog will automatically backtrack if this is necessary for satisfying a goal.
  - However, uncontrolled backtracking may cause inefficiency in a program.
  - Therefore, we sometimes want to control, or to prevent, backtracking.
  - We can do this in Prolog by using the 'cut' facility.

- Consider the double-step function shown in Figure 5.1.
  - The relation between X and Y can be specified by three rules:

  Rule 1: if x < 3 then Y = 0.

  Rule 2: if 3<= X and X < 6 then Y = 2.

  Rule 3: if 6<= X then Y = 4.



5.1 Preventing backtracking

- This can be written in Prolog as a binary relation:

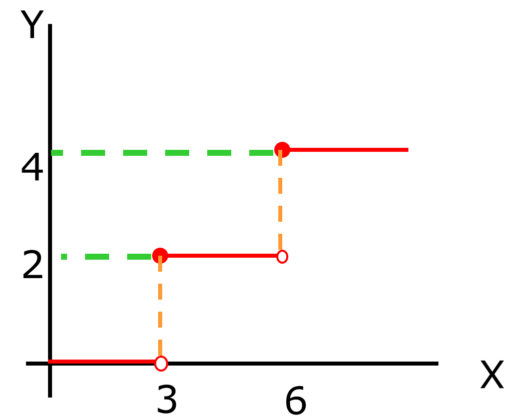
  f(X, Y)

  as follows:

  (Rule1) f(X, 0) :- X < 3.

  (Rule2) f(X, 2) :- 3=<X, X<6.

  (Rule3) f(X, 4) :- 6 =< X.



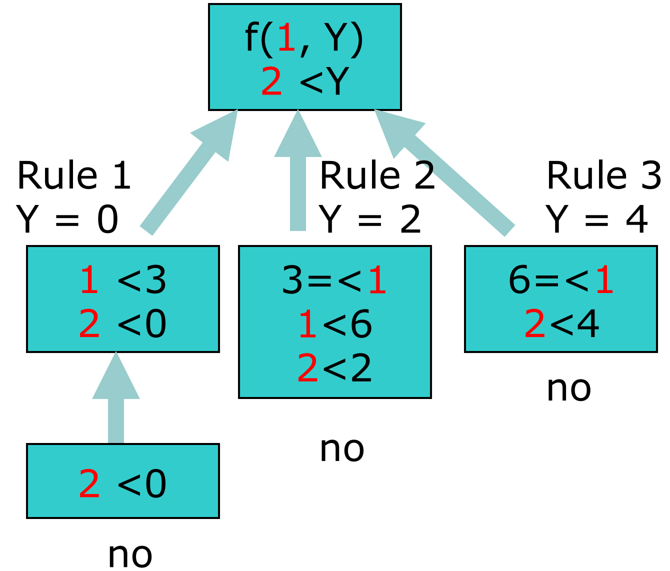
- Experiment 1

  ?- f(1, Y), 2 < Y.

  First goal: f(1, Y). ➔ Y = 0.

  Second goal: 2<0. ➔ fail

  …



5.1 Preventing backtracking

- Experiment 1
  - The three rules about the **f** relation are mutually (互相) exclusive (排外) so that one of them at most will succeed.
  - Therefore we know that as soon as one rule succeeds there is no point in trying to use the others.
  - In the example of Figure 5.2, rule 1 has become known to succeed at the point indicated by 'cut'. At this point we have to tell Prolog explicitly not to backtrack.

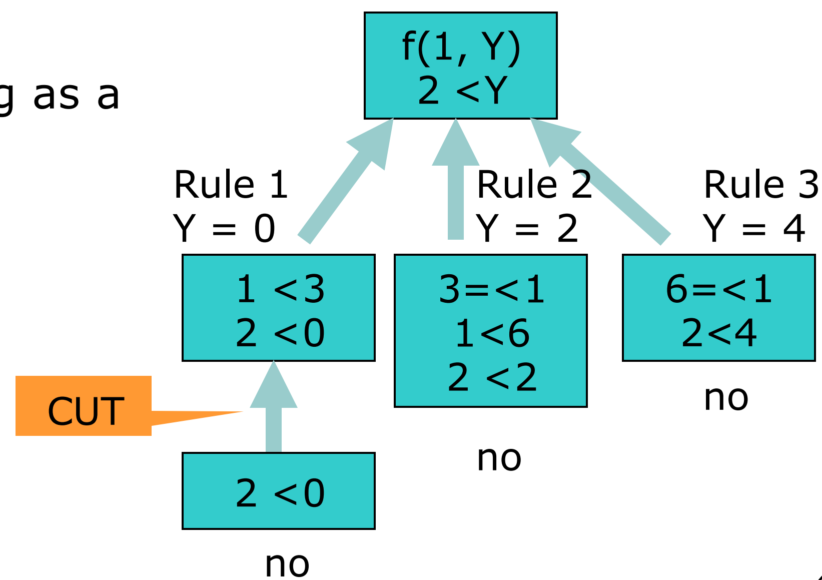
- This can be written in Prolog as a binary relation:

  f(X, Y)

  as follows:

  CUT

```
f( X, 0) :- X < 3, !.
f( X, 2) :- 3=<X, X<6, !.
f( X, 4) :- 6 =< X.
```



5.1 Preventing backtracking

```
f( X, 0) :- X < 3.
f( X, 2) :- 3=<X, X<6.
f( X, 4) :- 6 =< X.

| ?- f(1, Y), 2<Y.
      1    1  Call: f(1,_16) ?
      2    2  Call: 1<3 ?
      2    2  Exit: 1<3 ?
      1    1  Exit: f(1,0) ?
      3    1  Call: 2<0 ?
      3    1  Fail: 2<0 ?
      1    1  Redo: f(1,0) ?
      2    2  Call: 3=<1 ?
      2    2  Fail: 3=<1 ?
      2    2  Call: 6=<1 ?
      2    2  Fail: 6=<1 ?
      1    1  Fail: f(1,_16) ?

 no
{trace}
```

```
f( X, 0) :- X < 3, !.
f( X, 2) :- 3=<X, X<6, !.
f( X, 4) :- 6 =< X.

| ?- f(1, Y), 2<Y.
      1    1  Call: f(1,_16) ?
      2    2  Call: 1<3 ?
      2    2  Exit: 1<3 ?
      1    1  Exit: f(1,0) ?
      3    1  Call: 2<0 ?
      3    1  Fail: 2<0 ?

no
{trace}
```

- Now, we have improved the efficiency of this program by adding cuts.

5.1 Preventing backtracking

- Experiment 2

  | ?- f(7, Y).

  Y = 4

  yes

  (Rule1) f(X, 0) :- X < 3, !.
  (Rule2) f(X, 2) :- 3=<X, X<6, !.
  (Rule3) f(X, 4) :- 6 =< X.

  - This produced the following sequence of goals:
    - Try rule 1: 7<3 fails, backtrack and try rule 2 (cut was not reached).
    - Try rule 2: 3=<7 succeeds, but then 7<6 fails, backtrack and try rules 3 (cut was not reached)
    - Try rule 3: 6=<7 succeeds.
  - If the test (X<3) in rule 1 is fail, then the test (3=< X) in rule 2 should be true. Therefore the second test is redundant (多餘的) and the corresponding goal can be omitted.
  - This leads to the more economical formulation of the three rules:

    If X < 3 then Y = 0,

    otherwise if X < 6 then Y = 2,

    otherwise Y = 4.

5.1 Preventing backtracking

- Experiment 2

  If X < 3 then Y = 0,

  otherwise if X < 6 then Y = 2,

  otherwise Y = 4.

  Then the third version of the program:

```
f( X, 0) :- X < 3, !.
f( X, 2) :- X < 6, !.
f( X, 4).
```

```
f( X, 0) :- X < 3, !.
f( X, 2) :- 3=<X, X<6, !.
f( X, 4) :- 6 =< X.
```

- Here we can not remove the cuts.
  - If the cuts be removed, it may produce multiple solutions, and some of which are not correct.
  - For example:

```
?- f(1,Y).
Y = 0;
Y = 2;
Y = 4;
no
```

```
f( X, 0) :- X < 3.
f( X, 2) :- X < 6.
f( X, 4).
```

5.1 Preventing backtracking

- CUT:

  H :- B1, B2, …, Bm, !, …, Bn.

  - Assume that this clause was invoked by a goal G that match **H**. Then G is the parent goal.
  - At the moment that the cut is encountered, the system has already found some solution of the goals **B1, …, Bm.**
  - When the cut is executed, this (current) solution of **B1, …, Bm** becomes frozen and all possible remaining alternatives(選擇) are discarded (忽略).
  - Also, the goal G now becomes committed (堅定的) to this clause: any attempt to match G with the head of some other clause is precluded (阻止).

5.1 Preventing backtracking

- For example:

  C :- P, Q, R, !, S, T, U.

  C :- V.

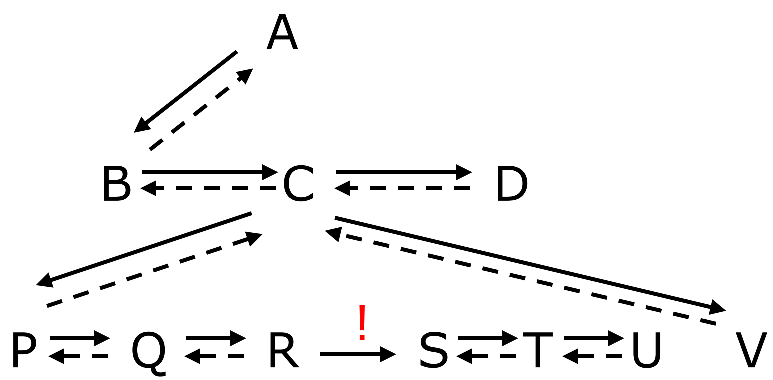
  A:- B, C, D.

  ?- A.

  - Here backtracking will be possible within the goal list **P**, **Q**, **R**.
  - As soon as the cut is reached, all alternative solutions of the goal list **P**, **Q**, **R** are suppressed (抑制).
  - The alternative clause about **C**,

    C:- V.

    will also be discarded.
  - However, backtracking will still be possible within the goal list **S**, **T**, **U**.
  - The cut will only affect the execution of the goal **C**.



The solid arrows indicate the sequence of calls, the dashed arrows indicate backtracking. There is 'one way traffic' between R and S.

5.2 Examples using cut
5.2.1 Computing maximum

- The procedure for finding the larger of two numbers can be programmed as a relation: **max(X, Y, Max)**

```
max( X, Y, X) :- X >= Y.
max( X, Y, Y) :- X < Y.
```

  - These two rules are mutually (互相) exclusive (排外).
  - Therefore a more economical formulation is possible:

```
max( X, Y, X) :- X >= Y, !.
max( X, Y, Y).
```

  - It should be noted that the use of this procedure requires care. For example:

    ?- max(3, 1, 1). ➔ the answer is yes.

  - The following reformulation of max overcomes this limitation:

```
max( X, Y, Max) :- X >= Y, !, Max = X
                   ;
                   Max = Y.
```

5.2.2 Single-Solution membership

- Use the relation **member(X, L)** to establish whether X is in list L.

  member(X, [X|L]).

  member(X, [Y|L]) :- member(X, L).

  - This is non-deterministic: if X occurs several times then any occurrence (事件) can be found.
  - Let us change member into a deterministic procedure which will find only the first occurrence.

    member(X, [X|L]) :- !.

    member(X, [Y|L]) :- member(X, L).

  - This program will generate just one solution. For example:

    ?- member(X, [a, b, c, a]).

    X = a;

    no

5.2.3 Adding an element to a list without duplication

- Let **add** relation can add an item **X** to the list **L** only if **X** is not yet in **L**. If **X** is already in **L** then **L** remains the same.

  add(X, L, L1)

  where **X** is the item to be added

  L is the list to which **X** is to be added

  L1 is the resulting new list

- Our rule for adding can be formulated as:

  If **X** is a member of list **L** then **L1** = **L**,

  Otherwise **L1** is equal **L** with **X** inserted.

- This is then programmed as follows:

```
add( X, L, L) :- member( X, L), !.
add( X, L, [X|L]).
```

5.2.3 Adding an element to a list without duplication

- For example:

```
| ?- add( a, [b, c], L).
L = [a, b, c]
yes

| ?- add( X, [b, c], L).
L = [b, c]
X = b
yes

| ?- add( a, [b, c, X], L).
L = [b, c, a]
X = a
Yes

| ?- add( a, [a, b, c], L).
L = [a, b, c]
yes
```

5.2.4 Classification into categories

- Assume we have a database of results of tennis games played by members of a club.

  beat(tom, jim)
  beat(ann, tom)
  beat(pat, jim)

  Tom win Jim

- Now we want to define a relation

  class(Player, Category)

  that ranks the players into categories.
- We have just three categories:
  - winner: every player who won all his or her games is a winner
  - fighter: any player that won some games and lost some
  - sportsman: any player who lost all his or her games.
- For example:
  - Ann and pat are winners.
  - Tom is a fighter.
  - Jim is a sportsman.

5.2.4 Classification into categories

- The rule for a fighter:

  X is a fighter if

  there is some Y such that X beat Y **and**

  there is some Z such that Z beat X.

- The rule for a winner:

  X is a winner if

  X beat some Y **and**

  X was not beaten by anybody

  - This formulation contains 'not' which cannot be directly expressed with our present Prolog facilities.

- Thus we need an alternative formulation:

  If X beat somebody and X was beaten by somebody

  then X is a fighter,

  otherwise if X beat somebody

  then X is a winner,

  otherwise if X got beaten by somebody

  then X is a sportsman.

5.2.4 Classification into categories

If X beat somebody and X was beaten by somebody
  then X is a fighter,
  otherwise if X beat somebody
    then X is a winner,
    otherwise if X got beaten by somebody
      then X is a sportsman.

```
class( X, fighter) :- beat( X, _), beat(_, X), !.
class( X, winner) :- beat( X, _), !.
class( X, sportsman) :- beat(_, X).
```

```
| ?- class( tom, C).
C = fighter
yes              % as intended (預期的)

| ?- class( tom, sportsman).
true ?
yes              % not as intended (非預期的)
```

```
beat( tom, jim)
beat( ann, tom)
beat( pat, jim)
```

Exercise

- Exercise 5.1
  - Let a program be:

```
p(1).
p(2) :- !.
p(3).
```

Write all Prolog's answers to the following questions:

(a) ?- p(X).

(b) ?- p(X), p(Y).

(c) ?- p(X), !, p(Y).

- Exercise 5.3
  - Define the procedure

split(Numbers, Positives, Negatives)

which splits a list of numbers into two lists: positive ones (including zero) and negative ones.

For example:

split([3,-1,0,5,-1], [3,0,5], [-1,-2])

5.3 Negation as failure

- 'Mary like all animals but snakes'. How can we say this in Prolog?

  If X is a snake then 'Mary likes X' is not true,

  otherwise if X is an animal then Mary likes X.

  - That something is not true can be said in Prolog by using a special goal, **fail**, which always fails, thus forcing the parent goal to fail.

    likes(mary, X):- snake(X), !, fail.

    likes(mary, X) :- animal(X).

  - If X is a snake then the cut will prevent backtracking (thus excluding the second rule) and fail will cause the failure.
  - These two clauses can be written more compactly as one clause:

```
likes( mary, X):- snake( X), !, fail
                  ;
                  animal( X).
```

5.3 Negation as failure

- Define the **difference** relation with the same idea.

  If X and Y match then **difference(X, Y)** fails,
  otherwise **difference(X, Y)** succeeds.

```
difference( X, X):- !, fail.
difference( X, Y).
```

```
difference( X, Y):- X=Y, !, fail
                           ;
                           true.
```

- **True** is a goal that always succeeds.

5.3 Negation as failure

- These examples indicate that it would be useful to have a unary predicate ‘**not**’ such that

  not(Goal)

  is true if **Goal** is not true.

  - If Goal succeeds then **not(Goal)** fails, otherwise **not(Goal)** succeeds.

  -
```
not( P):- P, !, fail
          ;
          true.
```

  - Now, we assume that **‘not’** is a built-in Prolog procedure that behaves as defined here.

5.3 Negation as failure

- If we define

  :- op(900, fy, not).

  then

  we can write the goal

  not(snake(X))

  as

  not snake(X).

- Applications:

  (1) likes(mary, X) :- animal(X), not snake(X).

  (2) difference(X, Y) :- not(X = Y).

  (3) class(X, fighter) :- beat(X, _), beat(_, X).

  class(X, winner) :- beat(X, _), not beat(_, X).

  class(X, sportsman) :- beat(_, X), not beat(X, _).

5.3 Negation as failure

- Applications:

(4) the eight queens program (Compare with Figure 4.7)

```
solution( [] ).
solution( [X/Y | Others] )  :-
        solution( Others),
        member( Y, [1,2,3,4,5,6,7,8] ),
        not attacks( X/Y, Others).

attacks( X/Y, Others)  :-
        member(X1/Y1, Others),
       (Y1 = Y;
        Y1 is Y + X1 – X;
        Y1 is Y – X1 + X).
```

4.5.1 The eight queens problem—Program 1

```
%  Figure 4.7  Program 1 for the eight queens problem.

solution( [] ).
solution( [X/Y | Others] )  :-
  solution( Others), member( Y, [1,2,3,4,5,6,7,8] ),
  noattack( X/Y, Others).

noattack( _, [] ).
noattack( X/Y, [X1/Y1 | Others] )  :-
  Y =\= Y1,  Y1-Y =\= X1-X, Y1-Y =\= X-X1, noattack( X/Y, Others).

member( Item, [Item | Rest] ).
member( Item, [First | Rest] )  :-  member( Item, Rest).

% A solution template

template( [1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8] ).
```

Exercise

- Exercise 5.5
  - Define the set subtraction relation

    set_difference(Set1, Set2, SetDifference)

    where all the three set are represented as lists.

    For example:

    set_difference([a,b,c,d],[b,d,e,f], [a,c])

5.4 Problems with cut and negation

- The advantages of 'cut':
  - With cut we can often improve the efficiency of the program.
  - Using cut we can specify mutually exclusive rules. So we can express rules of the form:

    if condition P then conclusion Q,
    otherwise conclusion R

    In this way, cut enhances the expressive power of the language.

5.4 Problems with cut and negation

- Disadvantages:
  - In the programs with cuts, a change in the order of clauses may affect the declarative meaning. This means that we can get different results.
  - For example:

p :- a, b.

p :- c.

The declarative meaning of this program is:

p <==> (a & b) V c

Let us now insert a cut:

p :- a, !, b.

p :- c.

The declarative meaning is:

p <==> (a & b) V (~a & c)

5.4 Problems with cut and negation

- If we swap the clauses:

  p :- c.

  p :- a, !, b.

  The declarative meaning of this program is:

  p <==> c V (a & b)

- The important point is that when we use the cut facility we have to pay more attention to the procedural aspects.

- This additional difficulty increases the probability of a programming error.

5.4 Problems with cut and negation

- Cuts:
  - Green cuts: the cuts had no effect on the declarative meaning
  - Red cuts: the cuts that do affect the declarative meaning.
  - Red cuts are the ones that make programs hard to understand, and they should be used with special care.
  - Cut is often used in combination with a special goal, **fail**.
  - For reasons of clarity we will prefer to use **not** instead of the *cut-fail* combination, because the negation is clearer than the *cut-fail* combination.

5.4 Problems with cut and negation

- The problems of **not**:
  - If we ask Prolog:

    ?**- not human(mary).**

    Prolog will probably answer 'yes'.

  - What Prolog means is:
    - There are not enough information in the program to prove that Mary is human.

  - When we do not explicitly enter the clause

    human(mary).

    into our program, we do not mean to imply that Mary is not human.

5.4 Problems with cut and negation

- Another example:

```
good_standard( wangsteak).
good_standard( tasty).
expensive(wangsteak).
reasonable( Restaurant) :-
        not expensive( Restaurant).
```

  - If we ask
    ?- good_standard(X), reasonable(X).
    Prolog will answer:
    X = tasty.

  - If we ask apparently the same question:
    ?- reasonable(X), good_standard(X).
    then Prolog will answer:
    no.

5.4 Problems with cut and negation

```
| ?- good_standard( X), reasonable( X).
      1    1  Call: good_standard(_16) ?
      1    1  Exit: good_standard(wangsteak) ?
      2    1  Call: reasonable(wangsteak) ?
      3    2  Call: not expensive(wangsteak) ?
      4    3  Call: '$call'(expensive(wangsteak),not,1,true) ?
      5    4  Call: expensive(wangsteak) ?
      5    4  Exit: expensive(wangsteak) ?
      4    3  Exit: '$call'(expensive(wangsteak),not,1,true) ?
      6    3  Call: fail ?
      6    3  Fail: fail ?
      3    2  Fail: not expensive(wangsteak) ?
      2    1  Fail: reasonable(wangsteak) ?
      1    1  Redo: good_standard(wangsteak) ?
      1    1  Exit: good_standard(tasty) ?
      2    1  Call: reasonable(tasty) ?
      3    2  Call: not expensive(tasty) ?
      4    3  Call: '$call'(expensive(tasty),not,1,true) ?
      5    4  Call: expensive(tasty) ?
      5    4  Fail: expensive(tasty) ?
      4    3  Fail: '$call'(expensive(tasty),not,1,true) ?
      3    2  Exit: not expensive(tasty) ?
      2    1  Exit: reasonable(tasty) ?

X = tasty

(31 ms) yes
```

```
good_standard(wangsteak).
good_standard( tasty).
expensive(wangsteak).
reasonable( Restaurant) :-
    not expensive( Restaurant).
```

5.4 Problems with cut and negation

```
{trace}
| ?- resonable( X), good_standard(X).
      1    1  Call: resonable(_16) ?
      2    2  Call: not expensive(_16) ?
      3    3  Call: '$call'(expensive(_16),not,1,true) ?
      4    4  Call: expensive(_16) ?
      4    4  Exit: expensive(wangsteak) ?
      3    3  Exit: '$call'(expensive(wangsteak),not,1,true) ?
      5    3  Call: fail ?
      5    3  Fail: fail ?
      2    2  Fail: not expensive(_16) ?
      1    1  Fail: resonable(_16) ?

(16 ms) no
{trace}
```

```
good_standard(wangsteak).
good_standard( tasty).
expensive(wangsteak).
reasonable( Restaurant) :-
    not expensive( Restaurant).
```

5.4 Problems with cut and negation

- Discuss:
  - The key difference between both questions is that the variable X is, in the first case, always instantiated when **reasonable(X)** is executed, whereas X is not yet instantiated in the second case.
  - The general hint is: **not Goal** works safely if the variable in **Goal** are instantiated at the time **not Goal** is called. Otherwise we may get unexpected results due to reasons explained in the sequel.

5.4 Problems with cut and negation

```
good_standard( wangsteak).
good_standard( tasty).
expensive(wangsteak).
reasonable( Restaurant) :-
        not expensive( Restaurant).
```

```
| ?- expensive( X).
X = wangsteak
yes

| ?- not expensive( X).
no
```

- The answer is not X = tasty.