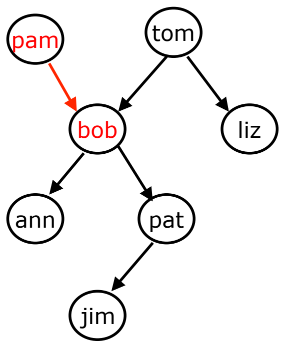
A short introduction to Prolog

(Highlights from Chapters 1, 2, and 3 in Bratko, *Prolog Programming for AI*, A-W, 2001)

Part 1: Prolog

1.1 Defining relations by facts

- Given a whole family tree

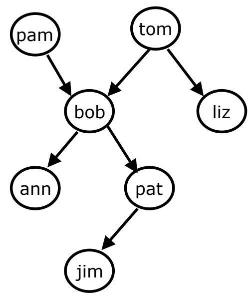


- The tree defined by the Prolog program:

```
parent( pam, bob).
   % Pam is a parent of Bob
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```

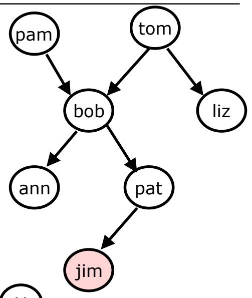

1.1 Defining relations by facts

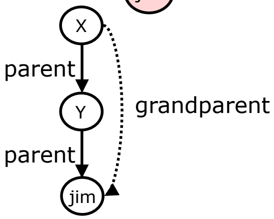
- Questions:
  - Is Bob a parent of Pat?
    - ?- parent(bob, pat).
    - ?- parent(liz, pat).
    - ?- parent(tom, ben).
  - Who is Liz's parent?
    - ?- parent(X, liz).
  - Who are Bob's children?
    - ?- parent(bob, X).



1.1 Defining relations by facts

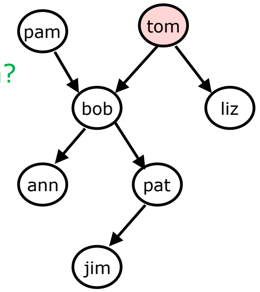
- Questions:
  - Who is a parent of whom?
    - Find X and Y such that X is a parent of Y.
    - ?- parent(X, Y).
  - Who is a grandparent of Jim?
    - ?- parent(Y, jim),
      parent(X, Y).





1.1 Defining relations by facts

- Questions:
  - Who are Tom's grandchildren?
    - ?- parent(tom, X),
      parent(X, Y).
  - Do Ann and Pat have a common parent?
    - ?- parent(X, ann),
      parent(X, pat).



1.1 Defining relations by facts

- It is easy in Prolog to define a relation
- The user can easily query the Prolog system about relations defined in the program
- A Prolog program consists of clauses.
  - Each clause terminates with a full stop
- The arguments of relations can be
  - Atoms: concrete objects or constants
  - Variables: general objects such as X and Y
    - Also numbers and structures
- Questions to the system consist of one or more goals
- An answer to a question can be either
  - positive (succeeded) or
  - negative (failed)
- If several answers satisfy the question then Prolog will find as many of them as desired by the user

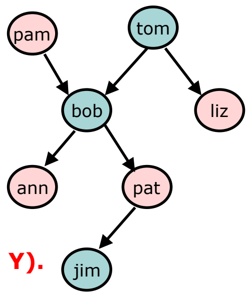
Example we will use (Prolog rules)

```
% Figure 1.8   The family program.

parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

female( pam).
female( liz).
female( ann).
female( pat).
male( tom).
male( bob).
male( jim).

offspring( Y, X)  :-
   parent( X, Y).
```

```
mother( X, Y)  :-
   parent( X, Y),
   female( X).

grandparent( X, Z)  :-
   parent( X, Y),
   parent( Y, Z).

sister( X, Y)  :-
   parent( Z, X),
   parent( Z, Y),
   female( X),
   X \= Y.

predecessor( X, Z)  :-   % Rule pr1
   parent( X, Z).

predecessor( X, Z)  :-   % Rule pr2
   parent( X, Y),
   predecessor( Y, Z).
```

1.2 Defining relations by rules

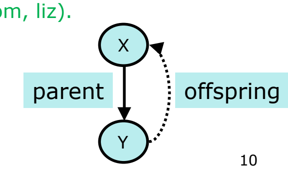
- Facts:
  - female(pam).   % Pam is female
  - female(liz).
  - female(ann).
  - female(pat).
  - male(tom).   % Tom is male
  - male(bob).
  - male(jim).



- Define the "offspring" relation:
  - Fact: offspring(liz, tom).
  - Rule: **offspring(Y, X) :- parent(X, Y).**
    - For all X and Y,
      Y is an offspring of X if
      X is a parent of Y

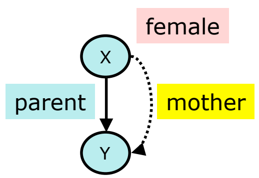
1.2 Defining relations by rules

- **Rules** have:
  - A conclusion part (head)
    - the left-hand side of the rule
  - A condition part (body)
    - the right-hand side of the rule
  - Example:
    - **offspring(Y, X) :- parent(X, Y).**
    - The rule is general in the sense that it is applicable to any objects X and Y
    - A special case of the general rule:
      - offspring(liz, tom) :- parent(tom, liz).
    - ?- offspring(liz, tom).
    - ?- offspring(X, Y).



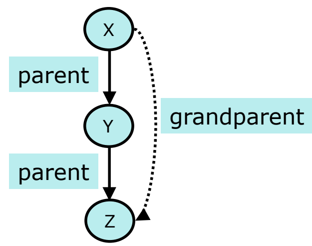
1.2 Defining relations by rules

- Define the "mother" relation:
  - **mother(X, Y) :- parent(X, Y), female(X).**
  - For all X and Y,
    X is the mother of Y if
    X is a parent of Y and
    X is a female



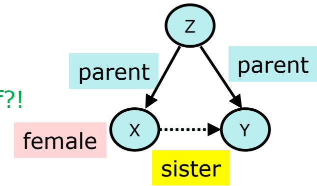
1.2 Defining relations by rules

- Define the "grandparent" relation:
  - **grandparent(X, Z) :-**
    parent(X, Y), parent(Y, Z).



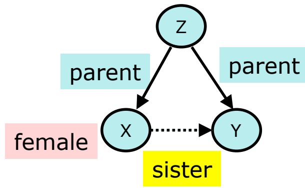
1.2 Defining relations by rules

- Define the “sister” relation:
  - **sister(X, Y) :-**
    parent(Z, X), parent(Z, Y), female(X).
  - For any X and Y,
    X is a sister of Y if
    (1) both X and Y have the same parent, and
    (2) X is female
  - ?- sister(ann, pat).
  - ?- sister(X, pat).
  - ?- sister(pat, pat).
    - Pat is a sister to herself?!



1.2 Defining relations by rules

- To correct the “sister” relation:
  - **sister(X, Y) :-**
    parent(Z, X), parent(Z, Y), female(X),
    different(X, Y).
  - different (X, Y) is satisfied if and only if X and Y are not equal.



1.2 Defining relations by rules

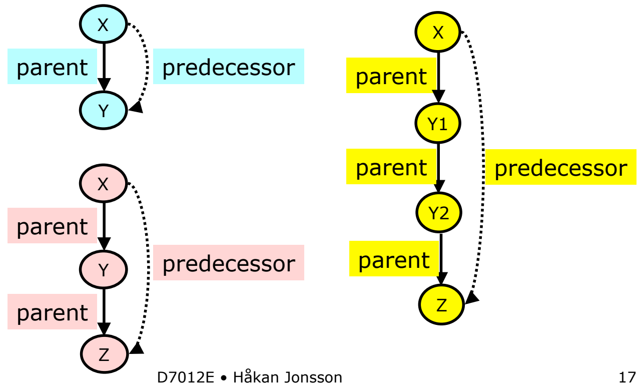
- Prolog clauses consist of
  - Head
  - Body: a list of goals separated by commas (,)
- Prolog clauses are of three types:
  - Facts:
    - declare things that are always true
    - facts are clauses that have a head and an empty body
  - Rules:
    - declare things that are true depending on a given condition
    - rules have the head and the (non-empty) body
  - Questions:
    - the user can ask the program what things are true
    - questions only have the body

1.2 Defining relations by rules

- A variable can be substituted by another object
- Variables are assumed to be universally quantified and are read as “for all”
  - For example:
    hasachild(X) :- parent(X, Y).
    can be read in two way
    (a) For all X and Y,
    if X is a parent of Y then X has a child
    (b) For all X,
    X has a child if there is some Y such that X is a parent of Y

1.3 Recursive rules

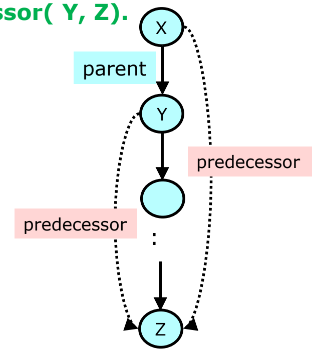
- Define the “predecessor” relation



1.3 Recursive rules

- Define the “predecessor” relation

```
predecessor( X, Z):- parent( X, Z).
predecessor( X, Z):-
    parent( X, Y), predecessor( Y, Z).
```



  - For all X and Z,
    X is a predecessor of Z if
    there is a Y such that
    (1) X is a parent of Y and
    (2) Y is a predecessor of Z
  - ?- predecessor(pam, X).

1.3 Recursive rules

```
% Figure 1.8   The family program.

parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

female( pam).
female( liz).
female( ann).
female( pat).
male( tom).
male( bob).
male( jim).

offspring( Y, X)  :-
   parent( X, Y).

mother( X, Y)  :-
   parent( X, Y),
   female( X).

grandparent( X, Z)  :-
   parent( X, Y),
   parent( Y, Z).

sister( X, Y)  :-
   parent( Z, X),
   parent( Z, Y),
   female( X),
   X \= Y.

predecessor( X, Z)  :-   % Rule pr1
   parent( X, Z).

predecessor( X, Z)  :-   % Rule pr2
   parent( X, Y),
   predecessor( Y, Z).
```


1.3 Recursive rules

- Procedure:
  - In figure 1.8, there are two “predecessor relation” clauses
    predecessor(X, Z) :- parent(X, Z).
    predecessor(X, Z) :- parent(X, Y), predecessor(Y, Z).
  - Such a set of clauses is called a **procedure**

- Comments:
  /* This is a comment */
  % This is also a comment

1.4 How Prolog answers questions

- To answer a question, Prolog tries to satisfy all the goals
- To satisfy a goal means to demonstrate that the goal is true, assuming that the relations in the program is true
- Prolog accepts facts and rules as a set of axioms, and the user's question as a conjectured theorem
- Example:
  - Axioms: All men are fallible
    Socrates is a man
  - Theorem: Socrates is fallible
  - For all X, if X is a man then X is fallible

```
fallible( X) :- man( X).
man( socrates).
```

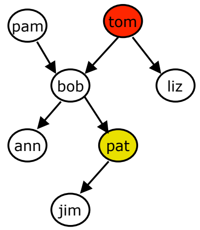
    - ?- fallible(socrates).

1.4 How Prolog answers questions

```
predecessor( X, Z)  :- parent( X, Z).        % Rule pr1
predecessor( X, Z)  :- parent( X, Y),        % Rule pr2
                       predecessor( Y, Z).
```

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```

- ?- predecessor(tom, pat).
  - How does the Prolog system actually find a proof sequence?
    - Prolog first tries that clause which appears first in the program. (rule pr1)
    - Now, X= tom, Z = pat.
    - The goal predecessor(tom, pat) is then replace by parent(tom, pat)
    - There is no clause in the program whose head matches the goal parent(tom, pat)
    - Prolog backtracks to the original goal in order to try an alternative way (rule pr2)

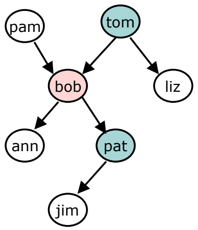


1.4 How Prolog answers questions

```
predecessor( X, Z)  :- parent( X, Z).        % Rule pr1
predecessor( X, Z)  :- parent( X, Y),        % Rule pr2
                       predecessor( Y, Z).
```

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```

- ?- predecessor(tom, pat).
    - Apply rule pr2, X = tom, Z = pat,
      but Y is not instantiated yet
    - The top goal predecessor(tom, pat) is replaces by two goals:
      - parent(tom, Y)
      - predecessor(Y, pat)
    - The first goal matches one of the facts. (Y = bob)
    - The remaining goal has become
      predecessor(bob, pat)
    - Using rule pr1, this goal can be satisfied
      - predecessor(bob, pat) :- parent(bob, pat)



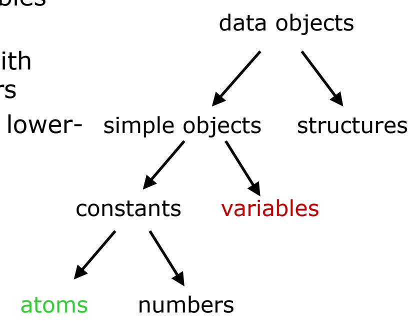
1.5 Declarative and procedural meaning of programs

- Two levels of meaning of Prolog programs:
  - The declarative meaning
    - concerned only with the relations defined by the program
    - determines what will be the output of the program
    - The programmer should concentrate mainly on the declarative meaning and avoid being distracted by the executional details
  - The procedural meaning
    - determines how this output is obtained
    - determines how the relations are actually evaluated by the Prolog system
    - The procedural aspects cannot be completely ignored by the programmer for practical reasons of executional efficiency

Part 2: More on the syntax and meaning of Prolog programs

2.1 Data Objects

- We have seen variables and atoms:
  - Variables start with upper-case letters
  - Atoms start with lower-case letters
  - There are also structures and numbers



2.1.1 Atoms and numbers

- **Atoms** can be constructed in three ways:
  - Strings of letters, digits and the underscore character,'_', starting with a lower case letter
    - anna, x25, x_35AB, x___y, miss_Jones
  - Strings of special characters
    - <--->, ===>, ...,::=,.:., (except **:-**)
  - Strings of characters enclosed in single quotes
    - 'Tom', 'South_America', 'Sarah Jones'
- **Numbers** used in Prolog include integer numbers and real numbers
    - Integer numbers: 1313, 0, -97
    - Real numbers: 3.14, -0.0035, 100.2
  - In symbolic computation, integers are often used

2.1.2 Variables

- **Variables** start with an upper-case letter or an underscore character
  - Examples: X, Result, _x23, _23
- Anonymous variables:
  - Examples:

```
hasachild( X) :- parent( X, Y).
hasachild( X) :- parent( X, _).
```

```
?- parent( X, _)
```

  - We are interested in people who have children, but not in the names of the children
  - _ is also called a "wildcard"

2.1.2 Variables

- The lexical scope of variable names is one clause
  - If the name X occurs in two clauses, then it signifies two different variables

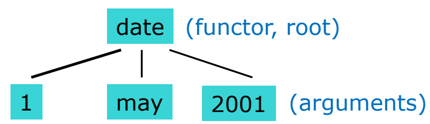
    hasachild(X) :- parent(X, Y).
    isapoint(X) :- point(X, Y, Z).

  - But each occurrence of X with in the same clause means the same variables

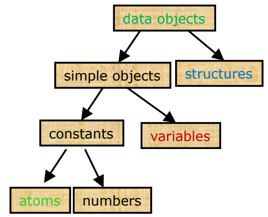
    hasachild(X) :- parent(X, Y).
- The same atom always means the same object in any clause throughout the whole program

2.1.3 Structures

- **Structured objects** are objects that have several components
- All structured objects can be pictured as trees.
  - The root of the tree is the functor
  - The offsprings of the root are the components
  - Components can also be variables or other structures
    date(Day, may, 2001)
  - Example: **date(1, may, 2001)**



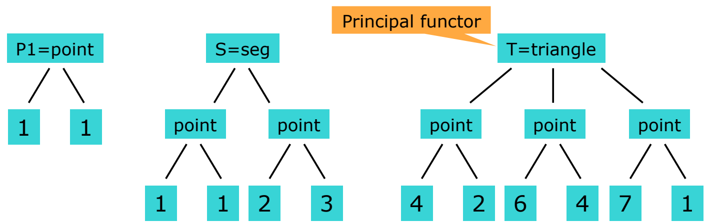
- All **data objects** in Prolog are terms



2.1.3 Structures

- Three structures and their tree representations:

  P1 = point(1, 1)
  P2 = point(2, 3)
  S = seg(P1, P2)
  = seg(point(1,1), point(2,3))
  T = triangle(point(4,2), point(6,4), point(7,1))



2.1.3 Structures

- Each functor is defined by two things:
  - The name, whose syntax is that of atoms;
  - The arity, that is, the number of arguments
  - For example:

    point(X1, Y1) and **point(X, Y, Z)** are different
    - The Prolog system will recognize the difference by the number of arguments, and will interpret this name (**point**) as two functors

2.2 Matching

- The most important operation on terms is matching
- Matching is a process that takes as input two terms and checks whether they match
  - Fails: if the terms do not match
  - Succeeds: if the terms do match
- Given two terms, we say that they match if:
  - they are identical, or
  - the variable in both terms can be instantiated to objects in such a way that after the substitution of variables by these objects the terms become identical
  - For example:
    - the terms **date(D, M, 2001)** and **date(D1, may, Y1)** match
    - the terms **date(D, M, 2001)** and **date(D1, M1, 1444)** do not match

2.2 Matching

- The request for matching, using the operator '=':

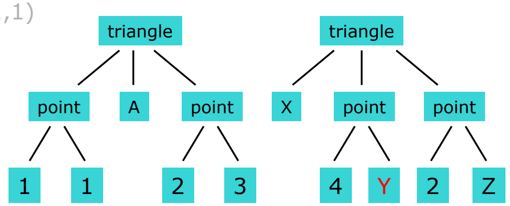
```
| ?- date( D, M, 2001) =
     date(D1, may, Y1).

D1 = D
M = may
Y1 = 2001
Yes

| ?- date( D, M, 2001) =
     date(D1, may, Y1),
      date( D, M, 2001) =
     date( 15, M, Y).

D = 15
D1 = 15
M = may
Y = 2001
Y1 = 2001
yes
```

- Matching in Prolog always
  - results in the most general instantiation
  - leaves the greatest possible freedom for further instantiations if further matching is required

2.2 Matching

- Matching

```
| ?- triangle( point(1,1), A, point(2,3))=
triangle( X, point(4,Y), point(2,Z)).

A = point(4,Y)
X = point(1,1)
Z = 3
yes
```



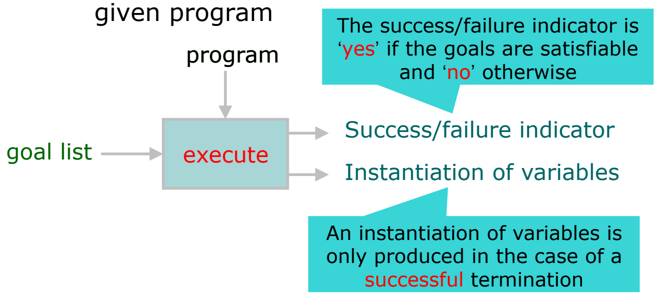
2.3 Declarative meaning of Prolog programs

- Consider a clause:

  P :- Q, R.
  - Some declarative readings of this clause are:
    - P is true if, and only if, Q and R are true
    - From Q and R follows P
  - Two procedural readings of this clause are:
    - To solve problem P, *first* solve the subproblem Q and *then* the subproblem R
    - To satisfy P, *first* satisfy Q and *then* R
  - Difference:
    - The procedural readings do not only define the logical relations between the head of the clause and the goals in the body, but also the order in which the goals are processed

2.4 Procedural meaning

- The procedural meaning:
  - The procedural meaning specifies how Prolog answers questions
  - The procedural meaning of Prolog is a procedure for executing a list of goals with respect to a given program



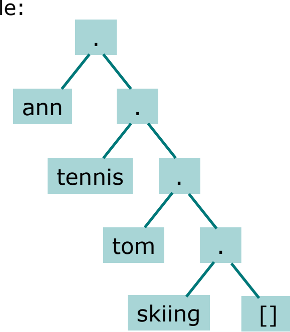
Part 3 Lists, operators, and arithmetic

3.1 Representation of list

- A list is a sequence of any number of items.
- For example:
  - [ann, tennis, tom, skiing]
- A list is either empty or non-empty.
  - Empty: []
  - Non-empty:
    - The first term, called the head of the list
    - The remaining part of the list, called the tail
    - Example: [ann, tennis, tom, skiing]
      - Head: ann
      - Tail: [tennis, tom, skiing]

3.1 Representation of list

- In general,
  - the head can be anything (for example: a tree or a variable)
  - the tail has to be a list
- The head and the tail are then combined into a structure by a special functor **.(head, Tail)**
  - For example:
    L = .(ann, .(tennis, .(tom, .(skiing, [])))).
    L = [ann, tennis, tom, skiing].
    are the same in Prolog.
  - Prolog accept lists written as:
    - [Item1, Item2,…]
    - [Head | Tail]
    - [Item1, Item2, …| Other]



3.1 Representation of lists

```
| ?- List1 = [a,b,c],
     List2 = .(a, .(b, .(c,[]))).

List1 = [a,b,c]
List2 = [a,b,c]

yes

| ?- Hobbies1 = .(tennis, .(music, [])),
     Hobbies2 = [skiing, food],
     L = [ann, Hobbies1, tom, Hobbies2].

Hobbies1 = [tennis,music]
Hobbies2 = [skiing,food]
L = [ann,[tennis,music],tom,[skiing,food]]

yes
```

```
| ?- L= [a|Tail].

L = [a|Tail]

yes

| ?- [a|Z] = .(X, .(Y, [])).

X = a
Z = [Y]

yes

| ?- [a|[b]] = .(X, .(Y, [])).

X = a
Y = b

yes
```


3.2 Some operations on lists

- Two common operations on lists are:
  - Member
  - Concatenation of two lists, obtaining a third list
    - which may correspond to the union of sets;

3.2.1 Membership

- The membership relation:
  member(X, L)
  where X is an object and L is list.
- The goal **member(X, L)** is true if X occurs in L.
- X is a member of L if either:
  (1) X is the head of L, or
  (2) X is a member of the tail of L.

```
member1( X, [X| Tail]).
member1( X, [Head| Tail]) :-  member1( X, Tail).
```

- For example:
  member(b, [a, b, c]) is true
  member(b, [a, [b, c]]) is not true
  member([b, c] , [a, [b, c]]) is true

3.2.2 Concatenation

- The concatenation relation:
  conc(L1, L2, L3)
  here L1 and L2 are two lists, and L3 is their concatenation.

- Definition

```
conc( [], L, L).
conc( [X| L1], L2, [X| L3]) :- conc( L1, L2, L3).
```

- For example:
  conc([a, b], [c, d], [a, b, c, d]) is true
  conc([a, b], [c, d], [a, b, a, c, d]) is not true

3.2.2 Concatenation

conc([], L, L).
conc([X| L1], L2, [X| L3]) :- conc(L1, L2, L3).

```
| ?- conc([a,b,c], [1,2,3], L).

L = [a,b,c,1,2,3]
yes

| ?- conc([a,[b,c],d], [a,[],b] ,L).

L = [a,[b,c],d,a,[],b]
yes
```

```
| ?- conc(L1, L2, [a,b,c]).

L1 = []
L2 = [a,b,c] ? ;

L1 = [a]
L2 = [b,c] ? ;

L1 = [a,b]
L2 = [c] ? ;

L1 = [a,b,c]
L2 = [] ? ;

no
```


3.2.2 Concatenation

```
| ?- conc( Before, [may| After], [jan, feb, mar, apr, may, jun, jul,
     aug, sep, oct, nov, dec]).
After = [jun,jul,aug,sep,oct,nov,dec]
Before = [jan,feb,mar,apr] ? ;
no

| ?- conc( _, [Month1,may, Month2|_], [jan, feb, mar, apr, may, jun,
     jul, aug, sep, oct, nov, dec]).
Month1 = apr
Month2 = jun ? ;
No

| ?- L1 = [a,b,z,z,c,z,z,z,d,e], conc(L2,[z,z,z|_ ], L1).
L1 = [a,b,z,z,c,z,z,z,d,e]
L2 = [a,b,z,z,c] ? ;
no
```


3.2.2 An alternative member

- Define the membership relation:
  member2(X, L):- conc(L1,[X|L2],L).
  X is a member of list L if L can be decomposed into two lists so that the second one has X as its head. Since we do not bother about L1 and L2:

  member2(X, L):- conc(_,[X|_],L).

  - Compare to the member relation defined on 3.2.1:
    member1(X, [X| Tail]).
    member1(X, [Head| Tail]) :- member1(X, Tail).

conc([], L, L).
conc([X| L1], L2, [X| L3]) :- conc(L1, L2, L3).

3.4 Arithmetic

- Predefined basic arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division
**	power
//	integer division
mod	modulo, the remainder of integer division

```
| ?- X = 1+2.
X = 1+2
yes
| ?- X is 1+2.
X = 3
yes
```

Operator ‘**is**’ is a built-in procedure.

3.4 Arithmetic

- Predefined comparison operators:

X > Y	X is greater than Y
X < Y	X is less than Y
X >= Y	X is greater than or equal to Y
X =< Y	X is less than or equal to Y
X =:= Y	the values of X and Y are equal
X =\= Y	the values of X and Y are not equal

```
| ?- 1+2 =:= 2+1.
yes
| ?- 1+2 = 2+1.
no
| ?- 1+A = B+2.
A = 2
B = 1
yes
```


3.4 Arithmetic

- Length counting problem:
  (Note: **length** is a build-in procedure)
  - Define procedure **length(List, N)** which will count the elements in a list **List** and instantiate **N** to their number.
    (1) If the list is empty then its length is 0.
    (2) If the list is not empty then **List = [Head|Tail];** then its length is equal to 1 plus the length of the tail **Tail**.
  - These two cases correspond to the following program:

```
length1( [], 0).
length1( [ _| Tail], N) :- length1( Tail, N1),
                           N is 1 + N1.
?- length1( [a, b, [c, d], e], N).
N = 4
```


3.4 Arithmetic

- Another programs:

```
length2( [], 0).
length2( [_ | Tail], N) :- length2( Tail, N1),
                           N = 1 + N1.

| ?- length2( [a, b, [c, d], e], N).
N = 1+(1+(1+(1+0)))

length3( [], 0).
length3( [_ | Tail], N) :- N = 1 + N1,
                           length3( Tail, N1).
➡ length3( [_ | Tail], 1 + N) :- length3( Tail, N).

| ?- length3([a,b,c],N), Length is N.
Length = 3
N = 1+(1+(1+0))
```



```
length :: [a] -> Int
length []     = 0
length (_:l)  = 1 + length l
```

```
length([],0).
length([_|L],N):-length(L,N0),
   N is 1+N0.
```

```
member :: (Eq a) => a->[a]->Bool
member x []      =  False
member x (y:ys) =  x == y ||
                   member x ys
```

```
member(X,[X|_]).
member(X,[_|Ys]):-member(X,Ys).
```



```
head :: [a] -> a
head (x:_)  =  x
```

```
head([X|_],X).
```

```
tail :: [a] -> [a]
tail (_:xs) =  xs
```

```
tail([_|Xs],Xs).
```

```
null :: [a] -> Bool
null []     =  True
null (_:_)  =  False
```

```
null([]).
```