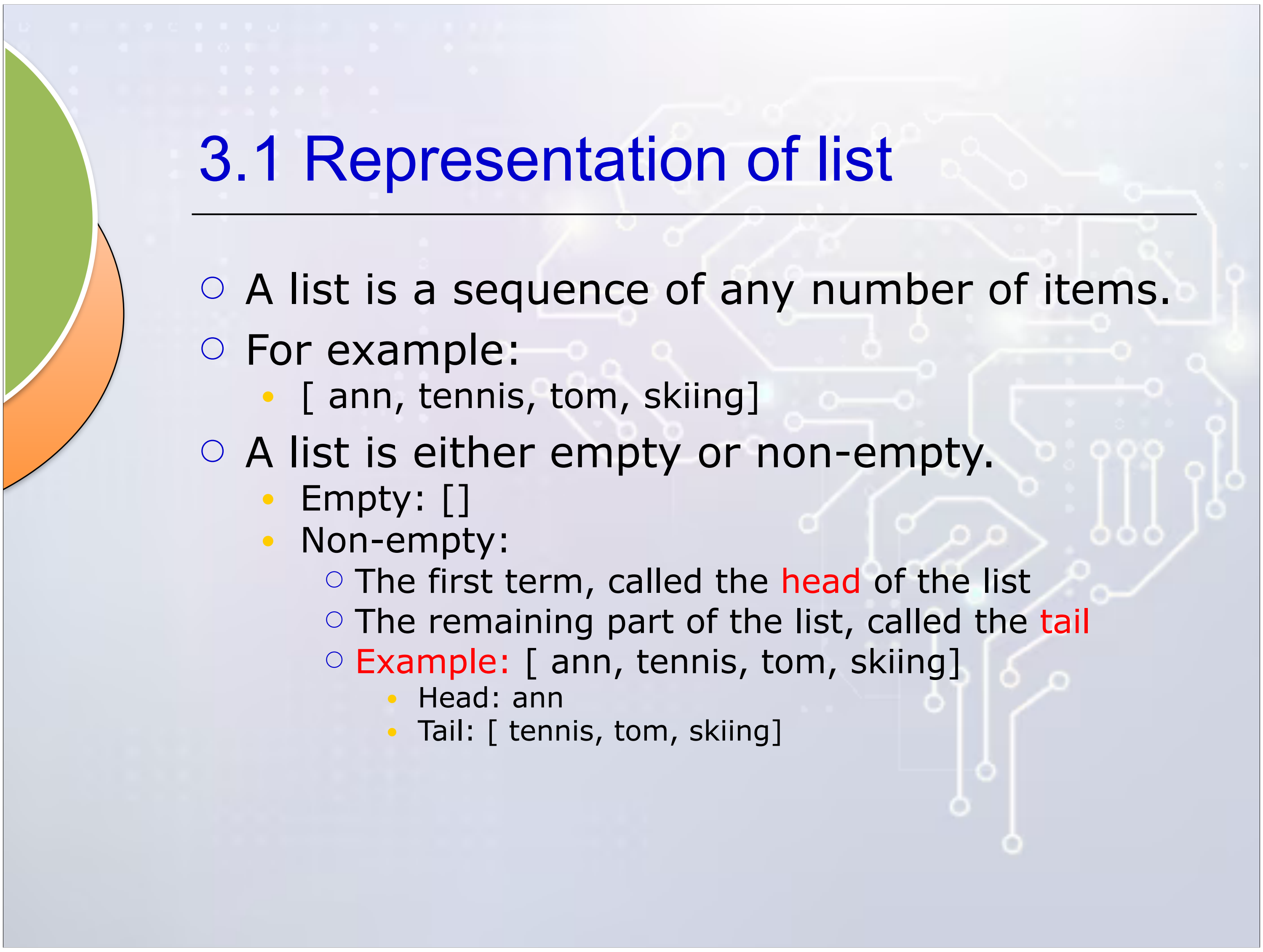
Part 1 The Prolog Language

Chapter 3
Lists, Operators, Arithmetic

By-Divyanshu Pathak,
Divjot Singh

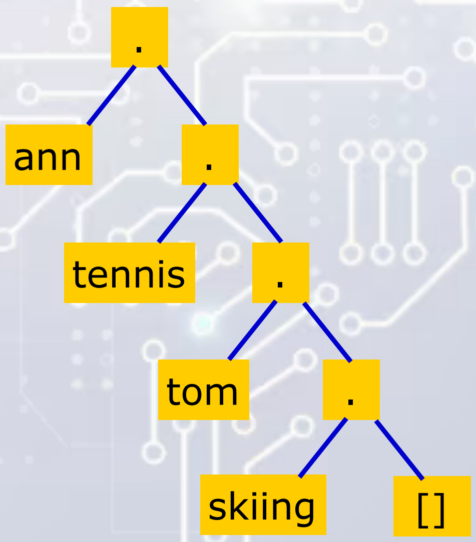


3.1 Representation of list

- 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

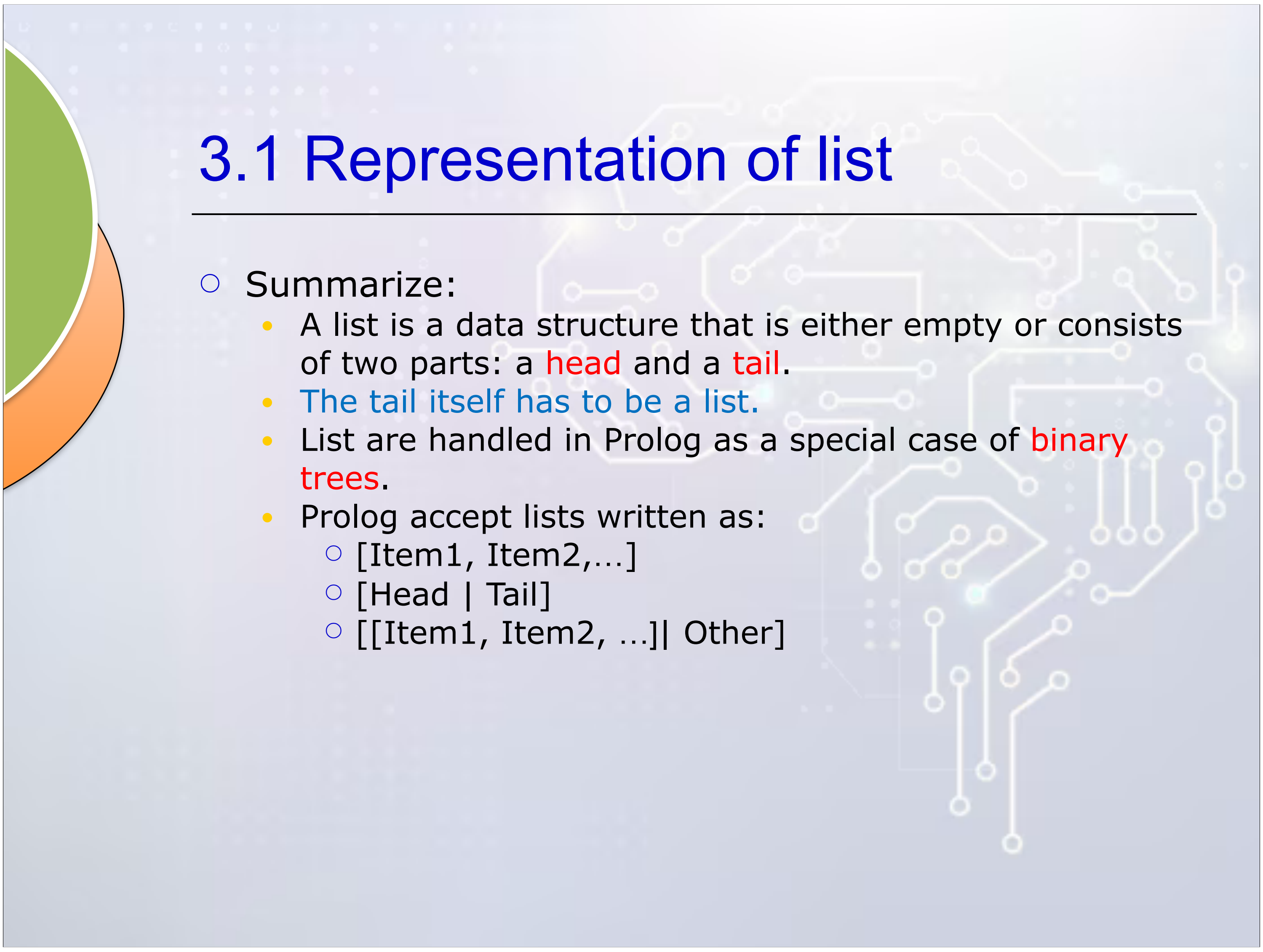
- In general,
  - the head can be anything (for example: a tree or a variable)
  - the tail has to be a list
- The head and the tail are then combined into a structure by a special functor
  .(head, Tail)
  - For example:
    .(ann, .(tennis, .(tom, .(skiing, []))))
    [ann, tennis, tom, skiing]
    are the same in Prolog.



3.1 Representation of list

```
| ?- 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
```



3.1 Representation of list

- Summarize:
  - A list is a data structure that is either empty or consists of two parts: a head and a tail.
  - The tail itself has to be a list.
  - List are handled in Prolog as a special case of binary trees.
  - Prolog accept lists written as:
    - [Item1, Item2,…]
    - [Head | Tail]
    - [[Item1, Item2, …]| Other]

3.2 Some operations on lists

- The most common operations on lists are:
  - Checking whether some object is an element of a list, which corresponds to checking for the set membership;
  - Concatenation of two lists, obtaining a third list, which may correspond to the union of sets;
  - Adding a new object to a list, or deleting some object form it.

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.
  - 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

3.2.1 Membership

- 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).
```

```
| ?- member1(3, [1,2,3,4]).
      1    1  Call: member1(3,[1,2,3,4]) ?
      2    2  Call: member1(3,[2,3,4]) ?
      3    3  Call: member1(3,[3,4]) ?
      3    3  Exit: member1(3,[3,4]) ?
      2    2  Exit: member1(3,[2,3,4]) ?
      1    1  Exit: member1(3,[1,2,3,4]) ?
true ?
Yes
```

3.2.2 Concatenation

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

  - 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

- Two case of concatenation relation:

  (1) If the first argument is the empty list then the second and the third arguments must be the same list.

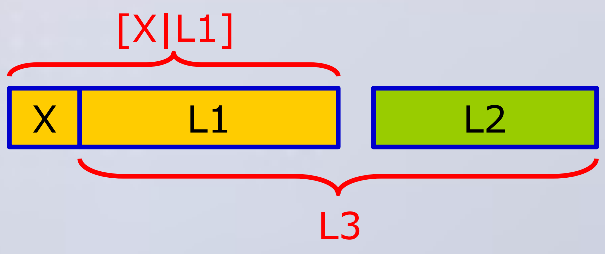
  conc([], L, L).

  (2) If the first argument is an non-empty list then it has a head and a tail and must look like this

  [X | L1]

  the result of the concatenation is the list [X| L3] where L3 is the concatenation of L1 and L2.

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



3.2.2 Concatenation

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

```
| ?- conc( [a, b], [c, d], A).
      1    1  Call: conc([a,b],[c,d],_31) ?
      2    2  Call: conc([b],[c,d],_64) ?
      3    3  Call: conc([],[c,d],_91) ?
      3    3  Exit: conc([],[c,d],[c,d]) ?
      2    2  Exit: conc([b],[c,d],[b,c,d]) ?
      1    1  Exit: conc([a,b],[c,d],[a,b,c,d]) ?
A = [a,b,c,d]
yes
```

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, jum, jul,
     aug, sep, oct, nov, dec]).
After = [jum,jul,aug,sep,oct,nov,dec]
Before = [jan,feb,mar,apr] ? ;
no

| ?- conc( _, [Month1,may, Month2|_], [jan, feb, mar, apr, may, jum,
     jul, aug, sep, oct, nov, dec]).
Month1 = apr
Month2 = jum ? ;
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 Concatenation

- 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.

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

```
| ?- member2(3, [1,2,3,4]).
      1    1  Call: member2(3,[1,2,3,4]) ?
      2    2  Call: conc(_58,[3|_57],[1,2,3,4]) ?
      3    3  Call: conc(_85,[3|_57],[2,3,4]) ?
      4    4  Call: conc(_112,[3|_57],[3,4]) ?
      4    4  Exit: conc([],[3,4],[3,4]) ?
      3    3  Exit: conc([2],[3,4],[2,3,4]) ?
      2    2  Exit: conc([1,2],[3,4],[1,2,3,4]) ?
      1    1  Exit: member2(3,[1,2,3,4]) ?
true ?
(15 ms) yes
```

  - 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).
```

3.2.3 Adding an item

- To add an item to a list, it is easiest to put the new item in front of the list so that it become the new head.
- If X is the new item and the list to which X is added is L then the resulting list is simply:
  [X|L].
- So we actually need no procedure for adding a new element in front of the list.
- If we want to define such a procedure:
  add(X, L,[X|L]).

```
| ?- add(4, [1,2,3],Y).
      1    1  Call: add(4,[1,2,3],_29) ?
      1    1  Exit: add(4,[1,2,3],[4,1,2,3]) ?
Y = [4,1,2,3]
Yes
```

3.2.4 Deleting an item

- Deleting an item X form a list L can be programmed as a relation:

  del(X, L, L1)

  where L1 is equal to the list L with the item X removed.

- Two cases of delete relation:
  (1) If X is the head of the list then the result after the deletion is the tail of the list.
  (2) If X is in the tail then it is deleted from there.

```
del( X, [X| Tail], Tail).
del( X, [Y| Tail], [Y|Tail1]) :- del( X, Tail, Tail1).
```

3.2.4 Deleting an item

```
del( X, [X| Tail], Tail).
del( X, [Y| Tail], [Y|Tail1]) :- del( X, Tail, Tail1).
```

```
| ?- del(4, [1,2,3,4,5,6],Y).
      1    1  Call: del(4,[1,2,3,4,5,6],_35) ?
      2    2  Call: del(4,[2,3,4,5,6],_68) ?
      3    3  Call: del(4,[3,4,5,6],_95) ?
      4    4  Call: del(4,[4,5,6],_122) ?
      4    4  Exit: del(4,[4,5,6],[5,6]) ?
      3    3  Exit: del(4,[3,4,5,6],[3,5,6]) ?
      2    2  Exit: del(4,[2,3,4,5,6],[2,3,5,6]) ?
      1    1  Exit: del(4,[1,2,3,4,5,6],[1,2,3,5,6]) ?
Y = [1,2,3,5,6] ?
(31 ms) yes
```

3.2.4 Deleting an item

- Like **member, del** is also non-deterministic.

```
| ?- del(a,[a,b,a,a],L).
L = [b,a,a] ? ;
L = [a,b,a] ? ;
L = [a,b,a] ? ;
(47 ms) no
```

- **del** can also be used in the inverse direction, to add an item to a list by inserting the new item anywhere in the list.

```
| ?- del( a, L, [1,2,3]).
L = [a,1,2,3] ? ;
L = [1,a,2,3] ? ;
L = [1,2,a,3] ? ;
L = [1,2,3,a] ? ;
(16 ms) no
```

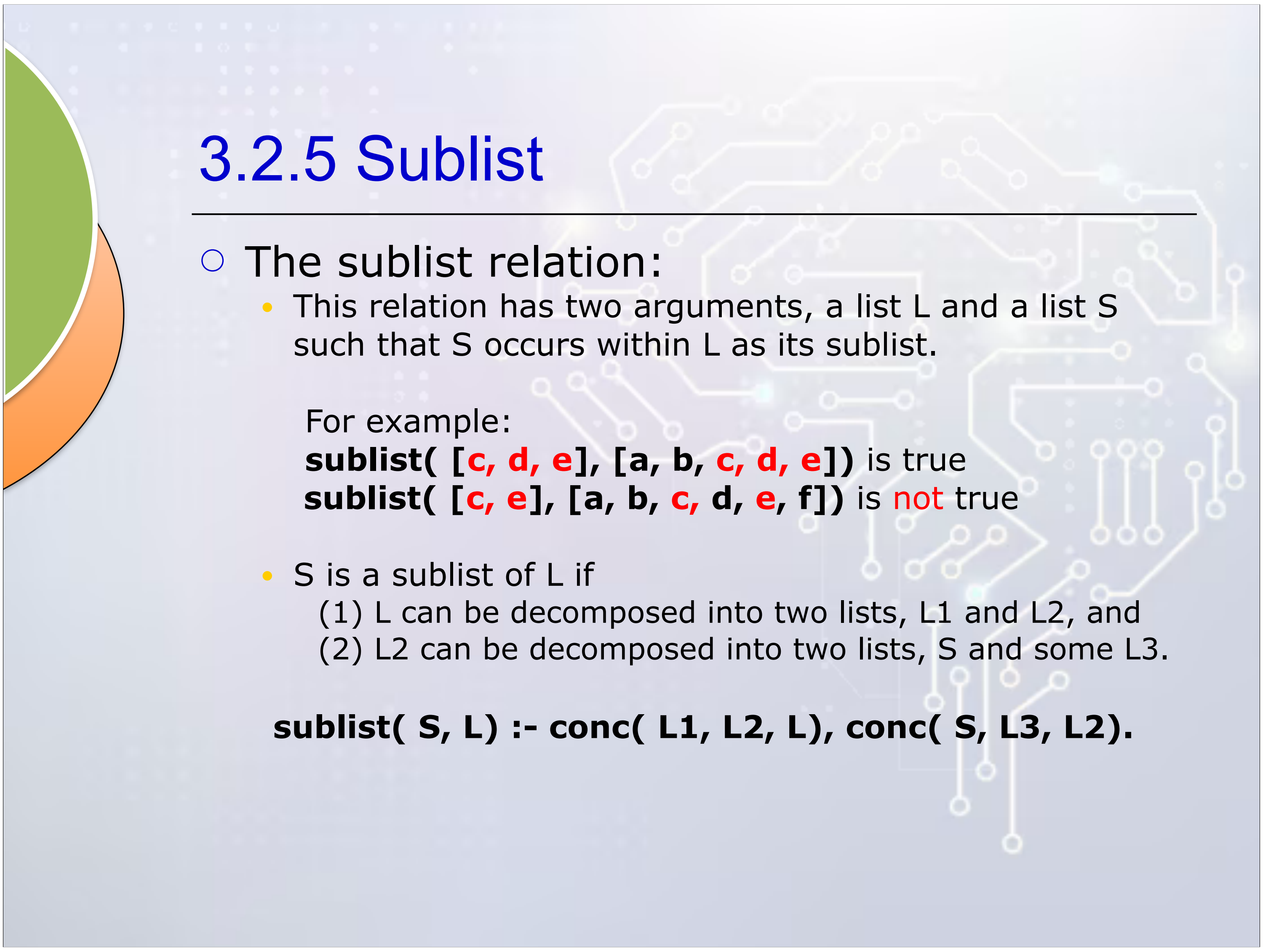
3.2.4 Deleting an item

- Two applications:
  - Inserting X at any place in some list **List** giving **BiggerList** can be defined:

```
insert( X, List, BiggerList) :-
              del( X, BiggerList, List).
```

  - Use **del** to test for membership:

```
member2( X, List) :- del( X, List, _).
```



3.2.5 Sublist

- The sublist relation:
  - This relation has two arguments, a list L and a list S such that S occurs within L as its sublist.

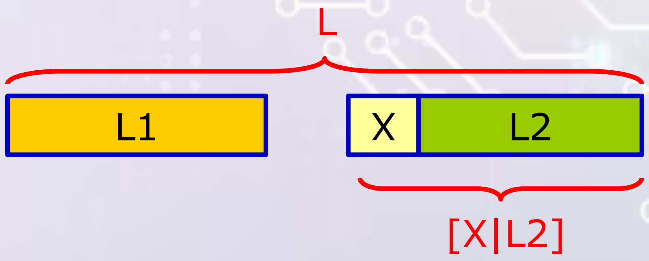
    For example:
    sublist([c, d, e], [a, b, c, d, e]) is true
    sublist([c, e], [a, b, c, d, e, f]) is not true

  - S is a sublist of L if
    (1) L can be decomposed into two lists, L1 and L2, and
    (2) L2 can be decomposed into two lists, S and some L3.

sublist(S, L) :- conc(L1, L2, L), conc(S, L3, L2).

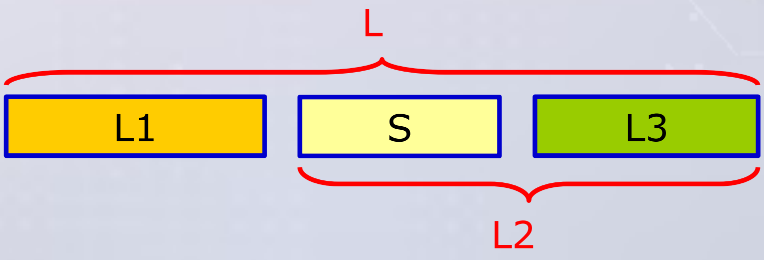
3.2.5 Sublist

- Compare to **member** relation:
  member2(X, L):- conc(L1,[X|L2],L).



member(X, L)

sublist(S, L) :- conc(L1, L2, L), conc(S, L3, L2).



sublist(S, L)

3.2.5 Sublist

- An example:

```
| ?- sublist(S, [a,b,c]).

S = [a,b,c] ? ;
S = [b,c] ? ;
S = [c] ? ;
S = [] ? ;
S = [b] ? ;
S = [a,c] ? ;
S = [a] ? ;
S = [a,b] ? ;
(31 ms) no
```

The power set of [a, b, c]

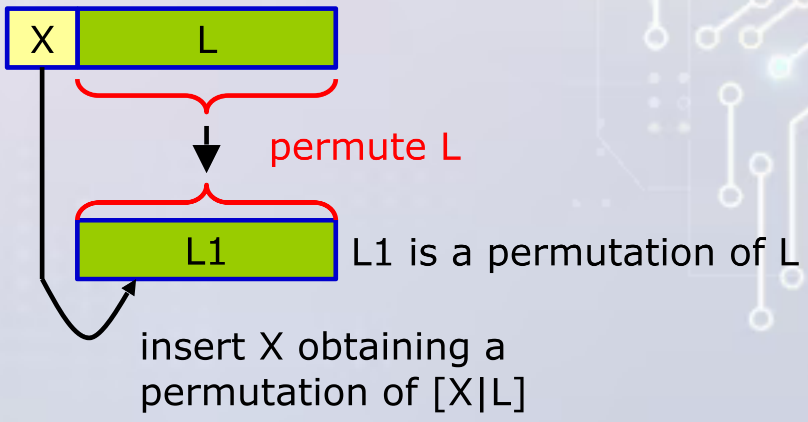
3.2.6 Permutations

- An permutation example:

```
| ?- permutation( [a, b, c], P).

P = [a,b,c] ? ;
P = [a,c,b] ? ;
P = [b,a,c] ? ;
P = [b,c,a] ? ;
P = [c,a,b] ? ;
P = [c,b,a] ? ;
(31 ms) no
```

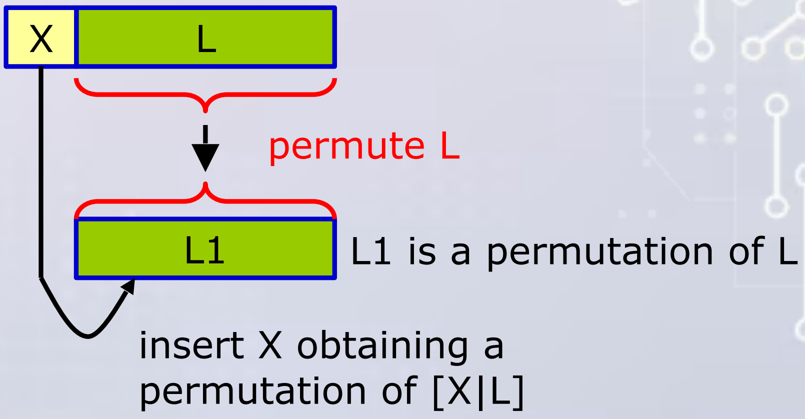
3.2.6 Permutations

- Two cases of permutation relation:
  - If the first list is empty then the second list must also be empty.
  - If the first list is not empty then it has the form [X|L], and a permutation of such a list can be constructed as shown in Fig: first permute L obtaining L1 and then insert X at any position into L1.



3.2.6 Permutations

```
permutation1([],[]).
permutation1([ X| L], P):-
        permutation1( L, L1), insert( X, L1, P).

insert( X, List, BiggerList) :-
                del( X, BiggerList, List).
```



3.2.6 Permutations

- Another definition of permutation relation:

```
permutation2([],[]).
permutation2(L, [ X| P]):-
        del( X, L, L1), permutation2(L1, P).
```

  - To delete an element X from the first list, permute the rest of it obtaining a list P, and add X in front of P.

3.2.6 Permutations

- Examples:

```
| ?- permutation2([red,blue,green], P).
P = [red,blue,green] ? ;
P = [red,green,blue] ? ;
P = [blue,red,green] ? ;
P = [blue,green,red] ? ;
P = [green,red,blue] ? ;
P = [green,blue,red] ? ;
no

| ?- permutation( L, [a, b, c]).
```

(1) Apply **permutation1**: The program will instantiate L successfully to all six permutations, and then get into an infinite loop.

(2) Apply **permutation2**: The program will find only the first permutation and then get into an infinite loop.

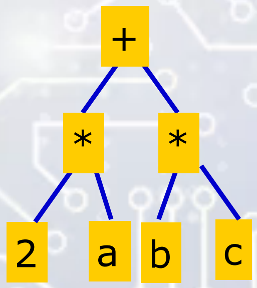
3.3 Operator notation

- In particular, + and * are said to be infix operators because they appear between the two arguments.

  2*a+b*c

- Such expressions can be represented as trees, and can be written as Prolog terms with + and * as functors:

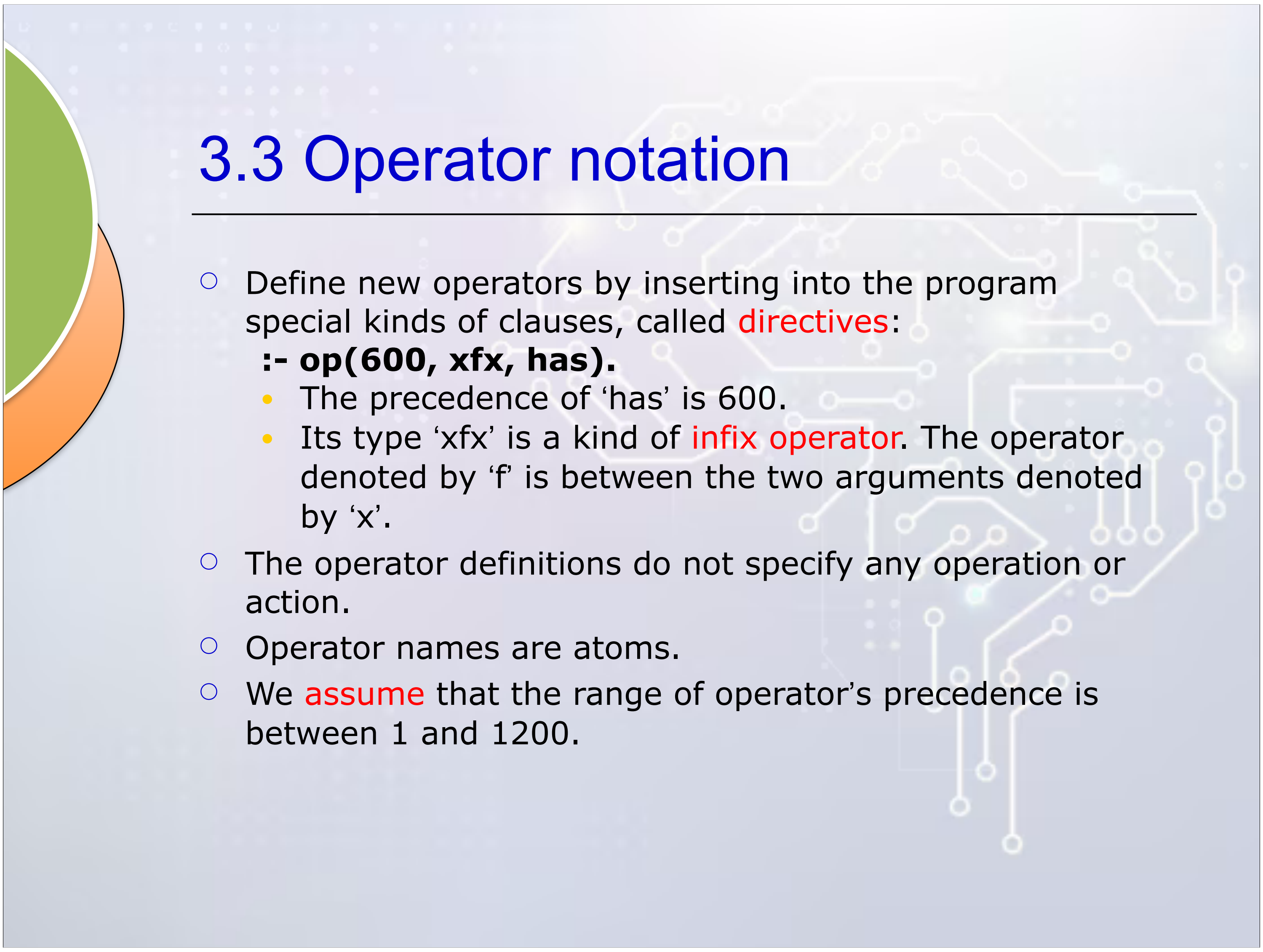
  +(*(2,a),*(b,c))



- The general rule is that the operator with the highest precedence is the principal functor of the term.
  - If '+' has a higher precedence than '*', then the expression **a+ b*c** means the same as **a + (b*c).** **(+(a, *(b,c)))**
  - If '*' has a higher precedence than '+', then the expression **a+ b*c** means the same as **(a + b)*c.** **(*(+(a,b),c))**

3.3 Operator notation

- A programmer can define his or her own operators.
- For example:
  - We can define the atoms **has** and **supports** as infix operators and then write in the program facts like:
    peter has information.
    floor supports table.
  - The facts are exactly equivalent to:
    has(peter, information).
    supports(floor, table).



3.3 Operator notation

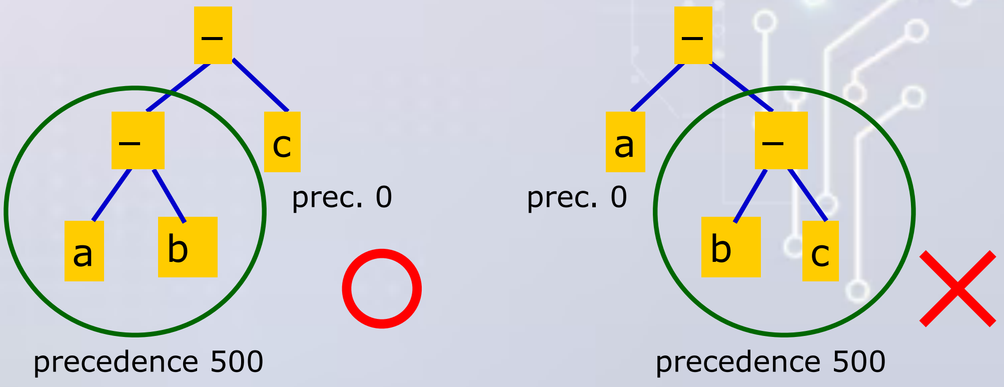
- Define new operators by inserting into the program special kinds of clauses, called directives:
  :- op(600, xfx, has).
  - The precedence of 'has' is 600.
  - Its type 'xfx' is a kind of infix operator. The operator denoted by 'f' is between the two arguments denoted by 'x'.
- The operator definitions do not specify any operation or action.
- Operator names are atoms.
- We assume that the range of operator's precedence is between 1 and 1200.

3.3 Operator notation

- There are three groups of operator types:
  (1) **Infix** operators of three types:
  xfx xfy yfx
  (2) **Prefix** operators of two types:
  fx fy
  (3) **postfix** operators of two types:
  xf yf
- Precedence of argument:
  - If an argument is enclosed in parentheses or it is an unstructured object then its precedence is 0.
  - If an argument is a structure then its precedence is equal to the precedence of its principal functor.
  - 'x' represents an argument whose precedence must be strictly lower than that of the operator.
  - 'y' represents an argument whose precedence is lower or equal to that of the operator.

3.3 Operator notation

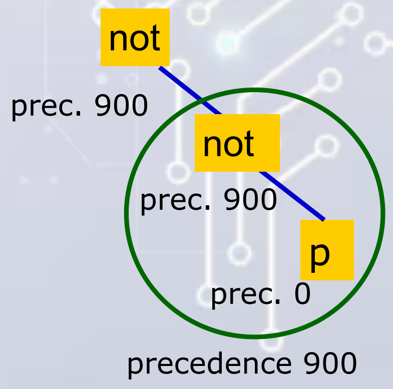
- Precedence of argument:
  - This rules help to disambiguate expressions with several operators of the same precedence.
  - For example: **a – b – c** is **(a – b) – c** not **a –(b – c)**
  - The operator '–' is defined as **yfx**.
    - Assume that '–' has precedence 500. If '–' is of type **yfx**, then the right interpretation is invalid because the precedence of **b-c** is not less than the precedence of '–'.



3.3 Operator notation

- Another example: operator **not**
  - If **not** is defined as **fy** then the expression
    not not p
    is legal.
  - If **not** is defined as **fx** then the expression
    not not p
    is illegal, because the argument to the first **not** is **not p**.
    ➔ here **not (not p)** is legal.

```
:- op( 900, fy, not).
| ?- X = not(not(P)).
X = (not not P)
Yes
:- op( 900, fx, not).
| ?- X = not(not(P)).
X = (not (not P))
Yes
```



3.3 Operator notation

- A set of predefined operators in the Prolog standard.

```
:- op( 1200, xfx, [:-, -->]).
:- op( 1200, fx, [:-, ?-]).
:- op( 1050, xfy, ->).
:- op( 900, fy, not).
:- op( 700, xfx, [=, \=, ==, \==, =..]).
:- op( 700, xfx, [is, =:=, =\=, <, =<, >, >=, @<, @=<,
                  @>, @>=]).
:- op( 500, yfx, [+, -]).
:- op( 400, yfx, [*, /, //, mod]).
:- op( 200, xfx, **).
:- op( 200, xfy, ^).
:- op( 200, fy, -).
```

3.3 Operator notation

- An example: Boolean expressions

  de Morgan's theorem:

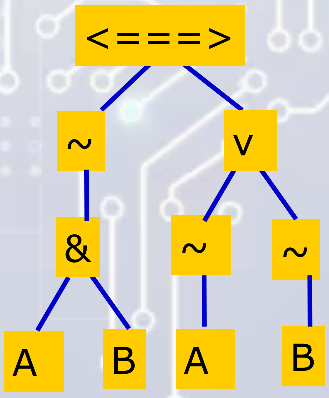
  ~(A & B) <===> ~A v ~B

  - One way to state this in Prolog is

    equivalence(not(and(A, B)), or(not(A), not(B))).
  - If we define a suitable set of operators:

    ```
    :- op( 800, xfx, <===>).
    :- op( 700, xfy, v).
    :- op( 600, xfy, &).
    :- op( 500, fy, ~).
    ```
  - Then the de Morgan's theorem can be written as the fact.

    ~(A & B) <===> ~A v ~B



3.3 Operator notation

- Summarize:
  - Operators can be infix, prefix, or postfix.
  - Operator definitions do not define any action, they only introduce new notation.
  - A programmer can define his or her own operators. Each operator is defined by its name, precedence, and type.
  - The precedence is an integer within some range, usually between 1 and 1200.
  - The operator with the highest precedence in the expression is the principal functor of the expression.
  - Operators with lowest precedence bind strongest.
  - The type of an operator depends on two things:
    - The position of the operator with respect to the arguments
    - The precedence of the arguments compared to the precedence of the operator itself.
    - For example: **xfy**

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

- Another example:

```
| ?- X is 5/2,
     Y is 5//2,
     Z is 5 mod 2.
X = 2.5
Y = 2
Z = 1
```

- Since X is 5-2-1 $\boxed{?}$ X is (5-2)-1, parentheses can be used to indicate different associations. For example, X is 5-(2-1).
- Prolog implementations usually also provide standard functions such as sin(X), cos(X), atan(X), log(X), exp(X), etc.

```
| ?- X is sin(3).
X = 0.14112000805986721
```

- Example:

```
| ?- 277*37 > 10000.
yes
```

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

- GCD (greatest common divisor) problem:
  - Given two positive integers, X and Y, their greatest common divisor, D, can be found according to three cases:
    (1) If X and Y are equal then D is equal to X.
    (2) If X < Y then D is equal to the greatest common divisor of X and the difference Y-X.
    (3) If Y < X then do the same as in case (2) with X and Y interchanged.
  - The three rules are then expressed as three clauses:

```
gcd( X, X, X).
gcd( X, Y, D) :- X<Y, Y1 is Y-X, gcd( X, Y1, D).
gcd( X, Y, D) :- Y<X, gcd( Y, X, D).
?- gcd( 20, 25, D)
D=5.
```

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:

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

3.4 Arithmetic

- Another programs:

```
length1( [], 0).
length1( [_ | Tail], N) :- length1( Tail, N1),
                           N = 1 + N1.

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

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

| ?- length2([a,b,c],N), Length is N.
Length = 2
N = 1+(1+(1+0))
```

3.4 Arithmetic

- Summarize:
  - Build-in procedures can be used for doing arithmetic.
  - Arithmetic operations have to be explicitly requested by the built-in procedure **is**.
  - There are build-in procedures associated with the predefined operators +, -, *, /, **div** and **mod**.
  - At the time that evaluation is carried out, all arguments must be already instantiated to numbers.
  - The values of arithmetic expressions can be compared by operators such as <, =<, etc. These operators force the evaluation of their arguments.

THANK YOU