Part 1 The Prolog Language

Chapter 7
More Built-in Predicates

7.1 Testing the type of terms
7.1.1 Predicates var, nonvar, atom,…

- Term may be of different types: variable, integer, atom, etc.
  - If a term is variable, then it can be instantiated (舉例說明) or uninstantiated.
  - If it is instantiated, its value can be an atom, a structure, etc.
  - It is sometimes useful to know what the type of this value is.
    - For example, we may add the values of two variables, X and Y, by:

      Z is X + Y
    - Before this goal is executed, X and Y have to be instantiated to numbers.
    - If we are not sure that X and Y will indeed be instantiated to numbers at this point then we should check this in the program before arithmetic is done.

7.1 Testing the type of terms
7.1.1 Predicates var, nonvar, atom,…

- **number(X):**
  - **number** is a built-in predicate.
  - **number(X)** is true if X is a number or if it is a variable whose value is a number.
  - The goal of adding X and Y can then be protected by the following test on X and Y:

    …, number(X), number(Y), Z is X + Y,…
  - If X and Y are not both numbers then no arithmetic will be attempted.

7.1 Testing the type of terms

7.1.1 Predicates var, nonvar, atom,…

- Built-in predicates of this sorts are:

var(X)	succeeds if X is currently an uninstantiated variable
nonvar(X)	succeeds if X is not a variable, or X is an already instantiated variable
atom(X)	is true if X currently stands for an atom
integer(X)	is true if X currently stands for an integer
float(X)	is true if X currently stands for a real number
number(X)	is true if X currently stands for a number
atomic(X)	is true if X currently stands for a number or an atom
compound(X)	is true if X currently stands for a compound term (a structure)

7.1 Testing the type of terms
7.1.1 Predicates var, nonvar, atom,…

```
| ?- var(Z), Z = 2.
Z = 2
yes
| ?- Z = 2, var(Z).
no


| ?- integer( Z), Z = 2.
no
| ?- Z = 2, integer( Z),
    nonvar( Z).
Z = 2
yes
```

```
| ?- atom( 3.14).
no
| ?- atomic( 3.14).
yes
| ?- atom( ==>).
yes
| ?- atom( p(1)).
no

| ?- compound( 2 + X).
yes
```

7.1 Testing the type of terms
7.1.1 Predicates var, nonvar, atom,…

- We would like to count how many times a given atom occurs in a given list of objects.
- To this purpose we will define a procedure:

  count(A, L, N)

  where **A** is the atom, **L** is the list and **N** is the number of occurrences.

```
count( _, [], 0).
count( A, [A|L], N) :- !, count( A, L, N1), N is N1 +1.
count( A, [_|L], N) :- count( A, L, N).
```

7.1 Testing the type of terms
7.1.1 Predicates var, nonvar, atom,…

```
| ?- count( a, [a, b, a, a], N).
N = 3 ?
yes

| ?- count( a, [a, b, X, Y], Na).
Na = 3
X = a
Y = a ?
yes

| ?- count( b, [a, b, X, Y], Nb).
Nb = 3
X = b
Y = b ?
yes
```

```
| ?- L=[a, b, X, Y],
     count( a, L, Na),
     count( b, L, Nb).
L = [a, b, a, a]
Na = 3
Nb = 1
X = a
Y = a ?
yes
```

7.1 Testing the type of terms
7.1.1 Predicates var, nonvar, atom,…

- We are interested in the number of the real occurrences of the given atom, and not in the number of terms that match this atom.

- The modified program is as follows:

```
count1( _, [], 0).
count1( A, [B|L], N) :- atom( B), A = B, !,
                        count1( A, L, N1), N is N1 +1
                        ;
                        count1( A, L, N).
```

7.1 Testing the type of terms
7.1.1 Predicates var, nonvar, atom,…

```
| ?- count1( b, [a, b, X, Y], Nb).
Nb = 1 ? ;
no

| ?- count1( a, [a, b, X, Y], Na).
Na = 1 ?
yes

| ?- L=[a,b,X,Y], count1( a, L, Na), count1( b, L, Nb).
L = [a,b,X,Y]
Na = 1
Nb = 1 ? ;
no
```

7.1.2 A cryptarithmetic puzzle using nonvar

- A popular example of a cryptarithmetic(密碼算術) puzzle is

```
   D O N A L D
 + G E R A L D
 -------------
   R O B E R T
```

- The problem here is to assign decimal digits to the letters D, O, N, etc., so that the above sum is valid.
- All letters have to be assigned different digits, otherwise trivial solutions are possible–for example, all letters equal zero.

7.1.2 A cryptarithmetic puzzle using nonvar

- Define a relation

  sum(N1, N2, N)

  where N1, N2 and N represent the three numbers of a given cryptarithmetic puzzle.
  - The goal **sum(N1, N2, N)** is true if there is an assignment of digits to letters such that N1 + N2 = N.
- Each number can be represented as a list of decimal digits.
  - For example, the number 225 would be represented by the list [2, 2, 5].
  - Using this representation, the problem can be depicted as:

    [D, O, N, A, L, D] + [G, E, R, A, L, D] = [R, O, B, E, R, T]
  - Then the puzzle can be stated to Prolog by the question:

    ?- sum([D,O,N,A,L,D], [G,E,R,A,L,D], [R,O,B,E,R,T]).

7.1.2 A cryptarithmetic puzzle using nonvar

Number1 = [D_{11}, D_{12}, ..., D_{1i},...]
Number2 = [D_{21}, D_{22}, ..., D_{2i},...]
Number3 = [D_{31}, D_{32}, ..., D_{3i},...]

Here carry must be 0

Carry from the right

Here carry is 0

0 C C_1 0

Number1	......................D_{1i}................
+ Number2	......................D_{2i}................
= Number3	......................D_{3i}................

The relations at the indicated ith digit position are:

D_{3i} = (C_1+D_{1i}+D_{2i}) mod 10;

C = (C_1+D_{1i}+D_{2i}) div 10.

7.1.2 A cryptarithmetic puzzle using nonvar

- Define a more general relation

  sum1(N1, N2, N, C1, C, Digits1, Digits)

  where **N1, N2** and **N** are our three numbers,

  C1 is carry from the right, and

  C is carry to the left (after the summation).

  Digits1 is the list of available digits for instantiating the variables in N1, N2, and N.

  Digits is the list of digits that were not used in the instantiation of these variables.

  - For example:

    ?- **sum1([H, E], [6, E], [U,S], 1, 1, [1,3,4,7,8,9], Digits).**

    H = 8

    E = 3

    S = 7

    U = 4

    Digits = [1, 9]

1 ←			←1
	H	E	
	6	E	
	U	S	

1 ←			←1
	8	3	
	6	3	
	4	7	

7.1.2 A cryptarithmetic puzzle using nonvar

- The definition of sum in terms of sum1 is as follows:

```
sum( N1, N2, N) :-
   sum1(N1, N2, N, 0, 0, [0,1,2,3,4,5,6,7,8,9],_).
```

  - This relation is general enough that it can be defined recursively.
  - We assume the threes lists representing the three numbers are of equal length.
  - Our example problem satisfies this constraint; if not, the 'shorter' number can be prefixed by zeros.

7.1.2 A cryptarithmetic puzzle using nonvar

- The definition of **sum1** can be divided into two cases:

  (1) The three numbers are represented by empty lists. Then:

  sum1([], [], [], C, C, Digs, Digs).

  (2) All three numbers have some left-most digit and the remaining digits on their right. So they are of the form:

  [D1 | N1], [D2 | N2], [D |N]

  In this case two conditions must be satisfied:

  (a) The three numbers N1, N2 and N have to satisfy the **sum1** relation, giving some carry digit, C2, to the left, and leaving some unused subset of decimal digits, **Digs2**.

7.1.2 A cryptarithmetic puzzle using nonvar

(b) The left-most digits D1, D2, and D, and the carry digit C2 have to satisfy the relation indicated in Figure 7.1: C2, D1, and D2 are added giving D and a carry to the left. This condition will be formulated in our program as a relation **digitsum.**

- Translating this case into Prolog we have:

```
sum1( [D1|N1], [D2|N2], [D|N], C1, C,
Digs1, Digs)  :-
    sum1( N1, N2, N, C1, C2, Digs1, Digs2),
    digitsum( D1, D2, C2, D, C, Digs2, Digs).
```

7.1.2 A cryptarithmetic puzzle using nonvar

- The definition of relation **digitsum**

  digitsum(D1, D2, C2, D, C, Digs2, Digs).

  - D1, D2 and D have to be decimal digits.
  - If any of them is not yet instantiated then it has to become instantiated to one of the digits in the list **Digs2**.
  - The digit has to be deleted from the set of **Digs2**.
  - If D1, D2 or D is already instantiated then none of avaiable digits will be spent.
  - This is realized in the program as a non-deterministic deletion of an item from a list.
  - If this item is non-variable then nothing is deleted.

```
del_var( A, L, L) :-  nonvar(A), !.
del_var( A, [A|L], L).
del_var( A, [B|L], [B|L1])  :- del_var(A, L, L1).
```

7.1.2 A cryptarithmetic puzzle using nonvar

```
% Figure 7.2 A program for cryptoarithmetic puzzles.
sum(N1, N2, N) :-
   sum1( N1, N2, N, 0, 0, [0,1,2,3,4,5,6,7,8,9], _).
sum1( [], [], [], C, C, Digits, Digits).
sum1( [D1|N1], [D2|N2], [D|N], C1, C, Digs1, Digs) :-
        sum1( N1, N2, N, C1, C2, Digs1, Digs2),
        digitsum( D1, D2, C2, D, C, Digs2, Digs).
digitsum( D1, D2, C1, D, C, Digs1, Digs) :-
       del_var( D1, Digs1, Digs2), del_var( D2, Digs2, Digs3),
       del_var( D, Digs3, Digs), S is D1 + D2 + C1,
       D is S mod 10, C is S // 10.
del_var( A, L, L) :- nonvar(A), !.
del_var( A, [A|L], L).
del_var( A, [B|L], [B|L1]) :- del_var(A, L, L1).

% Some puzzles
puzzle1( [D,O,N,A,L,D], [G,E,R,A,L,D], [R,O,B,E,R,T] ).
puzzle2( [0,S,E,N,D], [0,M,O,R,E], [M,O,N,E,Y] ).
```

7.1.2 A cryptarithmetic puzzle using nonvar

```
| ?- puzzle1(N1, N2, N), sum( N1, N2, N).

N  = [7,2,3,9,7,0]
N1 = [5,2,6,4,8,5]
N2 = [1,9,7,4,8,5] ? ;
(15 ms) no
```

```
  D O N A L D
+ G E R A L D
-------------
  R O B E R T
```

```
| ?- puzzle2(N1, N2, N), sum( N1, N2, N).

N  = [0,8,3,5,6]
N1 = [0,7,5,3,1]
N2 = [0,0,8,2,5] ? ;

N  = [0,6,3,7,8]
N1 = [0,5,7,3,1]
N2 = [0,0,6,4,7] ? ...
```

```
  0 S E N D
+ 0 M O R E
-----------
  M O N E Y
```

7.1.2 A cryptarithmetic puzzle using nonvar

```
| ?- sum1([5,O, N, A, L, 5],
          [G, E, R, A, L, 5],
          [R, O, B, E, R, T],
          0, 0, [0,1,2,3,4,5,6,7,8,9], _).
```

A = 4	A = 4	A = 4	A = 4
B = 2	B = 2	B = 3	B = 3
E = 9	E = 9	E = 9	E = 9
G = 1	G = 1	G = 1	G = 1
L = 8	L = 8	L = 8	L = 8
N = 5	N = 5	N = 6	N = 6
O = 3	O = 6	O = 2	O = 5
R = 7	R = 7	R = 7	R = 7
T = 0 ? ;	T = 0 ? ;	T = 0 ? ;	T = 0 ? ;
			(47 ms) no

Exercise (Homework)

- Exercise 7.1
  - Write a procedure **simplify** to symbolically simplify summation expressions with numbers and symbols (lower-case letters). Let the procedure rearrange the expressions so that all the symbols precede numbers. These are examples of its use:

```
?- simplify( 1+1+a, E).
E= a+2
?- simplify( 1+a+4+2+b+c, E).
E= a+b+c+7
?- simplify( 3+x+x, E).
E= 2*x+3
```

7.2 Constructing and decomposing terms: =.., functor, arg, name

- There are three built-in predicates for decomposing terms and constructing new terms: **functor, arg** and ’**=..**’.
- The goal ‘**Term =.. L**’ is true if L is a list that contains the principal functor of Term, followed by its arguments.
  - For example:

```
| ?- f( a, b) =.. L.
L = [f,a,b]
yes
| ?- T =.. [rectangle, 3, 5].
T = rectangle(3,5)
yes
| ?- Z =.. [p, X, f( X, Y)].
Z = p( X, f( X, Y))
yes
```

7.2 Constructing and decomposing terms: =.., functor, arg, name

- Consider a program that manipulates geometric figures.
  - Figures are squares, rectangles, triangles, circles, etc.
  - They can be represented as follows:

    square(Side)
    triangle(Side1, Side2, Side3)
    circle(R) …
  - One operation on such figures can be enlargement(擴展).

    enlarge(Fig, Factor, Fig1).
  - For example:

```
enlarge( square(A), F, square(A1)) :- A1 is F*A.
enlarge( circle(R), F, circle(R1)) :- R1 is F*R.
enlarge( rectangle(A,B), F, rectangle(A1,B1)) :-
 A1 is F*A, B1 is F*B.
...
```

7.2 Constructing and decomposing terms: =.., functor, arg, name

- Can any clause be used to fit every above cases?

```
enlarge( Fig, F, Fig1) :-
    Fig =.. [Type | Parameters],
    multiplylist( Parameters, F, Parameters1),
    Fig1 =.. [Type | Parameters1].

multiplylist( [], _, []).

multiplylist([X | L], F, [X1 | L1]):-
    X1 is F*X, multiplylist( L, F, L1).
```

7.2 Constructing and decomposing terms: =.., functor, arg, name

```
| ?- enlarge( square(3), 2, X).
X = square(6)
yes

| ?- enlarge( square(3), 2, square( X)).
X = 6
yes

| ?- enlarge( circle(3), 2, circle( X)).
X = 6
yes

| ?- enlarge( triangle(3,4,5), 2, X).
X = triangle(6,8,10)
yes
```

7.2 Constructing and decomposing terms: =.., functor, arg, name

- The next example of using the '=..' predicate comes from symbolic manipulation of formulas where a frequent operation is to substitute some subexpression by another expression.

- Define the relation

  substitute(Subterm, Term, Subterm1, Term1)

  if all occurrences of **Subterm** in **Term** are substituted by **Subterm1** then we get **Term1**.

  - For example:

    ?- substitute(sin(x), 2*sin(x)*f(sin(x)), t, F)

    F = 2 * t * f(t)

7.2 Constructing and decomposing terms: =.., functor, arg, name

- By ‘occurrence’ of **Subterm** in **Term** we will mean something in **Term** that matches subterm.
- We will look for occurrences from top to bottom.
- For example:

  ?- substitute(a+b, f(a, A+B), v, F).

  will produce

      F = f(a,v)
      A = a
      B = b

  and not

      F = f(a,v+v)
      A = a+b
      B = a+b

7.2 Constructing and decomposing terms: =.., functor, arg, name

- In defining the **substitute** relation we have to consider the following decisions depending on the case:

  If **Subterm** = **Term** then **Term1** = **Subterm1**
  otherwise if **Term** is ‘atomic’ (not a structure)
  then **Term1** = **Term** (nothing to be substituted),
  otherwise the substitution is to be carried out on
  the arguments of **Term**.

```
substitute( Subterm, Term,        Subterm1, Term1)
substitute( sin(x),  f(sin(x)),   t,        F)
```

7.2 Constructing and decomposing terms: =.., functor, arg, name

```
%  Figure 7.3  A procedure for substituting a subterm of a term by
   another subterm.

% Case 1: Substitute whole term
substitute( Term, Term, Term1, Term1)  :-  !.

% Case 2: Nothing to substitute
substitute( _, Term, _, Term)  :- atomic(Term), !.

% Case 3: Do substitution on arguments
substitute( Sub, Term, Sub1, Term1)  :-
      Term  =..  [F|Args],
      substlist( Sub, Args, Sub1, Args1),
      Term1  =..  [F|Args1].

substlist( _, [], _, []).
substlist( Sub, [Term|Terms], Sub1, [Term1|Terms1])  :-
      substitute( Sub, Term, Sub1, Term1),
      substlist( Sub, Terms, Sub1, Terms1).
```

7.2 Constructing and decomposing terms: =.., functor, arg, name

```
substitute( Term, Term, Term1, Term1)  :-  !.
substitute( _, Term, _, Term)  :- atomic(Term), !.
substitute( Sub, Term, Sub1, Term1)  :-
        Term  =..  [F|Args],
        substlist( Sub, Args, Sub1, Args1),
        Term1  =..  [F|Args1].
substlist( _, [], _, []).
substlist( Sub, [Term|Terms], Sub1, [Term1|Terms1]) :-
        substitute( Sub, Term, Sub1, Term1),
        substlist( Sub, Terms, Sub1, Terms1).
```

```
{trace}
| ?- substitute( a+b, f(a, A+B), v, F).
      1    1  Call: substitute(a+b,f(a,_19+_20),v,_27) ?
      2    2  Call: atomic(f(a,_19+_20)) ?
      2    2  Fail: atomic(f(a,_19+_20)) ?
      2    2  Call: f(a,_19+_20)=..[_76|_77] ?
      2    2  Exit: f(a,_19+_20)=..[f,a,_19+_20] ?
      3    2  Call: substlist(a+b,[a,_19+_20],v,_147) ?
      4    3  Call: substitute(a+b,a,v,_133) ?
      5    4  Call: atomic(a) ?
      5    4  Exit: atomic(a) ?
      4    3  Exit: substitute(a+b,a,v,a) ?
      6    3  Call: substlist(a+b,[_19+_20],v,_134) ?
      7    4  Call: substitute(a+b,_19+_20,v,_212) ?
      7    4  Exit: substitute(a+b,a+b,v,v) ?
      8    4  Call: substlist(a+b,[],v,_213) ?
      8    4  Exit: substlist(a+b,[],v,[]) ?
      6    3  Exit: substlist(a+b,[a+b],v,[v]) ?
      3    2  Exit: substlist(a+b,[a,a+b],v,[a,v]) ?
      9    2  Call: _27=..[f,a,v] ?
      9    2  Exit: f(a,v)=..[f,a,v] ?
      1    1  Exit: substitute(a+b,f(a,a+b),v,f(a,v)) ?
A = a
B = b
F = f(a,v) ?
(32 ms) yes
{trace}
```

7.2 Constructing and decomposing terms: =.., functor, arg, name

- Term that are constructed by the '=..' predicate can be also used as goals.
- For example:

  Obtain(Functor),

  Compute(Arglist),

  Goal =.. [Functor|Arglist],

  Goal (or call(Goal))

  - Here, **obtain** and **compute** are some user-defined procedures for getting the components of the goal to be constructed.
  - The goal is then constructed by '=..', and invoked for execution by simply stating its name, **Goal**.

7.2 Constructing and decomposing terms: =.., functor, arg, name

- **functor** and **arg**
  - A goal **functor(Term, F, N)** is true if **F** is the principal functor of **Term** and **N** is the arity of **F**.
  - A goal **arg(N, Term, A)** is true if **A** is the **N**th argument of **Term**, assuming that arguments are numbered from left to right starting with 1.

  - For example:

```
|?- functor( t( f(X), X, t), Fun, Arity).
     Arity = 3
     Fun = t
     yes
| ?- arg(2, f(X, t(a), t(b)), Y).
     Y = t(a)
     yes
```

7.2 Constructing and decomposing terms: =.., functor, arg, name

- For example:

```
| ?- functor( D, date, 3),
       arg( 1, D, 29),
       arg( 2, D, june),
       arg( 3, D, 1982).
D = date(29,june,1982)
yes
```

- The definition of **name(A, L):**

  name(A, L) is true if L is the list of ASCII codes of the characters in atom A.

```
| ?- name( a, X).
X = [97]
yes
```

7.3 Various kinds of equality and comparison

- Three kinds of equality in Prolog:
  - **X = Y**

    This is true if X and Y match.
  - **X is E**

    This is true if X matches the value of the arithmetic expression E.
  - **E1 =:= E2**

    This is true if the values of the arithmetic expressions E1 and E2 are equal.
    - In contrast, when the values of two arithmetic expressions are not equal, we write **E1 =\= E2**.

7.3 Various kinds of equality and comparison

- The literal equality of two terms:
  - **T1 == T2**
    - This is true if term T1 and T2 are identical.
    - They have exactly the same structure and all the corresponding components are the same.
    - In particular, the names of the variables also have to be the same.
  - **T1 \== T2**
    - The complementary relation is 'not identical'.

7.3 Various kinds of equality and comparison

```
| ?- f(a, b)==f(a, b).
yes
| ?- f(a, b)==f(a, X).
no
| ?- f(a, X)==f(a, Y).
no
| ?- X \== Y.
yes
| ?- t(X, f(a,Y))==t(X, f(a,Y)).
yes
```

```
| ?- f(a, b) = f(a,X).
X = b
yes
```

```
| ?- X is 3 + 2.
X = 5
yes
```

7.3 Various kinds of equality and comparison

- Another example, redefine the relation:

  count(Term, List, N)

  where **N** is the number of literal occurrences of the term **Term** in a list **List**.

```
count( _, [], 0).
count( Term, [Head|L], N) :-
                Term == Head, !,
                count( Term, L, N1), N is N1 +1
                ;
                count( Term, L, N).
```

Compare to Section 7.1.

```
count( _, [], 0).
count( A, [A|L], N) :- !,
          count( A, L, N1), N is N1 +1.
count( A, [_|L], N) :- count( A, L, N).
```

7.3 Various kinds of equality and comparison

- Another set of built-in predicates compare terms alphabetically:

  X @< Y

  - Term X precedes term Y.
  - The precedence between structures is determined by the precedence of their principal functors.
  - If the principal functors are equal, then the precedence between the top-most, left-most functors in the subterms in X and Y decides.
  - All the built-in predicates in this family are @<, @=<, @>, @>= with their obvious meanings.

7.3 Various kinds of equality and comparison

```
| ?- paul @< peter.
yes

| ?- f(2) @< f(3).
yes

| ?- g(2) @< f(3).
no

| ?- g(2) @>= f(3).
yes

| ?- f(a, g(b), c) @< f(a, h(a), a).
yes
```

7.4 Database manipulation

- A Prolog program can be viewed as such a database:
  - The specification of relations is partly explicit (facts) and partly implicit (rules).
  - Some built-in predicates make it possible to update this database during the execution of the program.
  - This is done by adding new clauses to the program or by deleting existing clauses.
  - Predicates that serve these purposes are **assert**, **asserta**, **assertz** and **retract**.

7.4 Database manipulation

- **assert(C)**
  - A goal **assert(C)** always succeeds and causes a clause C to be ‘asserted’—that is, added to the database.
- **retract(C)**
  - A goal **retract(C)** deletes a clause that matches C.

```
| ?- data.
uncaught exception:
   error(existence_error(procedure,data/0),top_level/0)
| ?- asserta( data).
yes
| ?- data.
yes
| ?- retract( data).
yes
| ?- data.
no
```

7.4 Database manipulation

```
| ?- raining.
uncaught exception: error(existence_error(procedure,raining/0),top_level/0)
| ?- asserta(raining).
yes
| ?- asserta(fog).
yes
| ?- nice.
uncaught exception: error(existence_error(procedure,sunshine/0),nice/0)
| ?- disgusting.
yes
| ?- retract(fog).
yes
| ?- disgusting.
uncaught exception:
   error(existence_error(procedure,disgueting/0),top_level/0)
| ?- asserta(sunshine).
yes
| ?- funny.
yes
| ?- retract(raining).
yes
| ?- nice.
yes
```

```
nice :- sunshine, not raining.
funny :- sunshine, raining.
disgusting :- raining, fog.
```

7.4 Database manipulation

- Clauses of any form can be asserted or retracted.

```
| ?- asserta( (fast( ann)) ).
yes
| ?- asserta( (slow( tom)) ).
yes
| ?- asserta( (slow( pat)) ).
yes
| ?- asserta( (faster( X, Y) :- fast(X), slow(Y)) ).
yes
| ?- faster( A, B).
A = ann
B = pat ? ;
A = ann
B = tom
yes
| ?- retract( slow(X)).
X = pat ? ;
X = tom
yes
| ?- faster( ann, _).
no
```

7.4 Database manipulation

- **asserta(C)** and **assertz(C)**
  - The goal **asserta(C)** adds C at the beginning of the database.
  - The goal **assertz(C)** adds C at the end of the database.

```
| ?- assertz( p(b)), assertz( p(c)), assertz( p(d)),
   asserta( p(a)).
yes
| ?- p(X).
X = a ? ;
X = b ? ;
X = c ? ;
X = d
yes
```

7.4 Database manipulation

- The relation between **consult** and **assertz:**
  - Consulting a file can be defined in terms of **assertz** as:

    To consult file, read each term (clause) in the file and assert it at the end of the database.
- One useful application of **asserta** is to store already computed answers to questions.
  - Let there be a predicate **solve(Problem, Solution)** defined in the program.
  - We may now ask some question and request that the answer be remembered for future questions.

    ?- solve(Problem1, Solution),
    asserta(solve(Problem1, Solution)).
    - If the first goal above succeeds then the answer **(Solution)** is stored and used in answering further questions.

7.4 Database manipulation

- Another example of **asserta:**
  - Generate a table of products of all pairs of integers between 0 and 9 as follows:
    - generate a pair of integers X and Y,
    - compute Z is X * Y,
    - assert the three numbers as one line of the product table, and then
    - force the failure
      - The failure will cause, through backtracking, another pair of integers to be found and so another line tabulated.

```
maketable :-
        L = [1,2,3,4,5,6,7,8,9],
        member( X, L), member( Y, L),
        Z is X * Y, asserta( product( X, Y, Z)), fail.
```

7.4 Database manipulation

```
maketable :-
        L = [1,2,3,4,5,6,7,8,9],
        member( X, L), member( Y, L),
        Z is X * Y, asserta( product( X, Y, Z)), fail.
```

For backtracking

```
| ?- maketable.
no

| ?- product( A, B, 8).
A = 8
B = 1 ? ;
A = 4
B = 2 ? ;
A = 2
B = 4 ? ;
A = 1
B = 8 ? ;
(31 ms) no
```

```
| ?- product( 2, 5, X).
X = 10 ?
yes
```

Exercise

- Exercise 7.6
  - Write a Prolog question to remove the whole product table from the database.
  - Modify the question so that it only removes those entries where the product is 0.

7.5 Control facilities

- The complete set of control facilities is presented here.
  - **cut** (!) prevents backtracking.

    once(P) :- P, !.

    once(P) produces one solution only.
  - **fail** is a goal that always fails.
  - **true** is a goal that always succeeds.
  - **not(P)** is negation as failure that behaves exactly as if defined as:

    not(P) :- P, !, fail; true.
  - **call(P)** invokes a goal P. It succeeds if P succeeds.
  - **repeat** is a goal that always succeeds.

7.5 Control facilities

- **repeat:**
  - **repeat** is a goal that always succeeds.
  - It is non-deterministic.
  - Each time it is reached by backtracking it generates another alternative execution branch.
  - **repeat** behaves as if defined by:

```
repeat.
repeat :- repeat.
```

  - An example:

```
dosquares :- repeat, read(X),
                     (X = stop, !
                      ;
                      Y is  X * X, write(Y), fail).
```

7.5 Control facilities

```
dosquares :- repeat, read(X),
             (X = stop, !
              ;
              Y is  X * X, write(Y), fail).
```

```
| ?- dosquares.
3.
9 4.
16 5.
25 6.
36 7.
49 8.
64 9.
81 10.
100 stop.
yes
```

7.6 bagof, setof and findall

- We can generate, by backtracking, all the objects, one by one, that satisfy some goal.
- Each time a new solution is generated, the previous one disappears and is not accessible any more.
- Sometimes we would prefer to have all the generated objects available together—for example collected into a list.
- The built-in predicates **bagof**, **setof**, and **findall** serve this purpose.

7.6 bagof, setof and findall

- **bagof**
  - The goal **bagof(X, P, L)** will produce the list **L** of all the objects **X** such that a goal **P** is satisfied.
  - If there is no solution for P in the **bagof** goal, then the goal simply fails.
  - If the same object X is found repeatedly, then all of its occurrences will appear in L, which leads to duplicate items in L.
  - For example:

```
age( peter, 7).
age( ann, 5).
age( pat, 8).
age( tom, 5).

?- bagof( Child, age( Child, 5), List).
List = [ann,tom]
yes
```

7.6 bagof, setof and findall

```
age( peter, 7).
age( ann, 5).
age( pat, 8).
age( tom, 5).
```

```
?- bagof( Child, age( Child, 5), List).
List = [ann,tom]
Yes

| ?- bagof( Child, age( Child, Age), List).
Age = 5
List = [ann,tom] ? ;
Age = 7
List = [peter] ? ;
Age = 8
List = [pat]
(15 ms) yes

| ?- bagof( Child, Age ^ age( Child, Age), List).
List = [peter,ann,pat,tom]
Yes
| ?- bagof( Child, Age ^ age( Child, 5), List).
List = [ann,tom]
yes
| ?- bagof( Child, 5 ^ age( Child, 5), List).
List = [ann,tom]
yes
```

'^' is a predefined infix operator of type xfy.
→We do not care about the value of **Age**.

7.6 bagof, setof and findall

- **setof**
  - The goal **setof(X, P, L)** will produce a list **L** of objects **X** that satisfy **P**.
  - The list **L** will be ordered, and duplicate items will be eliminated.
  - For example:

```
age( peter, 7).
age( ann, 5).
age( pat, 8).
age( tom, 5).

?- setof( Child, Age ^ age( Child, Age), ChildList),
   setof( Age, Child ^ age( Child, Age), AgeList).

  AgeList = [5,7,8]
  ChildList = [ann,pat,peter,tom]
  yes
```

7.6 bagof, setof and findall

```
| ?- setof( Child, Age ^ age( Child, Age), ChildList),
     setof( Age, Child ^ age( Child, Age), AgeList).
    AgeList = [5,7,8]
    ChildList = [ann,pat,peter,tom]
    Yes

| ?- bagof( Child, Age ^ age( Child, Age), ChildList),
     bagof( Age, Child ^ age( Child, Age), AgeList).
    AgeList = [7,5,8,5]
    ChildList = [peter,ann,pat,tom]
    Yes
```

```
age( peter, 7).
age( ann, 5).
age( pat, 8).
age( tom, 5).
```

```
?- setof( Age/Child, age( Child, Age), List).
    List = [5/ann,5/tom,7/peter,8/pat]
    yes
```

7.6 bagof, setof and findall

- **findall**
  - The goal **findall(X, P, L)** produces a list **L** of objects **X** that satisfy **P**.
  - The difference will respect to **bagof** is that all of the objects **X** are collected regardless of different solutions for variables in **P** that are not shared with **X.**
  - If there is no object **X** that satisfies **P** then **findall** will succeed with **L** = [].
  - For example:

```
age( peter, 7).
age( ann, 5).
age( pat, 8).
age( tom, 5).
```

```
| ?- findall( Child, age( Child, Age), List).
List = [peter,ann,pat,tom]
yes
```

```
| ?- bagof( Child,
age( Child, Age), List).
Age = 5
List = [ann, tom] ? ;
Age = 7
List = [peter] ? ;
Age = 8
List = [pat]
(15 ms) yes
```

7.6 bagof, setof and findall

- If **findall** is not available as a built-in predicate in the implementation used then it can be easily programmed as follows.

```
% Figure 7.4   An implementation of the findall relation.
findall( X, Goal, Xlist)  :-
  call( Goal),                          % Find a solution
  assertz( queue(X) ),                  % Assert it
  fail;                                 % Try to find more solutions
  assertz( queue(bottom) ),             % Mark end of solutions
  collect( Xlist).                      % Collect the solutions

collect( L)  :-
  retract( queue(X) ), !,               % Retract next solution
  ( X == bottom, !, L = []              % End of solutions?
    ;
    L = [X | Rest], collect( Rest) ).   % Otherwise collect the rest
```

Exercise

- Exercise 7.8
  - Use **setof** to define the relation

    powerset(Set, Subsets)

    to compute the set of all subsets of a given set (all sets represented as lists).