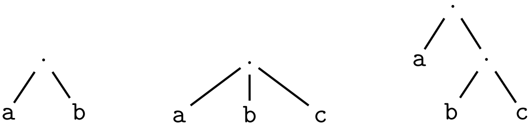
Tuples

Tuples and **lists** are a very useful data structure that aggregates objects in a specific order.

For example

```
.(a, b)       % Pair made of ‘a’ and ‘b’
.(a, b, c)    % Triple made of ‘a’, ‘b’ and ‘c’
.(a, .(b, c))
```

Tuples are common in mathematics, e.g. “Let *p* a point of coordinates (x, y) such as …” We saw page 58 that Prolog objects can be represented as trees:



Lists

Lists are similar to tuples

```
[a, b]         % List made of ‘a’ and ‘b’
[a, b, c]      % List made of ‘a’, ‘b’ and ‘c’
[a, [b, c]]
```

So, where is the difference? List have a special syntax that allows to distinguish and extract the first elements, called **head**, while the remaining elements are called the **tail**:

```
[a, b, c]      % List made of ‘a’, ‘b’ and ‘c’.
[a | [b,c]]    % Idem. Head is ‘a’ and tail is [b,c].
[a,b | [c]]    % Idem but head is [a, b] and tail is [c].
[a,b,c | []]   % Idem but head is [a, b, c] and tail is [].
```

Lists (cont)

Actually, list can be coded by means of tuples; all what is needed is a special atom for representing the empty list. Tradition notes it `[]`. Then

```
[a, b, c]                   % List made of ‘a’, ‘b’ and ‘c’
.(a, .(b, .(c, [])))        % Same
```

Remember that lists can be **heterogeneous**, i.e. elements can be of any kind (see `[a, [b, c]]` above).

Lists/Membership

Let us write a Prolog program that checks if a given object belongs to a given list. Let `member` be this binary relation.

For example, we want

```
?- member(b, [a,b,c]).        % True
?- member(b, [a,[b,c]]).      % False
?- member([b,c], [a,[b,c]]).  % True
```

This suggests the recursive definition

X is a member of a list L if either

- *X is the head of L, or*
- *X is a member of the tail of L.*

Lists/Membership (cont)

This informal definition can straightforwardly be translated to Prolog:

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

or

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

Remember that order matters for the procedural meaning!

Arithmetic

Arithmetic operators are special functors. The following are available:

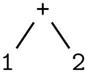
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`**`	Power
`//`	Integer division
`mod`	Modulo (remainder of integer division)

Arithmetic (cont)

If the Prolog interpreter is naively asked

```
?- X = 1 + 2.
X = 1+2
Yes
```

This because + is a functor and functors trigger no computation. Prolog builds the tree



In order to force the arithmetic interpretation, we use

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

Arithmetic (cont)

We have

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

Arithmetic/Comparison

The comparison operators force the evaluation of their argument, as `is` does:

```
?- 5 * 3 > 2.
Yes
```

Assume we have a relation `born` in a program, that relates people's name to their birth years. We can find all the persons born between 1980 and 1990 by the query

```
?- born(Name, Year),
   Year >= 1980,
   Year =< 1990.
```

Arithmetic/Comparison (cont)

The comparison operators are

`X > Y`	`X` is greater than `Y`
`X < Y`	`X` is smaller than `Y`
`X >= Y`	`X` is greater than or equal to `Y`
`X =< Y`	`X` is smaller 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

Arithmetic/Comparison (cont)

Beware! The goals `X = Y` (matching) and `X =:= Y` (arithmetic comparison) are completely different.
Consider

```
?- 1 + 2 =:= 2 + 1.
Yes
?- 1 + 2 = 2 + 1.
No
?- 1 + A = B + 2.
A = 2
B = 1
Yes
```

Arithmetic/Example

Let us define a relation `length` which associate a list and its length.

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

Note that "`N is 1 + M`" *must* be the second goal of the body. And what if

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

Arithmetic/Example (bis)

Answer:

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

This, again, because + is just a functor. Therefore, we can equivalently write

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

Arithmetic/Example (bis)

Or even shorter

```
length([],0).
length([_ | Tail],1 + M) :- length(Tail,M).

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

?- length([a,b,[c,d],e],N), Length is N.
N = 1 + (1 + (1 + 0))
Length = 3
Yes
```