# Defining relations with facts

Prolog is a programming language for symbolic (non-numeric) computation. Its name means **programming in logic**.

Let us approach it as if it were a **deductive database**. First, a database is populated by **relations**. A relation is what links two or more objects. This way we learn something about their relative meaning. For example, in "Tom is the parent of Bob," is a statement, the relation is parenthood and relates Tom and Bob.

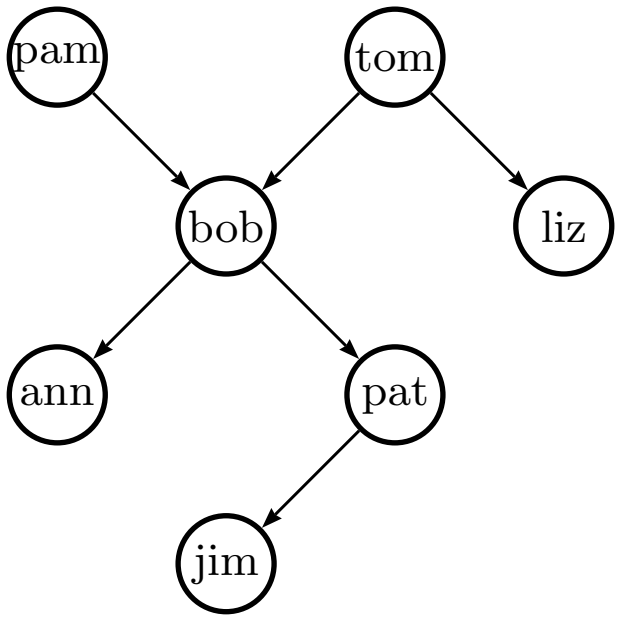
In Prolog, the simplest way of defining a relation is by enumerating **facts** (statements). For example

```
parent(tom,bob).
```

is a fact.

# Defining relations with facts (cont)

**A family tree encoded in Prolog**




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

# Queries

This program is composed of six **clauses**, each declaring a fact about the `parent` relation. As we shall see later, a clause may not be a fact.

We say that, for example, `parent(tom,bob)` is an **instance** of the `parent` relation.

In general, a relation is defined as the set of all its instances.

After loading this relation, the Prolog interpreter allows the user to ask some questions about it. For example, the query "Is Bob a parent of Pat?" is written

```
?- parent(bob,pat).
```

Having found that this a fact, the interpreter answers

```
Yes
```

# Ground queries

Another query can be

```
?- parent(liz,pat).
```

The interpreter answers

```
No
```

The same answers is given to query

```
?- parent(tom,ben).
```

because the interpreter never heard of `ben`.

A **query** starts with '`?-`', is followed by a **goal** and ended by a period.

# Existential queries

More interesting is asking "Who are the parents of Liz?" In this case what we do not know is a set of persons. The usual way of speaking about an unknown person is to name it with an arbitrary **variable**, e.g. `X`. The corresponding Prolog query is then

```
?- parent(X,liz).
```

In this case, we expect *all* the parents of Liz to be found — or none if no instance were defined. The interpreter answers

```
X = tom
Yes
```

A **ground query** is a query with no variables.

# Existential queries on one variable

If asked who are Bob's children:

```
?- parent(bob,X).
```

we expect several answers. The interpreter gives one and wait for the user to either type in a semi-colon for more answers or the return key to stop:

```
X = ann ;
X = pat
Yes
```

If we had exhausted all the solutions, the interpreter would print

```
No
```

# Existential queries on two variables

We can also query on two variables, like asking for all `X` and `Y` such that `X` is a parent of `Y`. In other words, find all the instances of the `parent` relation. This is formally written as

```
?- parent(X,Y).
```

The answer starts as

```
X = pam
Y = bob ;
X = tom
Y = bob ;
X = tom
Y = liz ;
```

# Conjunctive queries and shared variables

We could ask: "Who is the grand-parent of Jim?" But, since, we did not define the `grandparent` relation, we have to break this question in two parts:

1. Who is a parent of Jim? Assume that (s)he is named `Y`.
2. Who is a parent of `Y`? Assume that (s)he is named `X`.

The answer to the original question is `X`. Formally, this is written

```
?- parent(Y,jim), parent(X,Y).
```

The answer is (final `Yes` omitted):

```
X = bob
Y = pat
```

# Conjunctive queries and shared variables (cont)

Note that the Prolog interpreter returns the values for all the variables involved in the query (here `X` and `Y`), even if we are only interested in some of them (here `X`).

The previous query is made by tying two queries sharing a common variable, `Y`. It is a **conjunctive query**.

If we change the order of composition of the two queries, the logical meaning remains the same:

```
?- parent(X,Y), parent(Y,jim).
```

produces the same result. Similarly, we ask: "Who are Tom's grand-children?" by

```
?- parent(tom,X), parent(X,Y).
```

# Conjunctive queries and shared variables (cont)

Let us ask: "Do Ann and Pat have a common parent?" Again, the question can be broken down in two sub-questions:

1. Who is a parent `X` of Ann?
2. Is (the same) `X` a parent of Pat?

In Prolog, this conjunctive query is written

```
?- parent(X,ann), parent(X,pat).
```

The Prolog queries are usually called **goals**.

# Defining relations by rules

We can extend our example in many interesting ways. Let us first add the information about the sex of the people. This can be easily done by simply adding the following facts to our program:

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

The relations introduced are `male` and `female`. These relations only have one argument, and are called **predicates**.

# Defining relations by rules (cont)

Another way consists in defining a **binary relation** for the sex, i.e. a relation between two objects, like the `parent` relation. For example:

```
sex(pam,feminine).
sex(tom,masculine).
sex(bob,masculine).
...
```

Now let us introduce the **inverse relation** of `parent`, we call `offspring`. We can add new facts again. For example

```
offspring(liz,tom).
```

# Defining relations by rules (cont)

However, the Prolog language offers a more compact and elegant way to define the relation `offspring` in terms of the already defined relation `parent`.

Indeed, what we want to say is:

> *For all X and Y, Y is an offspring of X if X is a parent of Y.*

In Prolog this is expressed by means of a clause

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

Such clauses are called **rules**.

# Defining relations by rules (cont)

There is an important difference between facts and rules. A fact like

```
parent(tom,liz).
```

is something that is unconditionally true. On the other hand, rules specify things that are true **if** some condition is satisfied.
Rules have a *condition* part (right-part of the rule) and a *conclusion* (left-hand side). The conclusion is also called the **head** and the condition the **body**. For example

$$\underbrace{\texttt{offspring(Y,X)}}_{\text{head}} \texttt{ :- } \underbrace{\texttt{parent(X,Y)}}_{\text{body}} \texttt{.}$$

# Defining relations by rules (cont)

Let us ask now whether Liz is an offspring of Tom:

```
?- offspring(liz,tom).
```

Since there are no facts about offsprings in the program and there is a rule whose head is an instance of `offspring`, the Prolog interpreter tries to use the latter.
The head of this rule is `offspring(Y,X)`, implicitly for all `X` and `Y`. So `X` can be replaced by `tom` and `Y` by `liz`. This process is called **instantiation**. We get a special case of the rule:

```
offspring(liz,tom) :- parent(tom,liz).
```

# Defining relations by rules (cont)

Now the Prolog interpreter tries to prove the body `parent(tom,liz)`. This is actually a fact, so it answers `Yes`.

It does not give any variable information, like `X = tom`, because the original goal contained no variable.

Let us add more twists to our program by considering a `mother` relation. It could be informally defined by

> *For all `X` and `Y`, `X` is the mother of `Y` if `X` is a parent of `Y` and `X` is a female.*

This is written in Prolog

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

# Defining relations by rules (cont)

The grand-parent relation can be defined as

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

The interesting point here is that there is a variable, Z, that is shared between the two sub-goals in the body and that does not appear in the head.

The interpreter, when asked a question about two persons, on one being the grand-parent of the other, tries to find another person (Z) who is the child of the first and the parent of the second.

# Defining relations by rules (cont)

The sister-hood relation can be defined as

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

This definition seems right but something strange happens if we query

```
?- sister(X,pat).
```

which corresponds to the question: "Who are the sisters of Pat?" The answers are:

```
X = ann;
X = pat
```

Pat is her own sister!

# Defining relations by rules (cont)

To fix this, we need a binary relation `different` and use it in the rule defining the relation `sister`:

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

The relation `different` can be defined by listing all the pairs of different persons:

```
different(bob,pat).
different(bob,ann).
...
```

This approach does not scale. We will see later how to define a general `different`, based on equality.