



مجموعة جامعة
الجميع الذكية
للتعليم القابضة



Master of Science in Artificial intelligent

Lecture 10

AI511

Foundations of Artificial Intelligence

Prepared by
Prof. Lamiaa Fattouh Ibrahim

www.midocean.ae

Syntax and Meaning of Prolog Programs

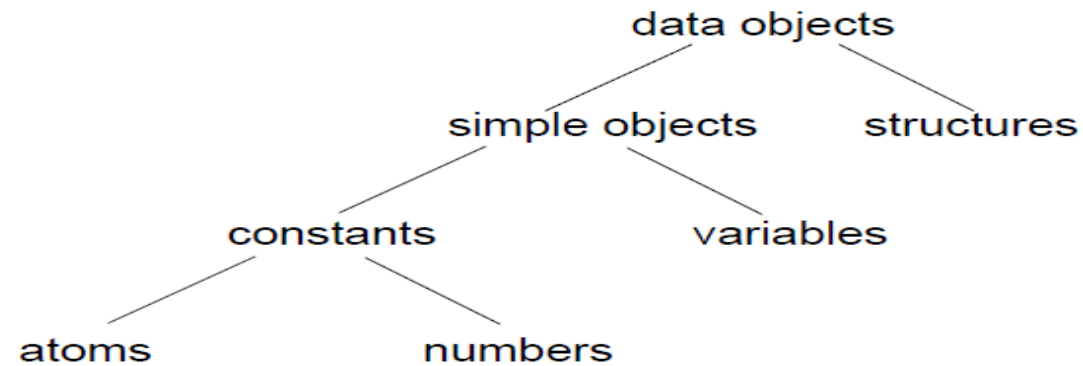
- simple data objects (atoms, numbers, variables)
- structured objects
- the matching operation on objects
- declarative meaning of programs
- procedural meaning of programs

Lists

- The Syntax of lists
- Write procedures involving lists
- Built in operators for arithmetic and comparison
- Simple debugging

Data Objects

A classification of Prolog data objects:



The Prolog syntax specifies different forms for each type of data object

~> automatic recognition of object type

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_25, x_25AB, miss_Jones

- Strings of *special characters*:

<--->, ===>, ...

Some such atoms have already a predefined meaning, e.g., ':-'.

- Strings of characters in *single quotes*:

'Tom', 'South_America', 'Sarah Jones'

Numbers are *integers* (1, 1313, 0, -97) or *reals* (3.14, -0.034, 11.2).

Variables

Variables are strings of letters, digits, or underscores. They start with an *uppercase letter* or an *underscore*. E.g.,

`X, Result, Object2, Participant_list, ShoppingList, _x23`

A variable *appearing only once* in a clause can be replaced by an underscore. E.g.,

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

is equivalent to

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

If we are *not interested in bindings* to a variable in a query, we can also use an underscore. E.g.,

`?- parent(X,_).`

is

equivalent to `?- hasachild(X).`

The *lexical scope* of variable names is *one clause*.

Structures

Structured objects (= *structures*) are objects that have several components. The components can be structures, in turn.

Example: “1 January, 2001” (a precise date, no variables):

$\underbrace{\text{date}}_{\text{functor}} \underbrace{(\text{1, january, 2001})}_{\text{arguments}}$

Example: “Any day in January” (using a variable):

`date(Day, january, 2001)`

Structures can be pictured as trees.

Example: Simple Geometric Objects

We consider the following types of *objects in the plane*:

Point: defined by *two coordinates*

Line segment: defined by *two points*

Triangle: defined by *three points*

We represent them using *three different functors*:

point for points

seg for line segments

triangle for triangles

What is the *intuitive meaning* of the following structures?

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))

... and what is the tree representation?

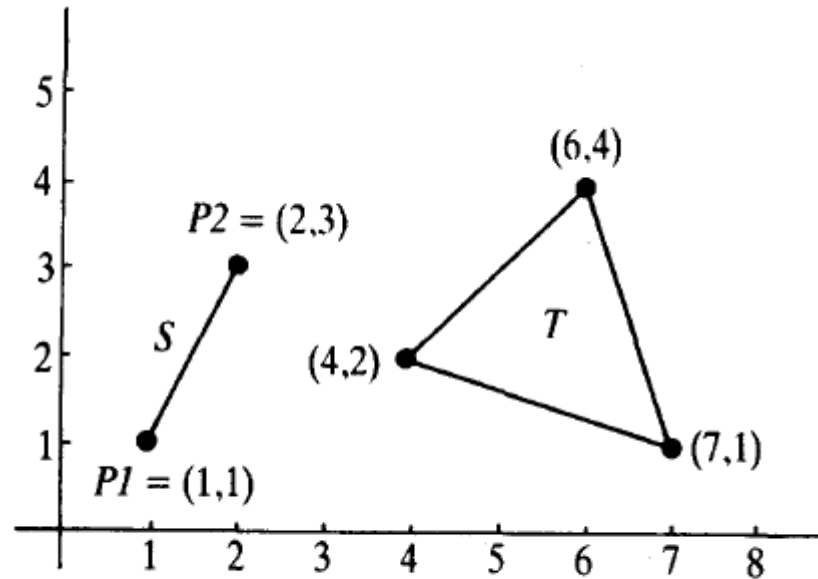


Figure 2.3 Some simple geometric objects.

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))

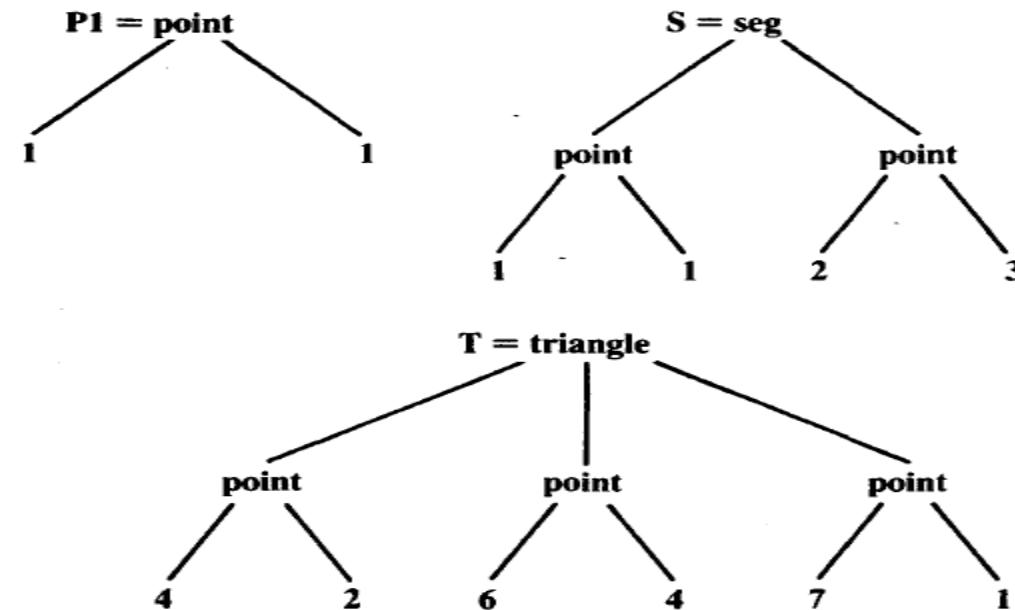


Figure 2.4 Tree representation of the objects in Figure 2.3.

Terms can be used to represent complex data objects.

The most important operation on terms is **matching**. Two terms *match* if

1. they are *identical*, or
2. the *variables* 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*.

Example: `date(D, M, 1983)` and `date(D1, may, Y1)`

Matching instantiation:

D	=
M	=
Y1	=

Most General Matchings

Prolog can be requested to match expressions by query using “=”:

```
?- date(D, M, 1983) = date(D1, may, Y1).
```

Prolog returns the *most general matching* $D = D1$, $M = \text{may}$, $Y1 = 1983$.

During a *computation*, instantiations become *more specific*:

```
?- date(D, M, 1983) = date(D1, may, Y1),  
   date(D, M, 1983) = date(15, M, Y).
```

The *first goal* is satisfied by

```
D = D1, M = may, Y1 = 1983.
```

When the *second goal* is satisfied, too, the instantiation becomes

```
D = 15, D1 = 15, M = may, Y1 = 1983, Y = 1983.
```

The Rules for Matching

Whether two *terms* (= structures) match, is determined by the following rules:

1. If S and T are *constants*, then S and T match only if they are the *same* object.
2. If S is a *variable* and T is anything, then they match, and S is instantiated to T . Conversely, if T is a *variable*, then T is instantiated to S .
3. If S and T are *structures* then they match only if
 - (a) S and T have the same principal functor, and
 - (b) their corresponding components match.

The resulting instantiation is determined by the matching of the components.

Matching: Example

Do these two terms/structures match?

```
triangle( point(1,1), A, point(2,3))
```

and

```
triangle( X, point(4, Y), point(2,Z))?
```

What is the most general matching?

Visualize the terms/structures as trees!

Matching: Example

The terms $\text{date}(D, M, 1983)$ and $\text{date}(D1, M1, 1444)$ do not match

The terms $\text{date}(X, Y, Z)$ and $\text{point}(X, Y, Z)$ do not match

We define *horizontal* and *vertical segments*:

```
vertical( seg(point(X, Y), point(X, Y1))).
```

```
horizontal( seg(point(X, Y), point(X1, Y))).
```

We query this program:

```
?- vertical( seg(point(1, 1), point(1, 2))).
```

```
yes
```

```
?- vertical( seg(point(1, 1), point(2, Y))).
```

```
no
```

```
?- horizontal( seg(point(1, 1), point(2, Y))).
```

```
Y = 1
```

The Meaning of Prolog Programs

`p :- q, r.`

Prolog programs can be understood **declaratively**, e.g.:

“p is true if q and r are true.”

“From q and r follows p.”

and **procedurally**, e.g.:

“To solve/satisfy p, **first** solve q, and **then** solve/satisfy r.”

The procedural reading not only defines 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.

Procedural Meaning

The procedural meaning of a program is a

*Procedure for executing a list of goals wrt a given program
(= try to satisfy them)*

Prolog tries to **satisfy a list of goals** by

- satisfying the *leftmost goal first*
- using the *rules in the order* in which they occur in the *program*

Lists

Lists are another commonly used data structure in Prolog.

Lists are represented as

```
[clare, sean, richard, paula]
```

The first item in a list is known as the *head* of the list.

The remaining part is the *tail*.

Note tail is a list whereas head is an element of a list.

In the above list `clare` is the head and

```
[sean, richard, paula]
```

 is the tail.

We can also represent the list as

```
[clare|[sean, richard, paula]]
```

which would match with `[Head|Tail]`.

```
member(H, [H|Tail]).
```

```
member(X, [H|Tail]):-  
    member(X, Tail).
```

The query

```
| ?- member(richard, [clare, sean, richard, paula])
```

doesn't match with the first clause but does with the
second, $X=richard$, $H=clare$, and
 $Tail=[sean, richard, paula]$ to create the new subgoal
`member(richard, [sean, richard, paula]).`

```
member(H, [H|Tail]).
```

```
member(X, [H|Tail]):-  
    member(X, Tail).
```

The subgoal

```
member(richard, [sean, richard, paula]).
```

doesn't match with the first clause but does with the second
so that the new subgoal is

```
member(richard, [richard, paula]).
```

Now this matches with the first clause so we obtain yes as
the result.

- The rules for member are

```
member (M, [M | _]) .  
Member (M, [_ | T]) :- member (M, T) .
```

The first rule says that M is a member of a list with head M. The second rule says that M is a member of a list if M is a member of its tail T.

Operations on Lists–member failing

The query

```
| ?- member(john, [clare, sean, richard, paula]).
```

keeps recursively calling member as follows.

```
member(john, [sean, richard, paula]).
```

```
member(john, [richard, paula]).
```

```
member(john, [paula]).
```

```
member(john, []).
```

There is no rule to apply to the empty list so

`member(john, [])` fails and we get the answer no.

Debugging Prolog Programmes

- The debugging tool called *tracing* is invoked by typing `trace.` at the prompt. Several options but pressing return allows you to see the program trace.
- Test smaller units eg individual procedures.
- Look out for infinite recursions.
- To stop tracing use the command `notrace.`

The Family Program

```
?- trace.
true.

[trace] ?- mother(x, bob).
  Call: (10) mother(x, bob) ? creep
  Call: (11) parent(x, bob) ? creep
  Fail: (11) parent(x, bob) ? creep
  Fail: (10) mother(x, bob) ? creep
false.

[trace] ?- mother(X, bob).
  Call: (10) mother(_17960, bob) ? creep
  Call: (11) parent(_17960, bob) ? creep
  Exit: (11) parent(pam, bob) ? creep
  Call: (11) female(pam) ? creep
  Exit: (11) female(pam) ? creep
  Exit: (10) mother(pam, bob) ? creep
X = pam ;
  Redo: (11) parent(_17960, bob) ? creep
  Exit: (11) parent(tom, bob) ? creep
  Call: (11) female(tom) ? creep
  Fail: (11) female(tom) ? Unknown option (h for help)
  Fail: (11) female(tom) ? creep
  Fail: (10) mother(_17960, bob) ? creep
false.

[trace] ?- notrace.
true.

[debug] ?- mother(X, bob).
X = pam ;
false.

[debug] ?- ■
```

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

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

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

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

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

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

Operations on Lists: Concatenation

We want to define the predicate

`conc(L1, L2, L3)`

that is true if L3 is the concatenation of L1 and L2. There are **two cases**:

1. the concatenation of the **empty list** and L2, yields L2
2. the concatenation of a **non-empty list** `[X | L1]` and L2, yields a list whose head is X and whose tail is the concatenation of L1 and L2.

In Prolog:

`conc([], L, L).`

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

In words,

The result of conc the empty list `[]` and a list L is L.

If the result of conc L and L2 is L3, then

the result of conc `[X | L]` and L2 is `[X | L3]`

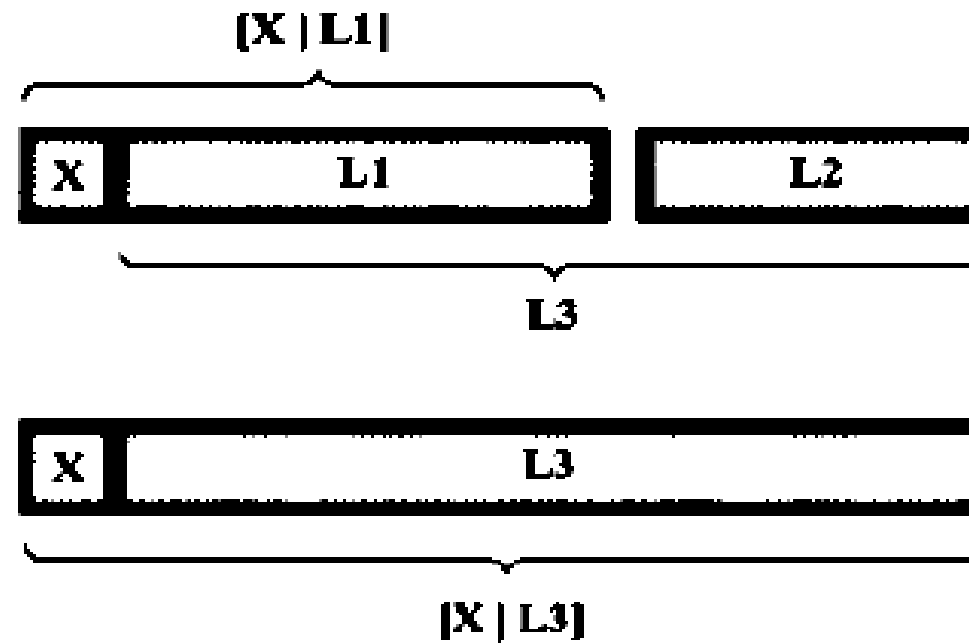


Figure 3.2 Concatenation of lists.

```

conc( [a,b,c], [1,2,3], L).
> L = [a,b,c,1,2,3]
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
conc( Before, [4|After], [1,2,3,4,5,6,7]).
> Before = [1,2,3]
    After = [5,6,7]
conc( _, [Pred, 4, Succ | _], [1,2,3,4,5,6,7]).
> Pred = 3
    Succ = 5

```

◆ Redefining member using conc:

- `member1(X, L) :-
 conc(_, [X|_], L) .`

◆ Adding an Item in the front:

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

◆ Deleting an item

- `del(X, [X|Tail], Tail) .
del(X, [Y|Tail], [Y|Tail1]) :-
 del(X, Tail, Tail1) .`
- If there are several occurrences of X in the list then del will be able to delete anyone of them.
- To insert an item at any place in the list:
`del(a, L, [1,2,3]) .`

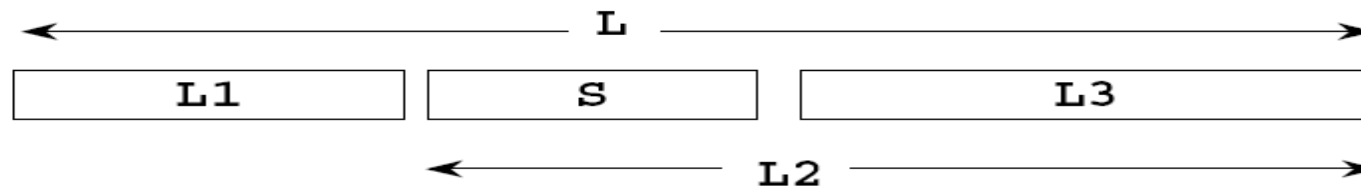
```
> L = [a,1,2,3];
> L = [1,a,2,3];
> L = [1,2,a,3];
> L = [1,2,3,a];
> no
```

◆ We can define insert using del:

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

◆ The sublist relation

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



- `sublist(S, [a,b,c]).`
`> S = [];`
`> S = [a];`
`...`
`> S = [b,c];`
`...`

◆ Permutations:

- `permutation([], []).`
`permutation([X|L], P) :-`
`permutation(L, L1),`
`insert(X, L1, P).`
- `permutation([a,b,c], P).`
`> P = [a,b,c];`
`> P = [a,c,b];`
`> P = [b,a,c];`
`...`
- `permutation2([], []).`
`permutation2(L, [X|P]) :-`
`del(X, L, L1),`
`permutation2(L1, P).`

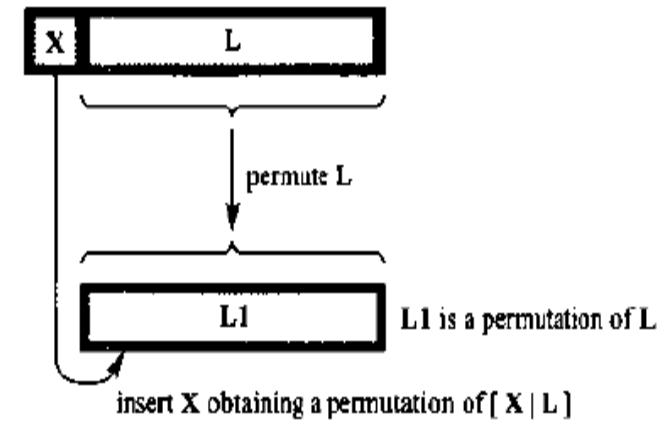


Figure 3.5 One way of constructing a permutation of the list $[X|L]$.

Length

- ◆ The length of a list can be calculated in the following way:
 - if the list is empty then its length is 0.
 - if the list is not empty then `List = [Head | Tail]`. In this case the length is equal to 1 plus the length of the tail `Tail`.
- ◆ `length` is built in. If you want to try defining it, change the name...
 - ```
length([], 0).
length([_ | Tail], N) :-
 length(Tail, N1),
 N is 1 + N1.
```
  - ```
length([a,b,[c,d],e], N).
> N = 4
length(L,4).
> [_5, _10, _15, _20] ;
..... ?
```

what happens if the order of these clauses is changed?

- **length/2**

length(L, N) returns the length **N**, i.e. the number of elements, in the list **L**.

?- **L = [a, b, c, d, e], length(L, N).**

N = 5

- **remove/3**

remove(X, L, RemainderList) removes **X** from **L** to leave **RemainderList**.

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

X = [a, c]

- **remove/3** can be used to add an element to a list.

?- **remove(nice, L, [have, a day]).**

L = [nice, have, a, day];

L = [have, nice, a, day];

L = [have, a, nice, day];

L = [have, a, day, nice]

- reverse/2

reverse(L, RevL) returns **RevL** in which the order of elements in **L** is reversed.

?- **reverse([apple, seasplash, holly, leaf], RevL).**

RevL = [leaf, holly, seasplash, apple]

- sort/2

sort(List, SortedList) returns **SortedList** as the sorted elements of **List** and removes duplicates.

For atoms it is just alphabetical.

For numbers it is just increasing.

?- **sort([apple, seasplash, holly, leaf], S).**

S = [apple, holly, leaf, seasplash]

?- **sort([3.45, 10, 4.75, 10, 1], S).**

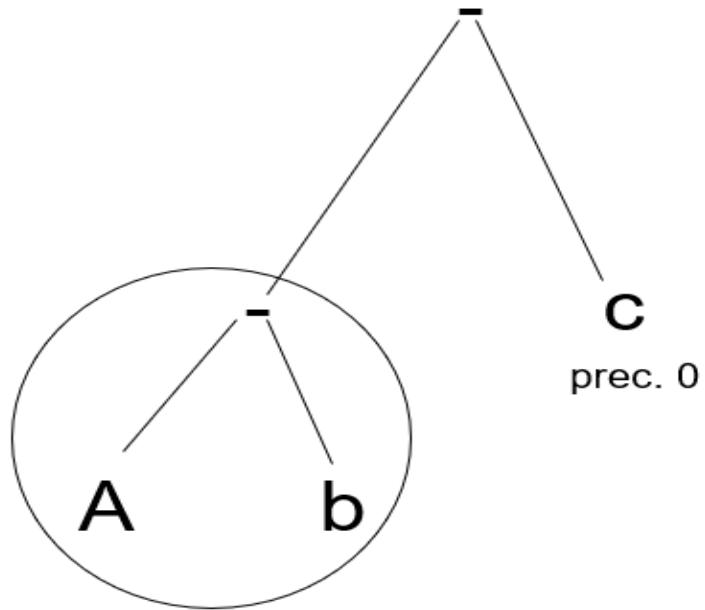
S = [1, 3.45, 4.75, 10]

Operator Notation

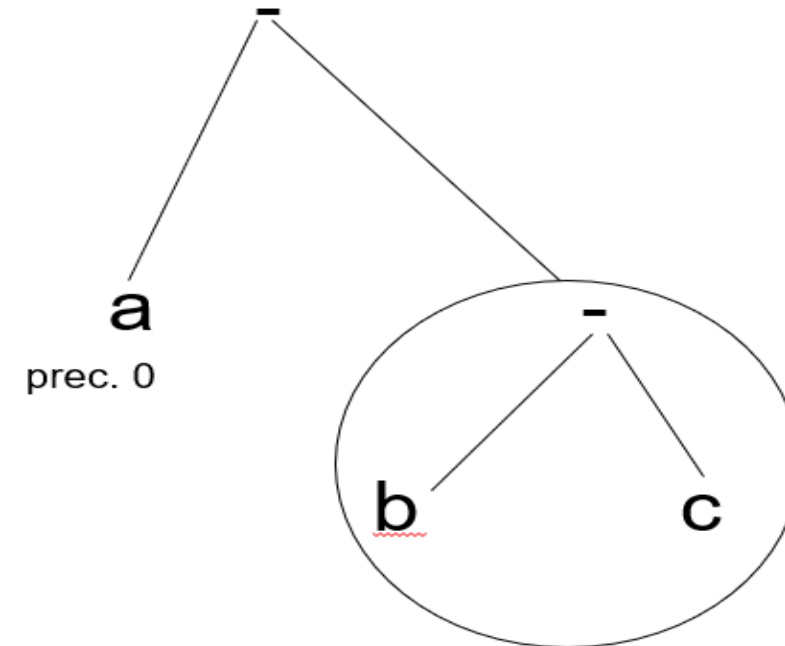
- ◆ Defining a relation as an operator:
 - `:- op( precedence, type, name)`
 - This is a special kind of clause called a *directive*.
 - `name` is an atom.
 - `precedence` is an integer. The range is implementation dependent. In your programs you should use the range 1-999. The operator with lowest precedence binds the strongest.
 - There are three groups of types:
 - `xfx xfy yfx` - infix operators
 - `fx fy` - prefix operators
 - `xf yf` - postfix operators
 - `f` represents the operator, `x` & `y` represent arguments
- ◆ Example:
 - `:- op(600, xfx, wants) .` (in program or at prompt)
 - `tim wants water.`

Precedence of Arguments

- ◆ The precedence of an atom is 0. The precedence of a structure is that 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 must be lower or equal to that of the operator.
- ◆ Examples:
 - $a - b - c$ Is normally understood as $(a - b) - c$. $\Rightarrow$ the operator '-' has to be defined as $y \text{ f } x$.
 - If the operator not is defined as fy then the expression $\text{not not } p$ is legal. If it is defined as fx it is illegal, and will have to be written as: $\text{not (not } p)$.
- ◆ The readability of programs can be often improved by using the operator notation.
- ◆ In principle, no operation on data is associated with an operator except in special cases.



Prec. 500



Prec 500

Two interpretations of the expression $a - b - c$ assuming that ' $-$ ' has precedent 500. if ' $-$ ' is of type yfx, then interpretation 2 is invalid because the precedence of $b - c$ is not less than the precedence of ' $-$ '

Predefined Operators in Prolog

```
:- op(1200, xfx, `:-').  
:- op(1200, fx, [:-, ?-]).  
:- op(1100, xfy, `;').  
:- op(1000, xfy, `,').  
:- op(700, xfx, [=, is, <, >, =<, >=, ==, =\=, \==, =:=]).  
:- op(500, yfx, [+,-]).  
:- op(500, fx, [+,-,not]).  
:- op(400, yfx, [*,/ ,div]).  
:- op(300, xfx, mod).
```

Arithmetic and Comparative Operators

- Position.

- infix operators: $+$, $-$, $*$, $/$.
- prefix operator: $-$.
- postfix operator: $!$.

- Precedence.

- Each operator has a precedence value associated with it.
- Precedence values are used to decide which operator is carried out first.
- In Prolog, multiplication and division have higher precedence values than addition and subtraction.

- Associativity.

- An operator is either left associative or right associative.
- In Prolog, arithmetic operations are left associative.
- Round brackets can be used to enforce precedence and associativity.

Arithmetic

Prolog has several built in operators for arithmetic.

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

Evaluating Arithmetic Operators

The query

```
| ?- X = 3 + 5
```

```
X = 3 + 5
```

does not evaluate the addition operation. However the following does.

```
| ?- X is 3 + 5
```

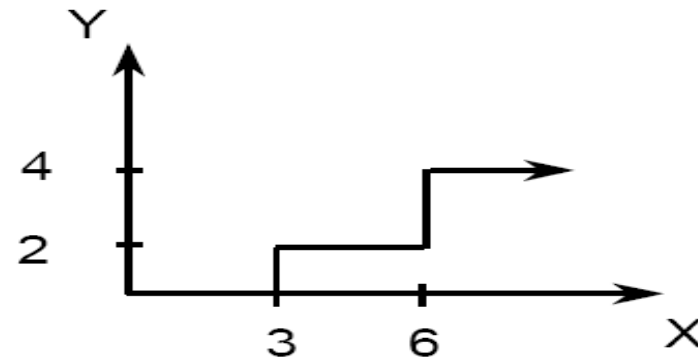
```
X = 8
```

Arithmetic and Comparative Operators

- *Arithmetic expression*: involving arithmetic operators
- A built-in operator (predicate) **is** can be used in an *evaluation goal* for evaluating an arithmetic expression (i.e., carrying out an arithmetic calculation).
 - The left hand side is usually a variable.
 - The right hand side is an arithmetical expression.
- Built-in predicates for making comparison between two arithmetic expressions in a goal.
 - $X == Y$ *the values of X and Y are equal.*
 - $X \neq Y$ *the values of X and Y are not equal.*
 - $X < Y$ *X is less than Y .*
 - $X > Y$ *X is greater than Y .*
 - $X \leq Y$ *X is less than or equal to Y .*
 - $X \geq Y$ *X is greater than or equal to Y .*

Controlling Backtracking

- ◆ Automatic backtracking can cause inefficiency:



- 1. if $X < 3$ then $Y = 0$
 2. if $3 \leq X$ and $X < 6$ then $Y = 2$
 3. if $6 \leq X$ then $Y = 4$

- ◆ The relation $f(X,Y)$ in prolog would be:
 $f(X, 0) \text{ :- } X < 3.$
 $f(X, 2) \text{ :- } 3 \leq X, X < 6.$
 $f(X, 4) \text{ :- } 6 \leq X.$
- ◆ This procedure assumes that before $f(X,Y)$ is executed X is already instantiated to a number.
- ◆ The goal: “ $f(1, Y), 2 < Y.$ ” fails, but before prolog replies ‘no’, it tries all 3 rules.
- ◆ The three rules are mutually exclusive so that one of them at most will succeed. If the goal matches the first rule and then fails, there is no point in trying the others.
- ◆ The CUT mechanism will help us prevent this.

CUT

- ◆ A cut prevents backtracking from some point on.
- ◆ Written as a '!' sub-goal that always succeeds, but prevents backtracking through it.
- ◆ Correcting the example:
 $f(X, 0) \text{ :- } X < 3, \text{ !.}$
 $f(X, 2) \text{ :- } 3 \leq X, X < 6, \text{ !.}$
 $f(X, 4) \text{ :- } 6 \leq X.$
- ◆ Whenever the goal $f(X, Y)$ is encountered, only the first rule that matches will be tried.
- ◆ If we now ask again " $f(2, Y), 2 < Y.$ " we will get the same answer, 'no', but only the first rule of 'f' will be tried,
- ◆ **note:** the declarative meaning of the procedure did not change.

Another problem:

- ◆ If we ask:
 $f(7, Y)$.
 $> Y=4$
- ◆ What happened:
 - $7 < 3 \rightarrow \text{fail}$
 - $3 = < 7, 7 < 6 \rightarrow \text{fail}$
 - $6 = < 7 \rightarrow \text{success.}$
- ◆ Another improvement: The logical rule
 - if $X < 3$ then $Y=0$,
otherwise if $X < 6$ then $Y=2$,
otherwise $Y=4$.

Is translated into:

- $f(X, 0) \quad :- \quad X < 3, \quad !.$
- $f(X, 2) \quad :- \quad X < 6, \quad !.$
- $f(X, 4) .$

- ◆ The last change improved efficiency. BUT, removing the cuts now will result in multiple answers, some of which are not correct.

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

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

- ◆ In this version the cuts do not only effect the procedural meaning of the program, but also change the declarative meaning.

Examples using CUT

◆ The maximum of two elements:

- `max(X,Y,Max) <=> Max is the maximum of X and Y.`
- `max(X, Y, X) :- X >= Y.`
`max(X, Y, Y) :- X < Y.`

A more economical solution (using “otherwise” logic):

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

◆ A single-solution membership:

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

◆ Adding elements to a list without duplication:
add(X,L,L1)

- If X is a member of L then L1=L.
Otherwise L1 is equal to L with X inserted.
- `add(X, L, L) :- member(X, L), !.
add(X, L, [X|L]).`

◆ Classification into categories.

- `beat(tom, jim).`
`beat(ann, tom).`
`beat(pat, jim).`
- winner: every player who won all his or her games.
fighter: any player that won some games and lost some.
sportsman: any player who lost all his or her games.
- `class(X, fighter) :-
beat(X, _), beat(_, X), !.
class(X, winner) :-` ← cut not necessary
`beat(X, _), !.
class(X, sportsman) :-
beat(_, X), !.`

Summary

- The list is a frequently used structure. It is either empty or consists of a head and a tail which is list as well. Prolog provides a special notation for lists.
- Common operations on lists, programmed in this lecture, are: list membership, concatenation, adding an item, deleting an item, sublist.
- The operator notation allows the programmer to tailor the syntax of programs toward particular needs. Using operators the readability of programs can be greatly improved.

Summary

- New operator are defined by the directive op, starting the name of an operator, its type and precedence.
- In principle, there is no operation associated with an operator; operators are merely a syntactic device providing an alternative syntax for terms.
- Arithmetic is done by built-in procedures. Evaluation of an arithmetic expression is forced by the procedure is and by the comparison predicates $<$, $=<$, etc.



مجموعة جامعة
الجميع الذكية
للتعليم القابضة

MIDOCEAN
UNIVERSITY
BY EVERYONE'S SMART UNIVERSITY



Thank You

