```
?- parent(X, Y).
Y = dan
X = adam ;
Y = christine
X = dan ;
Y = david
X = phil
```

(and so on)

```
?- parent(X, X).
no

?- parent(X, shula), female(X).
X = jill ;
no

?- parent(X, Y, Z).
no

?- X(adam, dan).
Syntax error
```

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

?- mother(Z, elizabeth).
Z = jill ;
no
```

```
father(X, Y) :- parent(X, Y), male(X).

?- father(X, elizabeth).
X = phil ;
no
```

```
grandparent(GP, GC) :-
   parent(GP, Int), parent(Int, GC).

?- grandparent(X, christine).
X = adam ;
no
```

```
grandfather(GF, GC) :-
   grandparent(GF, GC), male(GF).

?- grandfather(GF, kenton).
GF = dan ;
no
```

```
person(X) :- male(X).

person(X) :- female(X).
```

```
?- person(X).
X = adam ;
X = dan ;
⋮
X = kenton ;
X = doris ;
```

and so on

```
ancestor(A, D) :- parent(A, D).

ancestor(A, D) :-
    parent(A, Int1), parent(Int1, D).

ancestor(A, D) :-
    parent(A, Int1), parent(Int1, Int2),
    parent(Int2, D).
```

. . . and so on.

```
ancestor(A, D) :- parent(A, D).

ancestor(A, D) :-
    parent(A, Int), ancestor(Int, D).
```

Exercises

1. Write a query to find Phil’s sons.

2. What’s wrong with the following:

```
sibling(X, Y) :-
    father(F, X), father(F, Y),
    mother(M, X), mother(M, Y).
```

```
person(name(adam, archer), sex(male)).

person(name(dan, archer),  sex(male)).

person(name(christine, barford),
                           sex(female)).
```

```
?- person(name(X, archer), sex(male)).
X = adam ;
X = dan ;
no
```

```
person(name(first_name(adam),
            surname(archer)), sex(male)).
```

```
?- person(X, Y).
Y = sex(male)
X = name(first_name(adam), surname(archer))

?- person(name(X), sex(Z)).
no

?- person(name(X, Y), sex(Z)).
Z = male
Y = surname(archer)
X = first_name(adam)
```

Functions are not evaluated

```
sum(2 + 3).

?- sum(X + Y).
Y = 3
X = 2

?- sum(X).
X = 2 + 3

?- sum(5).
no
```

```
is_true(true).

is_true(and(X, Y)) :- is_true(X),
                      is_true(Y).

is_true(or(X, Y))  :- is_true(X).

is_true(or(X, Y))  :- is_true(Y).

is_true(not(X))    :- is_false(X).


is_false(false).

is_false(and(X, Y)) :- is_false(X).

is_false(and(X, Y)) :- is_false(Y).

is_false(or(X, Y))  :- is_false(X),
                       is_false(Y).

is_false(not(X))    :- is_true(X).
```

Functions versus relations

An n-ary function is typically an $(n + 1)$-ary relation.

E.g.

Unary function:
```
fact 0 = 1
fact (n+1) = n * fact (n-1)
```

Binary relation:
```
fact(0, 1).
fact(N, F) :- ...
```

Exceptions: functions that return Booleans.

E.g.

Unary function:
```
even n = (n mod 2) == 0
```

Unary relation:
```
even(N) :- ...
```

Nondeterminism

- Functions are $1 : 1$ or $m : 1$, so only one answer.
- Relations can be $1 : 1$, $m : 1$, $1 : m$, $m : n$, so there can be multiple answers.
- Logic programming languages, such as Prolog, can find all the answers.
- Examples

  `?- grandparent(alf, X).`
  — finds *all* grandchildren

  `?- permutation([a, b, c], Xs).`
  — finds *all* permutations

Polymodality

- In other programming languages, you need separate definitions for:
  - given a person, compute grandchild
  - given a person, compute grandparent
  - given two people, check whether the first is grandparent of second
  - compute all grandparent/ grandchild pairs

Polymodality continued

- Yet, the same relation is involved in all these cases.

- In, e.g., Prolog, we can perform all these operations on a single definition of the relation:

```
grandparent(GP, GC) :-
   parent(GP, Int), parent(Int, GC).
```

- All we need vary is the query. The variables `GP` and `GC` may be:

  - **instantiated** (mapped to constants)

  - **uninstantiated** (mapped to themselves or other variables)

  (A bit like saying which is the input argument and which the output argument.)

Polymodality continued

- The pattern of instantiation is the **mode** of use. Allowing multiple modes is (what we call) **polymodality**.

- Another two examples:

  – a definition of integer addition might allow:
  — addition, checking of addition, subtraction

  – a definition of list concatenation might allow:
  — concatenation, checking concatenation, prefix computation, suffix computation, computation of all prefix/suffix pairs.

Polymodality continued

- Benefits
  - programs more concise
  - relationships expressed once, so less danger of inconsistency
  - flexibility of use
- Costs
  - the best order of subgoals needed for efficiency can conflict with the order needed for polymodality
  - some Prolog impurities are incompatible with polymodality

A loss of polymodality

- We expect infinite computation (no matter what programming language we are using) when issuing certain queries on infinite relations, e.g.:

  `?- is_true(X).`
  — all satisfiable formulae of propositional logic

- However, *Prolog*'s evaluation strategy means that certain queries even on *finite* relations launch infinite computations. This makes certain modes unavailable.

- Prolog's evaluation is as follows:

  - try clauses in the order they appear in the database (from top to bottom)

  - try subgoals in the body of a rule from left-to-right.

A loss of polymodality continued

The following are *logically equivalent*, but the way Prolog evaluates queries on these relations means that we can get different outcomes.

1. ```
   anc(A, D) :- par(A, D).
   anc(A, D) :- par(A, I), anc(I, D).
   ```

2. ```
   anc(A, D) :- par(A, I), anc(I, D).
   anc(A, D) :- par(A, D).
   ```

3. ```
   anc(A, D) :- par(A, D).
   anc(A, D) :- anc(I, D), par(A, I).
   ```

4. ```
   anc(A, D) :- anc(I, D), par(A, I).
   anc(A, D) :- par(A, D).
   ```

A loss of polymodality continued

- Let

`++`	mean argument must be instantiated
`-`	mean argument must be uninstantiated
`?`	mean argument can be either

  then available modes are:

```
anc1(?, ?)        anc3(?, ++)
anc2(?, ?)        anc4 --- none
```

- Rules of thumb
  - do not have the recursive call as the first sub-goal
    (i.e. avoid **left-recursion**)
  - put the base case as the first clause
  - rules of thumb are made to be broken

Lists

- An exploitation of function symbols

- Recursive definition

  A list is
  either `nil`
  or `cons` applied to two arguments,
  the first being some item,
  the second being another list

- E.g.

```
favThings(julie,
          cons(raindrops,
               cons(kittens,
                    cons(string, nil)))).
```

Lists continued

- In fact, for lists, Prolog provides an in-built constant `[]` and an in-built function `.` (full stop).

- E.g.

```
favThings(julie,
              .(raindrops,
                   .(kittens,
                        .(string, []))))).
```

List examples

```
favThings(julie,
  .(raindrops, .(kittens, .(string, []))))).

favThings(dgb, []).

favThings(shakespeare, .(tioletDuck, [])).
```

What is wrong with:

```
favThings(shakespeare, toiletDuck).
```

And what is wrong with:

```
favThings(geek,
  .(plan9, .(visBasic, .(unix, starTrek)))).
```

List notation

Underlying representation   Syntactic sugar

```
[]
.(a, [])
.(a, .(b, []))
.(a, .(b, .(c, [])))
```

E.g.

```
favThings(julie,
            [raindrops, kittens, string]).
```

The bar notation

- Commas separate individual elements.
- The bar separates an element and a list.

`[Y | Ys]`	at least one
`[Y1, Y2 | Ys]`	at least two
`[Y1, Y2, Y3 | Ys]`	at least three
`[Y1 | Y2, Y3]`	syntax error
`[Y1 | [Y2, Y3]]`	same as `[Y1, Y2, Y3]`
`[Y1 | [Y2 | Y3]]`	same as `[Y1, Y2, | Y3]`
`[a | b]`	legal, but an ‘abuse’

List operations

- Definitions will reflect the recursive structure of the data.

- Example

```
code(b, h).     % b is the code for h
code(a, o).
code(j, w).
```

Define `crypto/2`, where `crypto(Xs, Ys)` is true if `Ys` is the decoding of `Xs`.

E.g.

```
?- crypto([b, a, j], Message).
Message = [h, o, w]
```

List operations continued

Define `member/2`, where `member(X, Ys)` is true if `X` is an element of list `Ys`.

E.g.

```
?- member(b, [a, b, c]).
yes
```

Polymodality — a refinement

- An argument variable may be
  - **uninstantiated** (mapped to itself or another variable)
  - **instantiated**
    - **fully instantiated** (mapped to a term containing *no* variables)
    - **partially instantiated** (mapped to a non-variable term that contains variables)

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

?- member(X, Xs).

?- member(f(X), [f(a), f(b), g(a) | Xs]).
```

Polymodality — a refinement continued

- This gives us more modes to consider.

- Let

`+`	mean argument must be (at least partially) instantiated
`++`	mean argument must be *fully* instantiated
`-`	mean argument must be uninstantiated
`?`	mean argument can be any of the above

- E.g.

```
member(?, ++)
```

(`++` is a bit of a simplification.)

Types

- Prolog is not a typed language. E.g. the following perhaps ought to give type errors, but don't:

```
?- member(a, [a | b]).
yes

?- member(b, [a | b]).
no
```

- Suppose we define `list/1`, where `list(X)` is true if `X` is a list.

```
list([]).

list([X|Xs]) :- list(Xs).
```

Types continued

- We can now add a form of type checking ourselves:

```
member(X, [X | Xs]) :- list(Xs).
member(X, [Y | Ys]) :- member(X, Ys).
```

Now,

```
?- member(a, [a | b]).
no

?- member(b, [a | b]).
no
```

- – error messages would be preferable
- – there is a run-time cost

List operations continued

Define `append/3`, where `append(Xs, Ys, Zs)` is true if `Zs` is the list formed by concatenating `Ys` to the end of `Xs`.

E.g.

```
?- append([a, b, c], [d, e], L).
L = [a, b, c, d, e]
```

List operations continued

- Notes on `append/3`:
  - note we recurse on only one argument; this is *often* the case
  - it is left-recursive
  - it describes an infinite relation
  - it is not fully polymodal

    ```
    append(++, ?, ?)     append(?, ?, ++)
    ```
  - of the modes it offers, some are very useful
  - note how `append/3`'s modes affect `reverse/2`

    ```
    reverse([], []).

    reverse([X | Xs], Ys) :-
        reverse(Xs, Zs),
        append(Zs, [X], Ys).
    ```