Logic and Constraint Programming

PROLOG

Prof. Fabrizio Fornari

May 11, 2022

SWI-Prolog

SWI-Prolog is a versatile implementation of the Prolog language. Although SWI-Prolog gained its popularity primarily in education, its development is mostly driven by the needs for **application development**.

SWI-Prolog aims at **scalability**. Its robust support for multi-threading exploits multi-core hardware efficiently and simplifies embedding in concurrent applications.

SWI-Prolog **unifies many extensions** of the core language that have been developed in the Prolog community such as *tabling*, *constraints*, *global variables*, *destructive assignment*, *delimited continuations* and *interactors*.

Let us download SWI Prolog

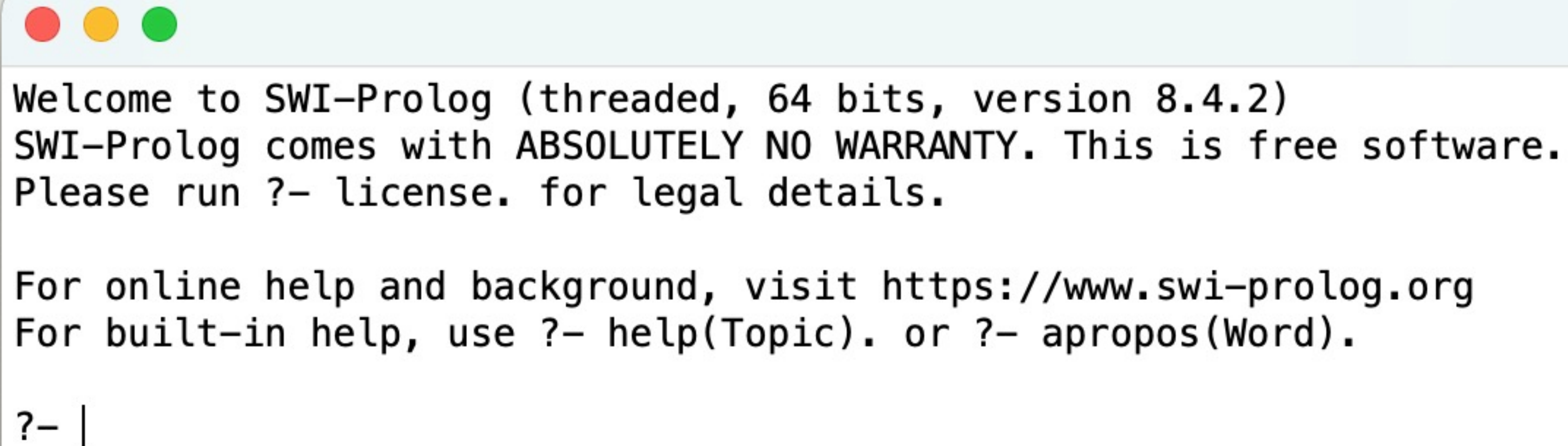
Stable version:

https://www.swi-prolog.org/download/stable

SWI Prolog documentation:

https://www.swi-prolog.org/download/stable/doc/SWI-Prolog-8.4.2.pdf

SWI Prolog Editor



SWI Prolog Editor

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.2)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- |
```

apt-get install –y swi-prolog

swipl

working_directory(D,D).

swipl -s fileName.pl

[fileName].

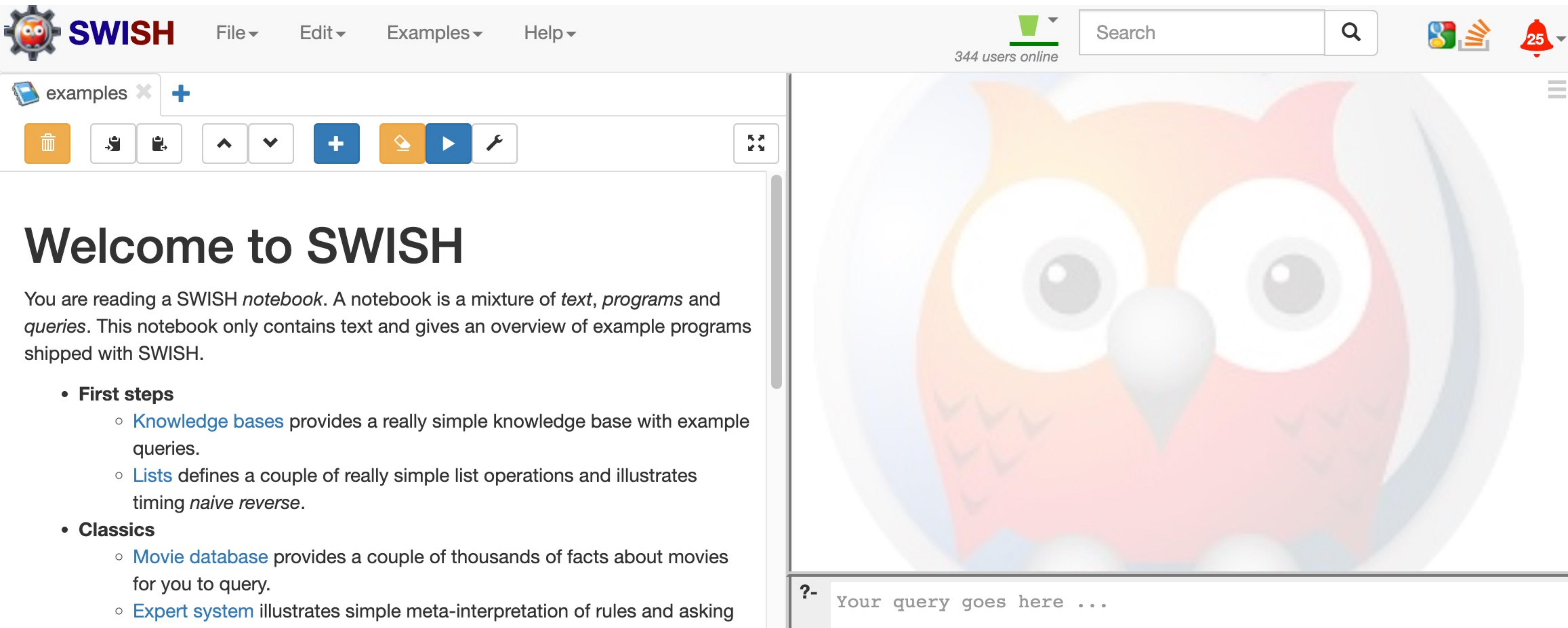
Load multiple file:

```
/* File: load.pl
Purpose: Load my program */

:- [ file1,
     file2,
     file3
   ].
```

SWI Prolog Editor

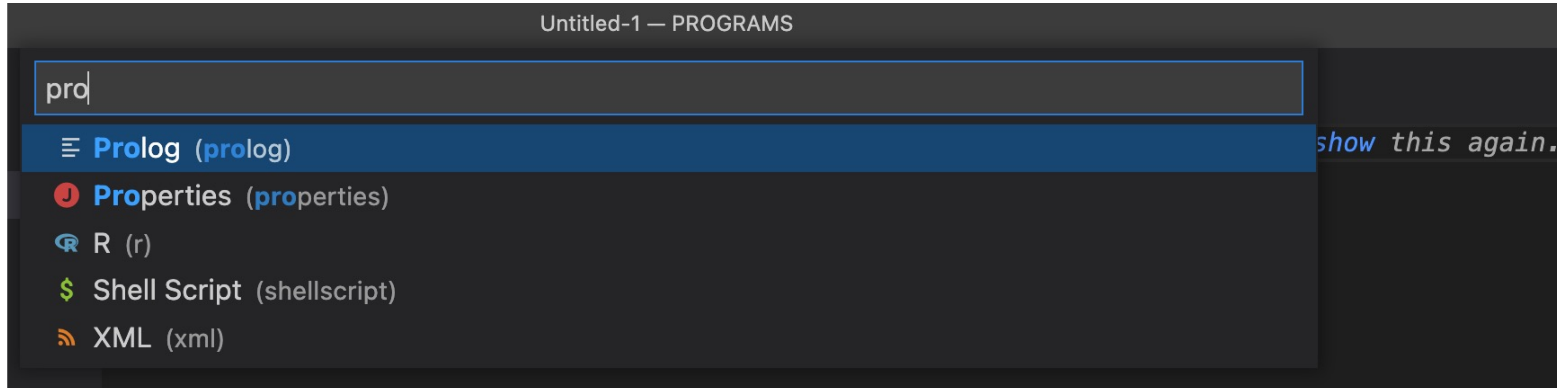
https://swish.swi-prolog.org/example/examples.swinb



VSC-Prolog

UNICAM
Università di Camerino
1336

VSC-Prolog



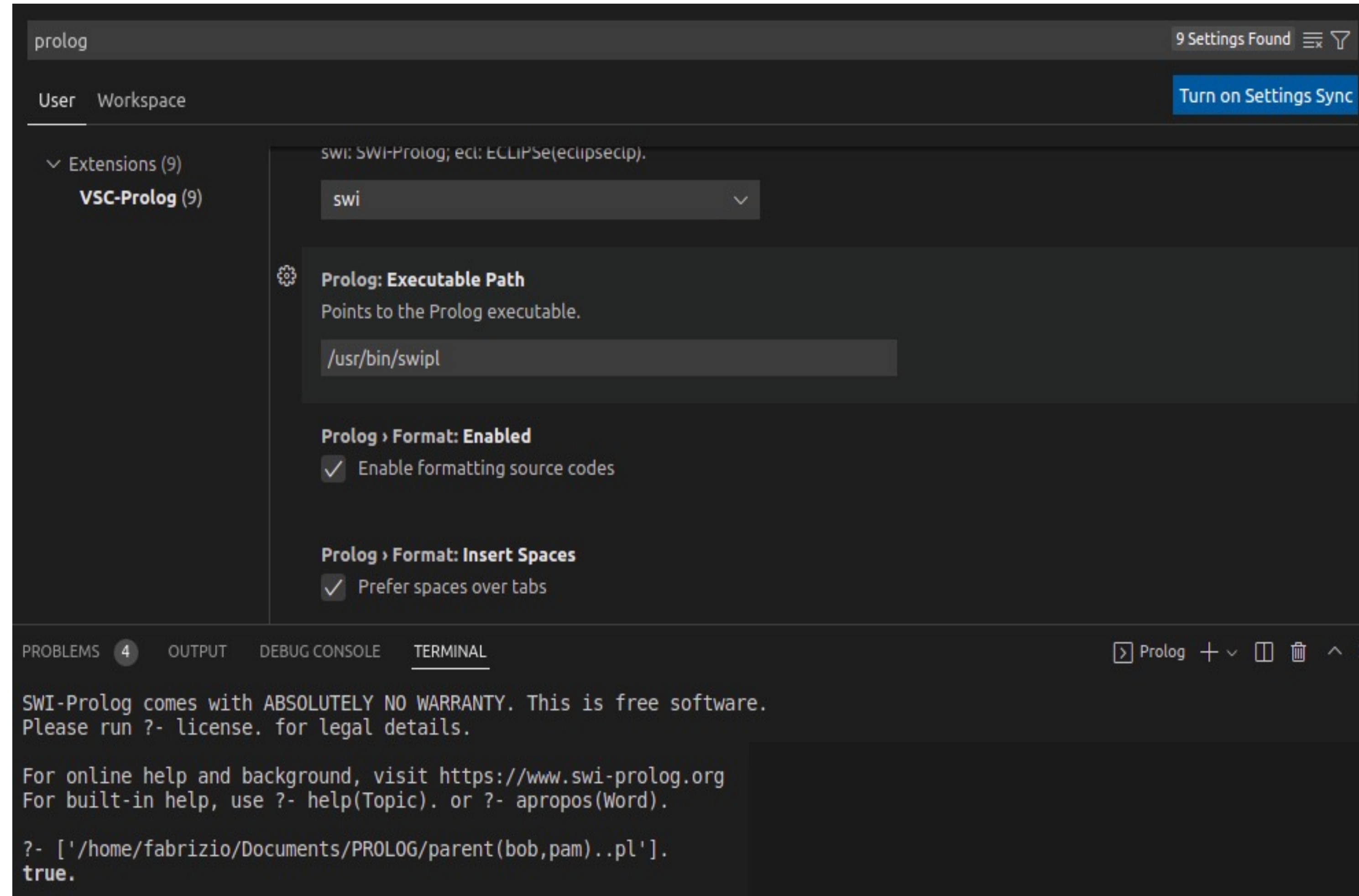
Ln 1, Col 1 Spaces: 4 UTF-8 LF Prolog

VSC-Prolog

Ubuntu:
/use/bin/swipl

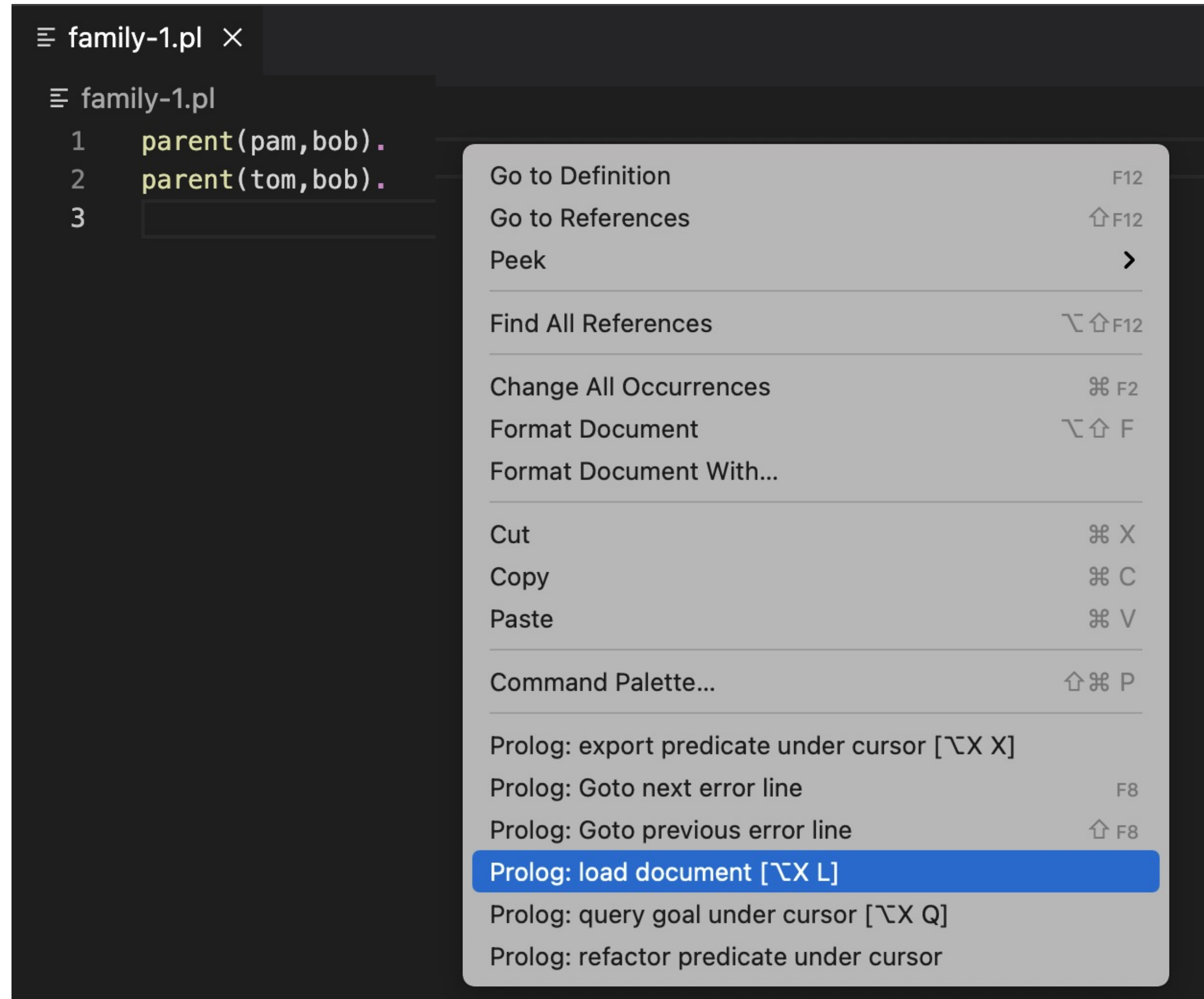
Windows:
C:\Program Files\swipl\bin\swipl.exe

Mac:
/Applications/SWI-Prolog.app/Contents/MacOS/swipl

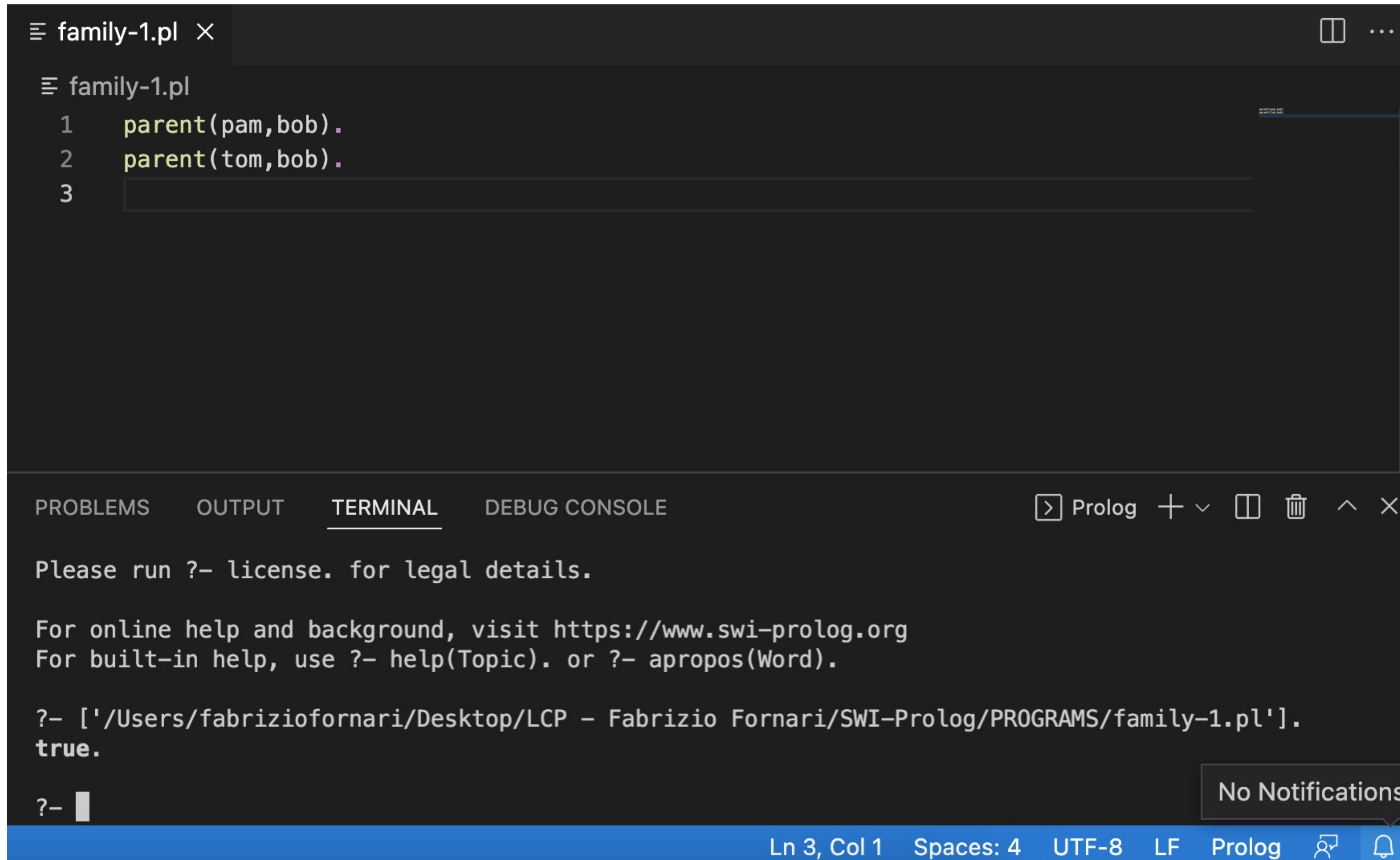


VSC-Prolog

parent(pam,bob).
parent(tom,bob).



VSC-Prolog



Prolog

parent(pam,bob).
parent(tom,bob).

Let us ask questions:

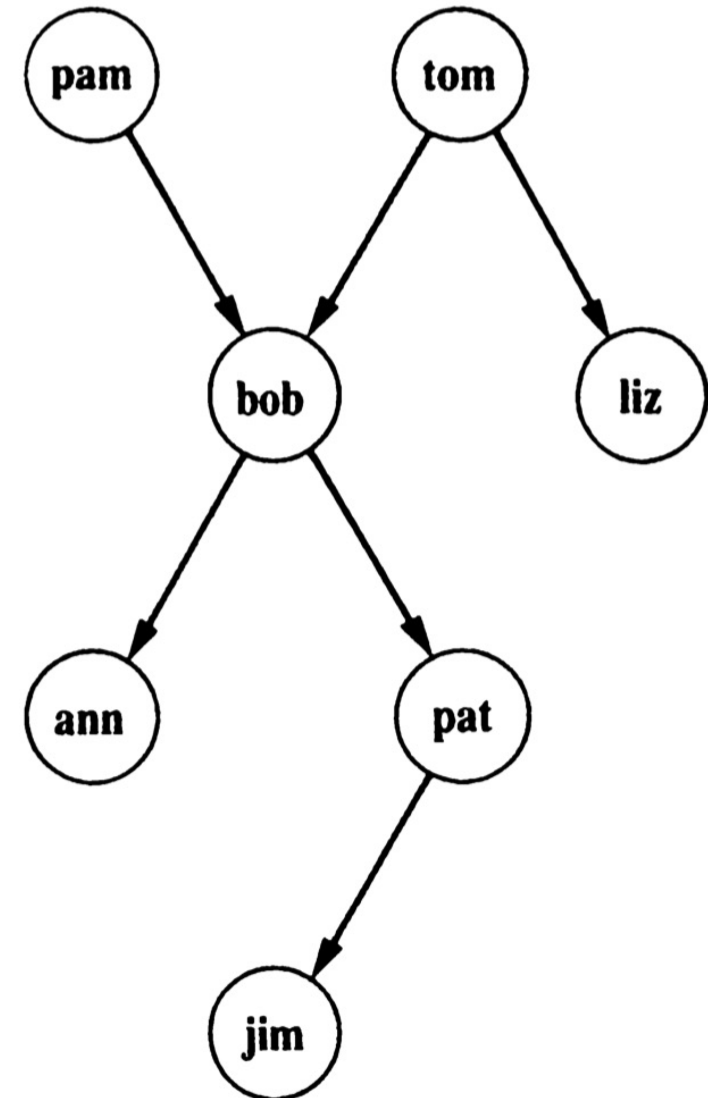
?- parent(bob,pam).
?- parent(pam,bob)

```
?- parent(bob,pam).
false.

?- parent(pam,bob).
true.
```

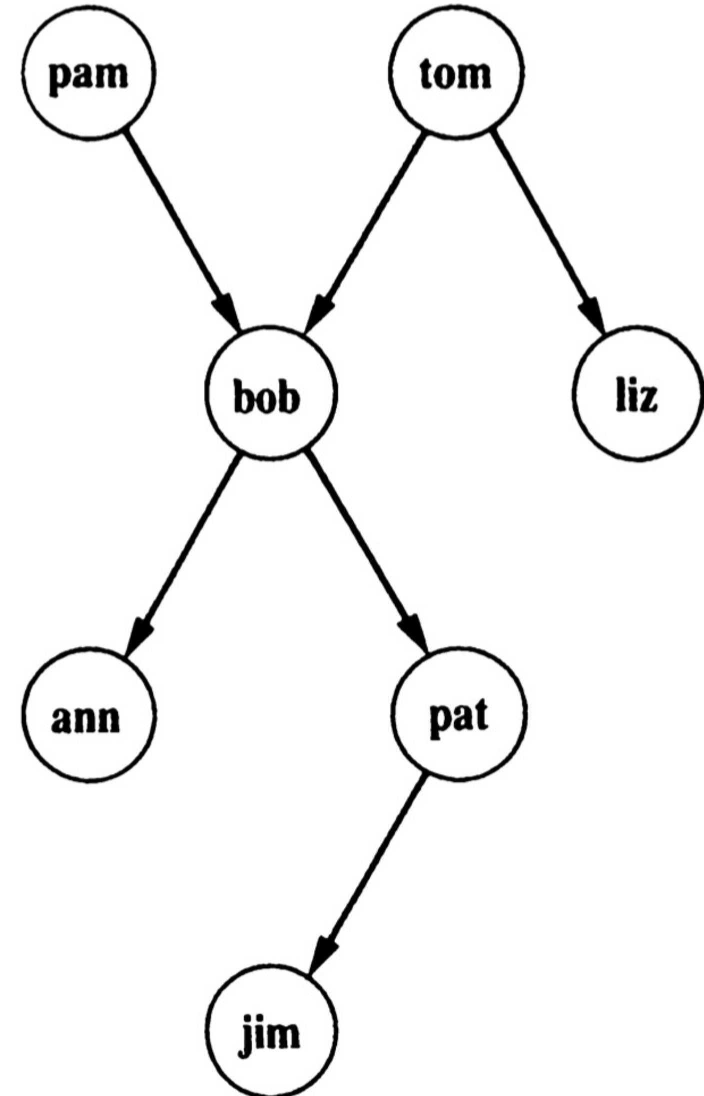
Prolog

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



Prolog

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



Prolog

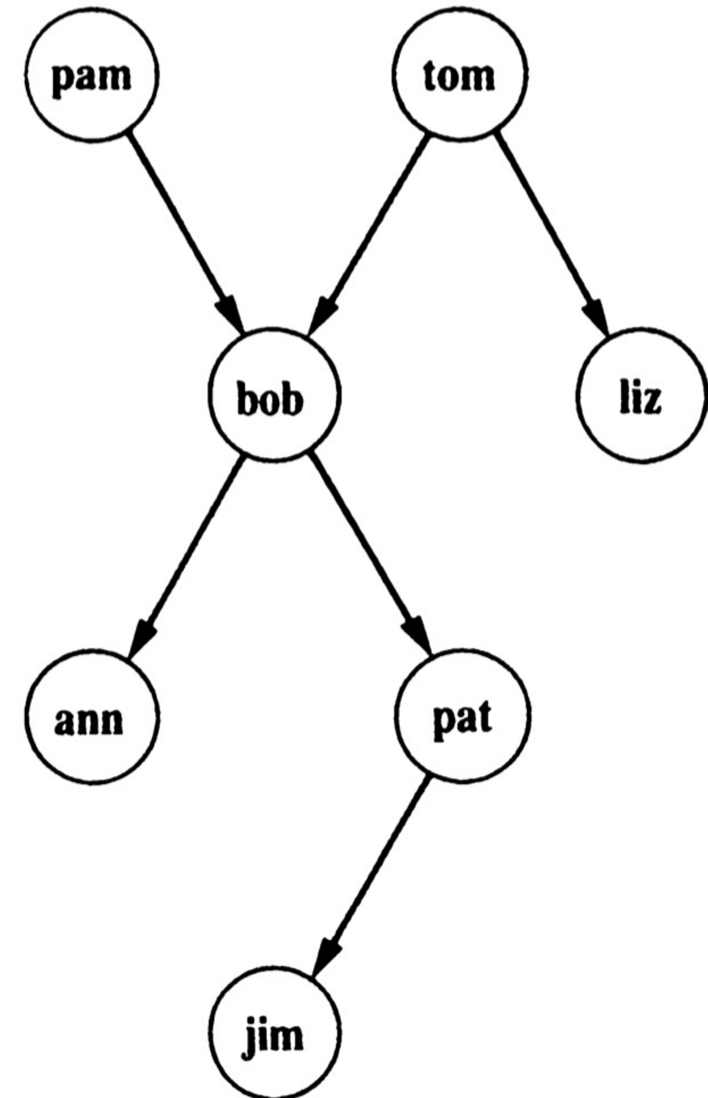
Let us ask questions:

?- parent(liz,pat).
?- parent(tom,ben).

?- parent(X,liz). Who is a parent of liz?

?- parent(bob,X). How many results?

Let us write a semicolon ;
to display other results



Prolog

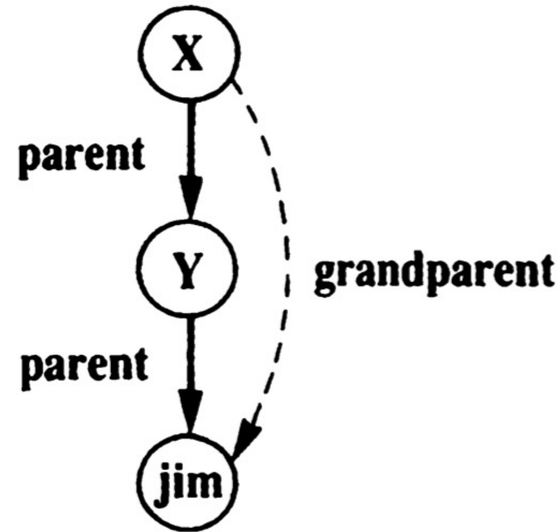
Let us ask questions:

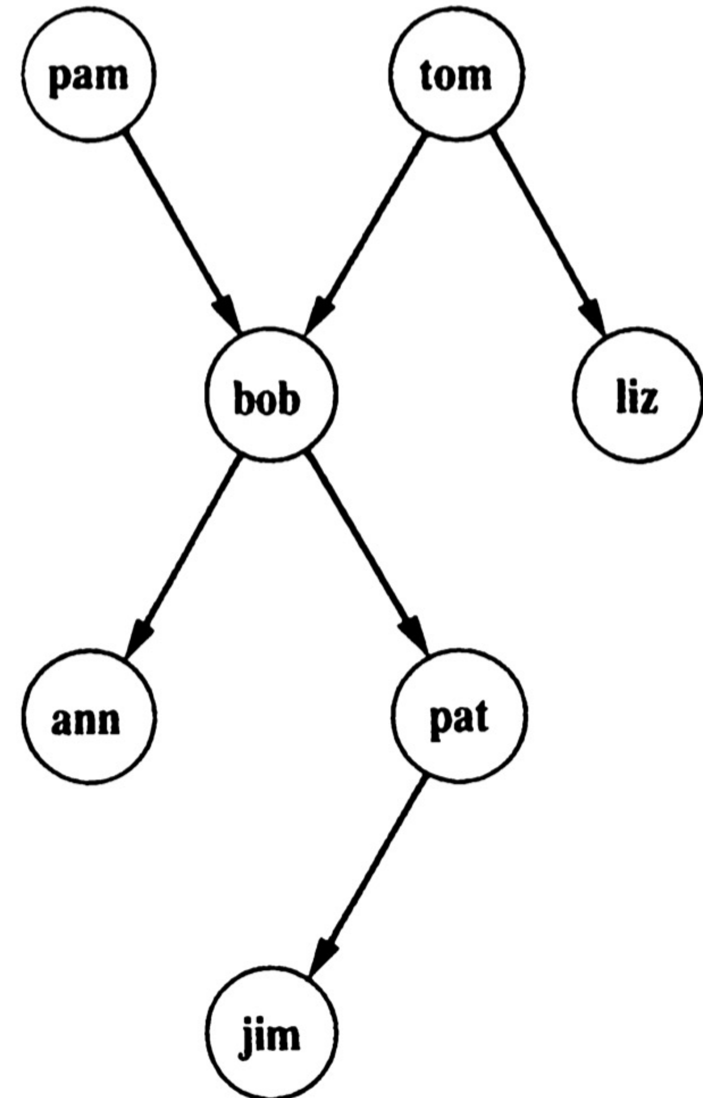
Who is a parent of whom?
?- parent(X,Y).

Who is the grandparent of jim?
?- parent(Y,jim),parent(X,Y).

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

Who are tom's grandchildren?
?- parent(tom,X),parent(X,Y).





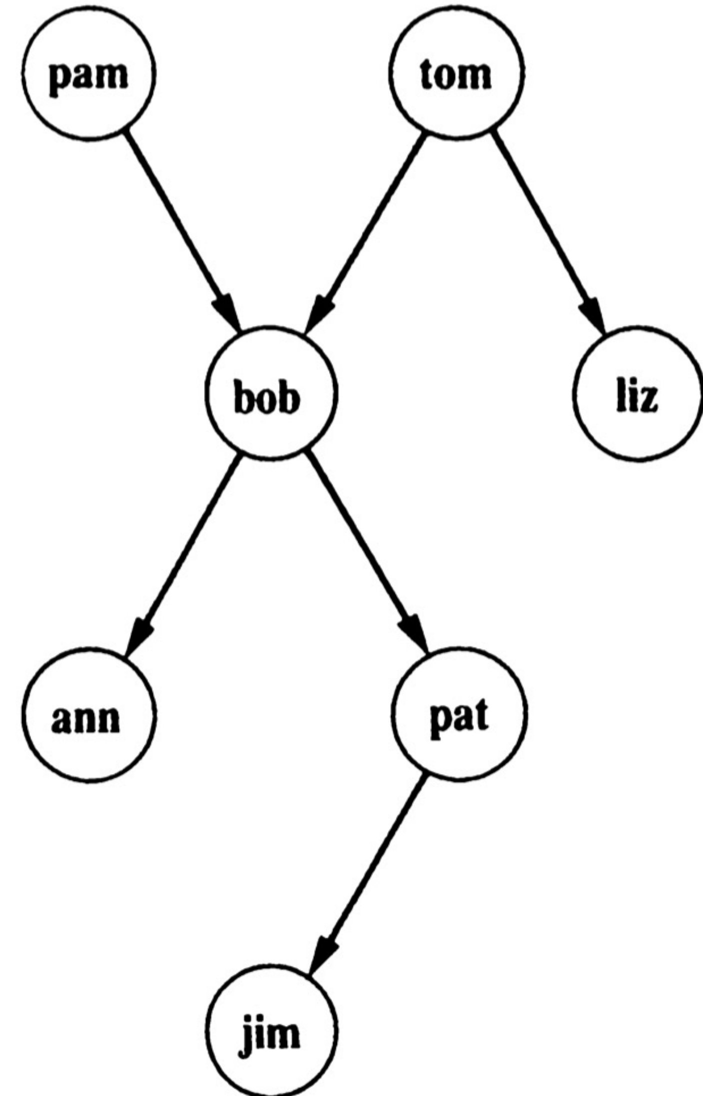
Prolog

Let us ask questions:

Do Ann and Pat have a common parent?

(1) Who is a parent, X, of Ann?
(2) Is X also a parent of Pat?

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



Prolog - Recap

It is easy in Prolog to define a relation, such as the **parent** relation, by stating the n-tuples of objects that satisfy the relation.

The user can easily query the Prolog system about relations defined in the program.

A Prolog program consists of *clauses*. Each clause terminates with a full stop.

The arguments of relations can (among other things) be: concrete objects, or constants, or general objects such as X and Y.

Prolog - Recap

Questions to Prolog consist of one or more *goals*. A sequence of goals, such as:
parent(X,ann),parent(X,pat)

Means the conjunction of the goals:

X is a parent of Ann, *and*
X is a parent of Pat.

The word ‘goals’ is used because Prolog interprets questions as goals that are to be satisfied. To ‘satisfy a goal’ means to logically deduce the goal from the program.

In the case of a positive answer we say that the corresponding goal was *satisfiable* and that the goal *succeded*. Otherwise the goal was *unsatisfiable* and it *failed*.

Prolog ISO/IEC 13211-1

The ISO Prolog standard: **ISO/IEC 13211-1** was published in 1995.
https://www.iso.org/standard/21413.html

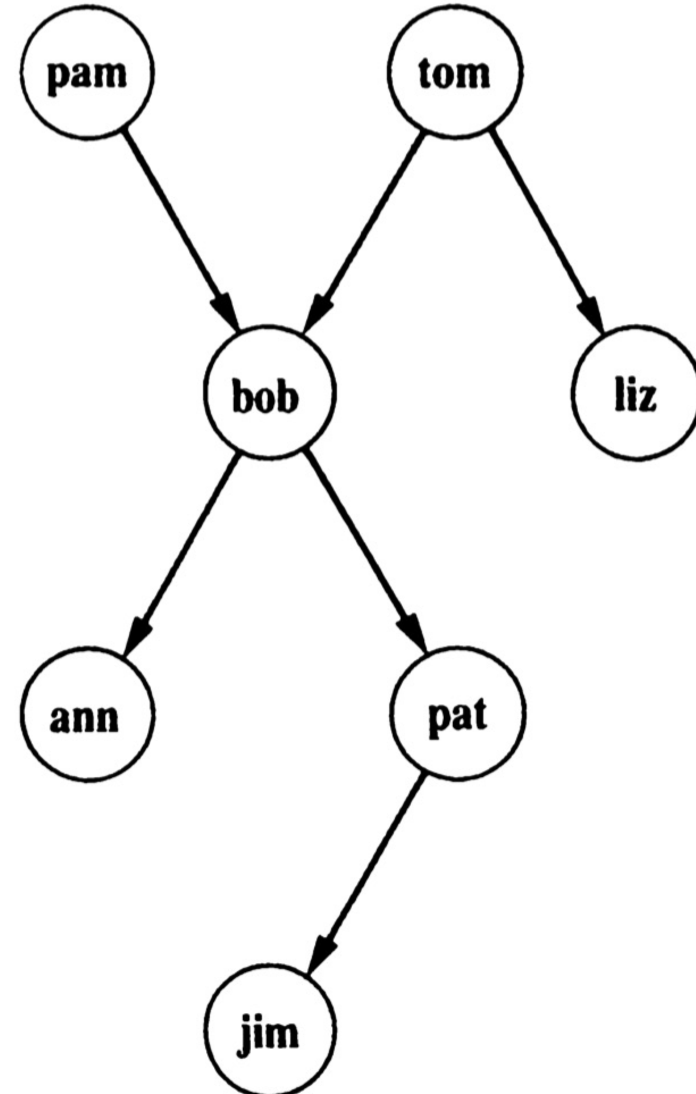
The original intention of the standards process was, to standardize the existing practices of the many implementations of Prolog.

https://en.wikipedia.org/wiki/Comparison_of_Prolog_implementations

Defining relations by rules

Extend the program specifying **female()** and **male()** relations.

female(pam).
male(tom).
male(bob).
..



Defining relations by rules

Extend the program specifying **female()** and **male()** relations.

female(pam).

male(tom).

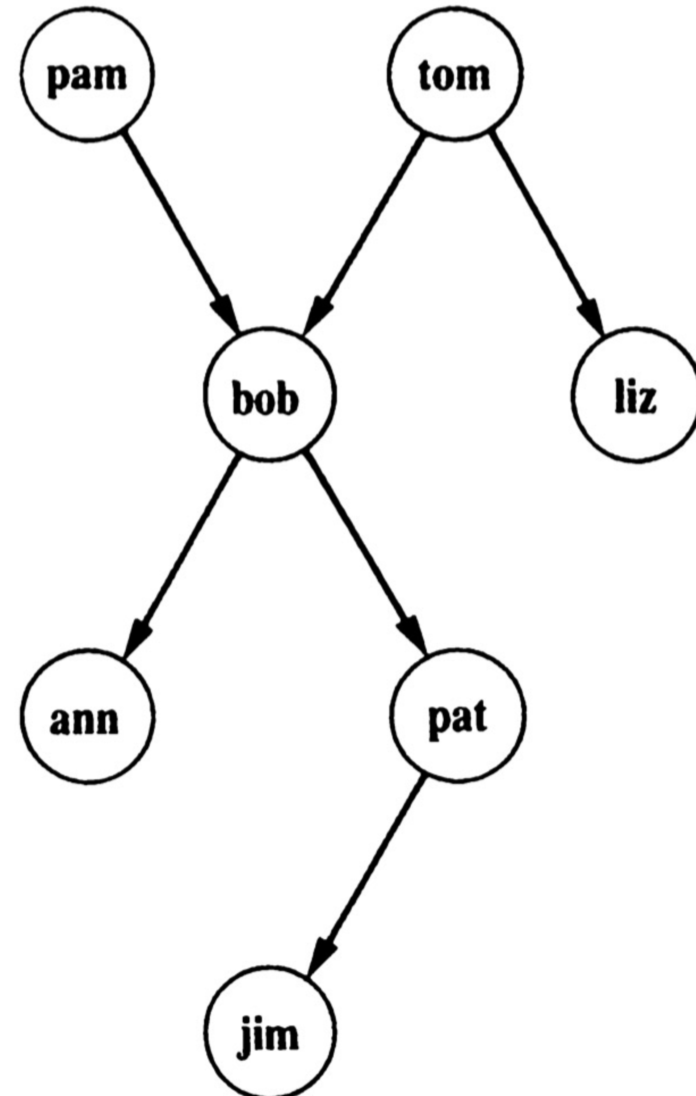
male(bob).

female(liz).

female(pat).

female(ann).

male(jim).



Defining relations by rules

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

```
?- ['/Users/fabriziofornari/Desktop/LCP - Fabrizio Fornari/SWI-Prolog/PROGRAMS/family.pl'].
Warning: /Users/fabriziofornari/Desktop/LCP - Fabrizio Fornari/SWI-Prolog/PROGRAMS/family.pl:11:
Warning:    Clauses of female/1 are not together in the source-file
Warning:    Earlier definition at /Users/fabriziofornari/Desktop/LCP - Fabrizio Fornari/SWI-Prolog/PROGRAMS
/family.pl:8
Warning:    Current predicate: male/1
Warning:    Use :- discontiguous female/1. to suppress this message
Warning: /Users/fabriziofornari/Desktop/LCP - Fabrizio Fornari/SWI-Prolog/PROGRAMS/family.pl:14:
Warning:    Clauses of male/1 are not together in the source-file
Warning:    Earlier definition at /Users/fabriziofornari/Desktop/LCP - Fabrizio Fornari/SWI-Prolog/PROGRAMS
/family.pl:9
Warning:    Current predicate: female/1
Warning:    Use :- discontiguous male/1. to suppress this message
true.

?-
```

Warning to enforce best practices, which is to put all related clauses together in the source file.

Defining relations by rules

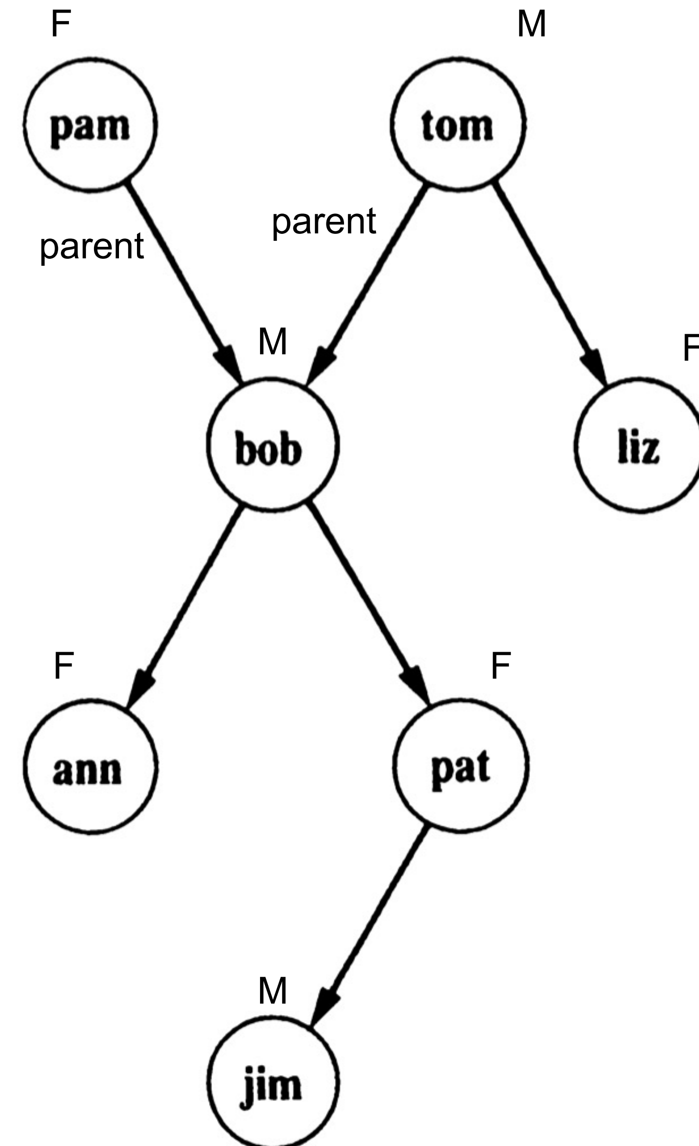
female and **male** are *unary* relations.

female(ann).

male(jim).

parent is a *binary* relation.

parent(bob, ann).
parent(pat,jim).



Defining relations by rules

female and **male** are *unary* relations.

female(ann).

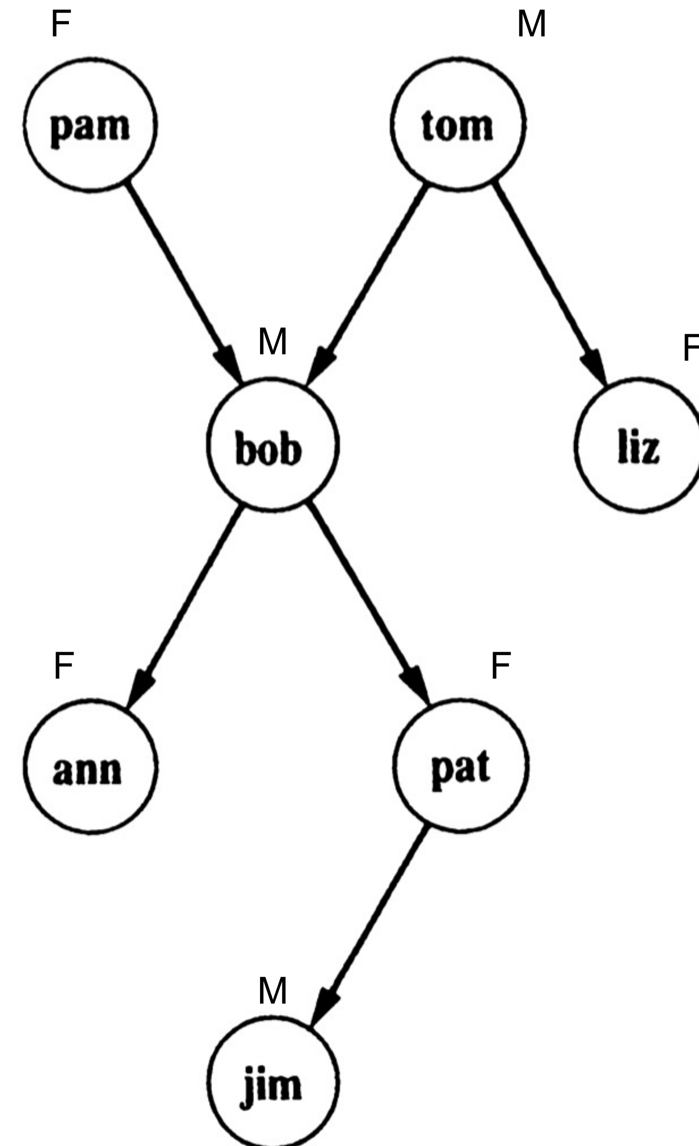
male(jim).

As an alternative, we could define **sex** as a *binary* relation.

sex(pam, feminine).

sex(tom, masculine).

…

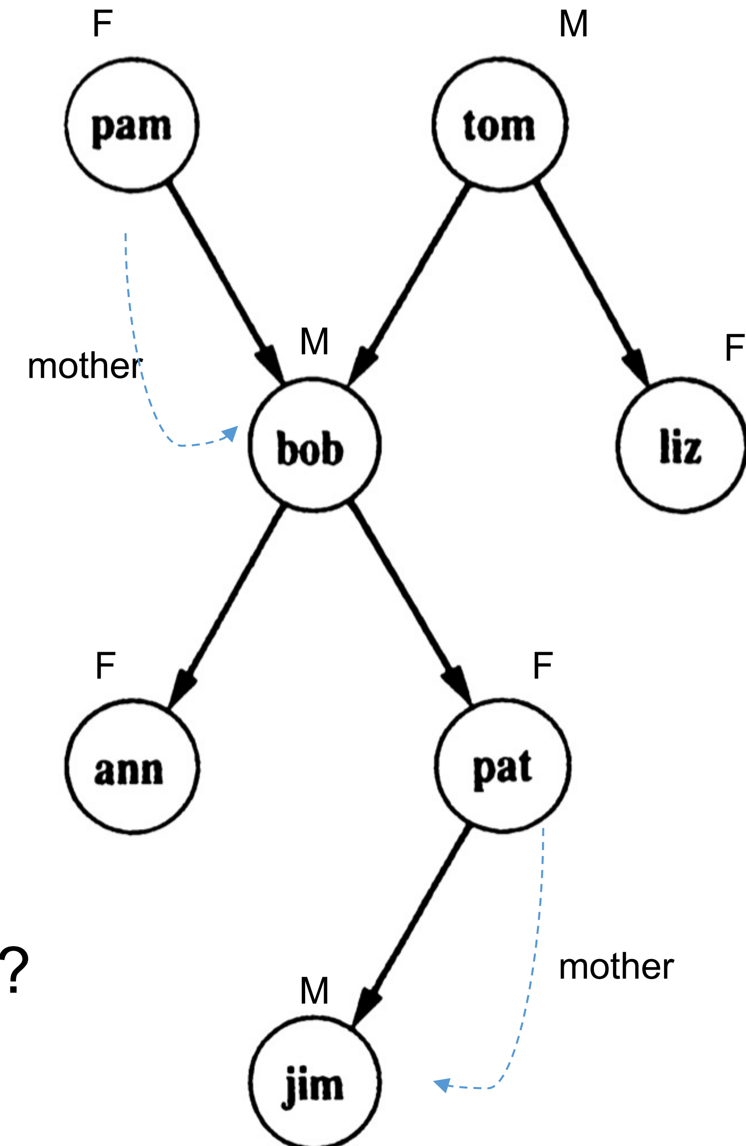


Defining relations by rules

Introduce the binary relation **mother**.

mother(pam,bob).
mother(pat,jim).

How did you figure out who is the mother of bob?



Defining relations by rules

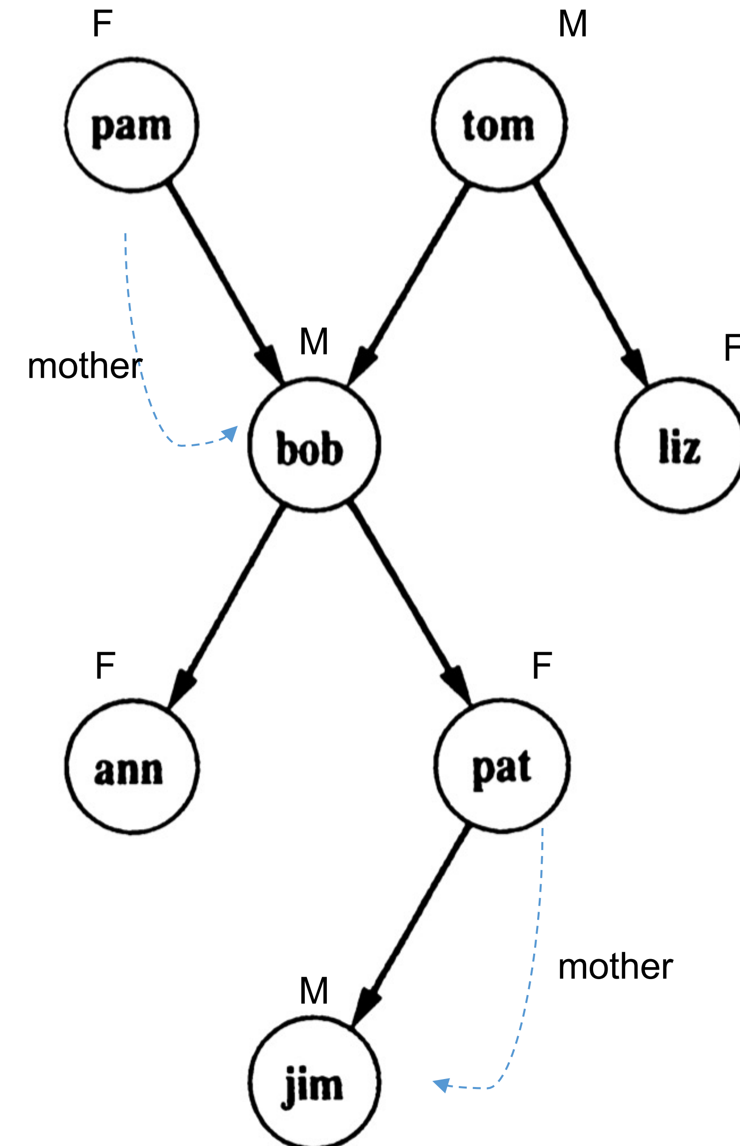
Introduce the binary relation **mother**.

What defines a mother? (in our environment)

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

In Prolog

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



Defining relations by rules

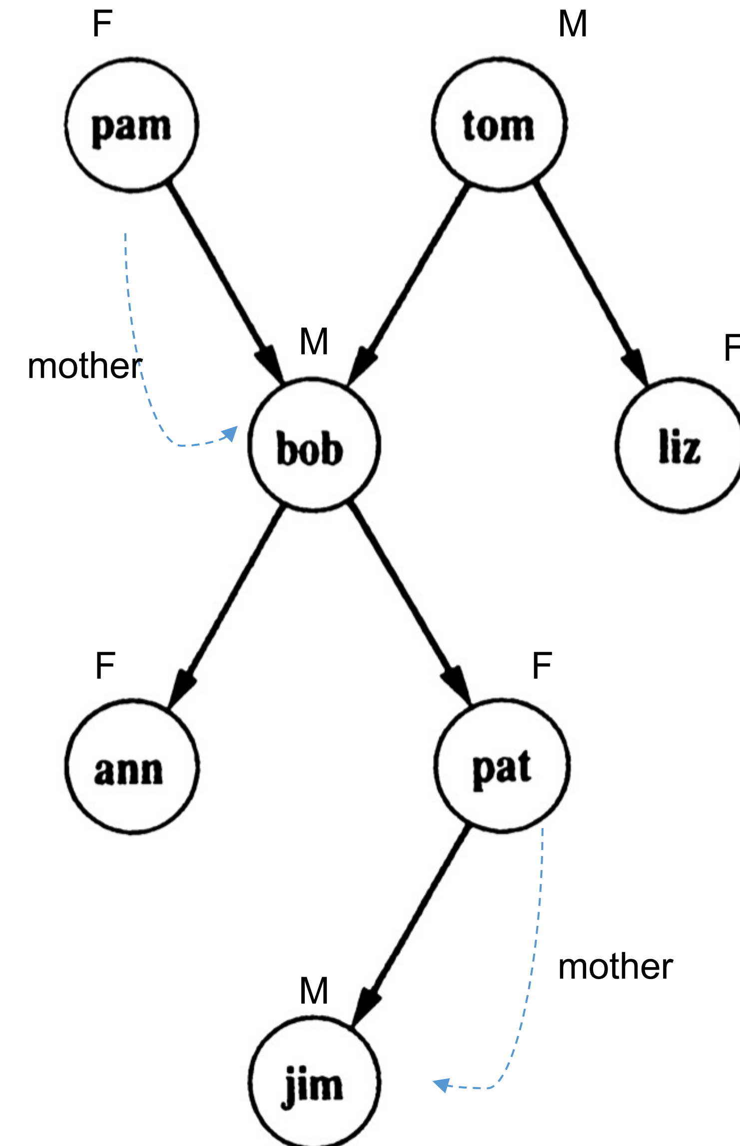
Introduce the binary relation **mother**.

In Prolog **rule**

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

The Prolog symbol ‘:-’ is read as ‘if’.

For all X and Y,
    if X is a parent of Y and X is female then
    X is the mother of Y.



Rules vs Facts

Rule

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

Rules specify things that are true if some condition is satisfied.

Fact

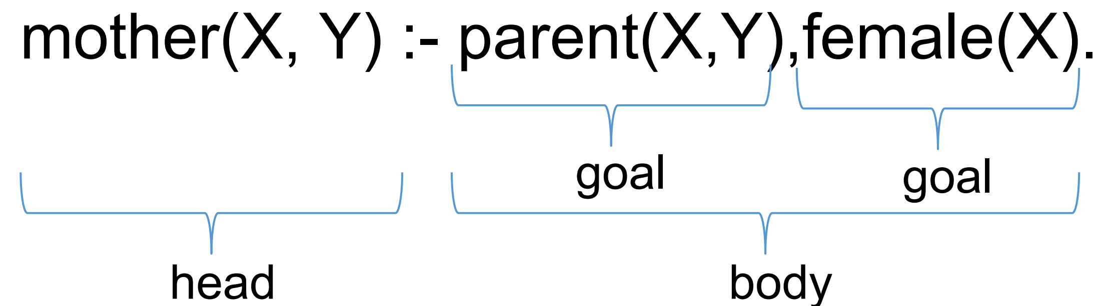
parent(tom, liz).

Something always true

Rules

Rules have:

- a condition part (the right-hand side of the rule). Also called the *body* of a clause.
- a conclusion part (the left-side of the rule). Also called the *head* of a clause.



If the condition part '**parent(X, Y), female(X)**' is *true* then a logical consequence of this is **mother(X, Y)**.

Rules

Extend the program with:

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

Ask whether Pam is mother of Bob:

```
?- mother(pam, bob).
```

Note: there is no fact about **mother** in the program.

Rules

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

?- mother(pam, bob).

When we specify **pam** and **bob** we are *instantiating* the variables X and Y.
X = **pam** and Y = **bob**

After the instantiation we have obtained a special case of our general rule.
The special case is:

mother(pam, bob) :- parent(pam, bob), female(pam).

Rules

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

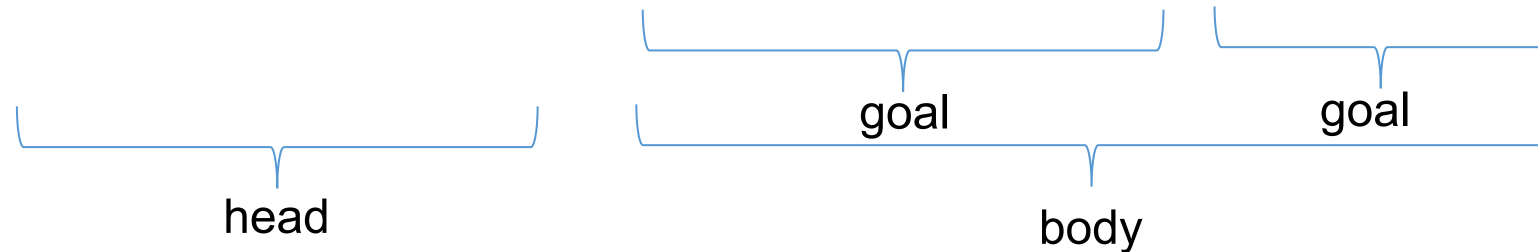
?- mother(pam, bob).

When we specify **pam** and **bob** we are *instantiating* the variables X and Y.
X = **pam** and Y = **bob**

After the instantiation we have obtained a special case of our general rule.
The special case is:

mother(pam, bob) :- parent(pam, bob), female(pam).

Rules

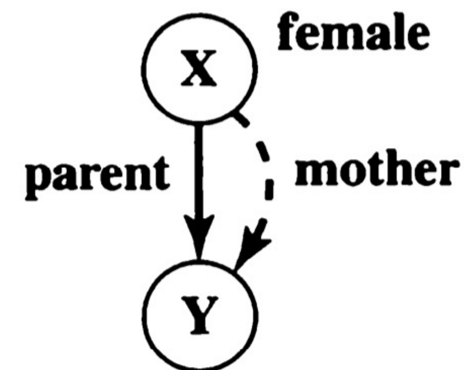


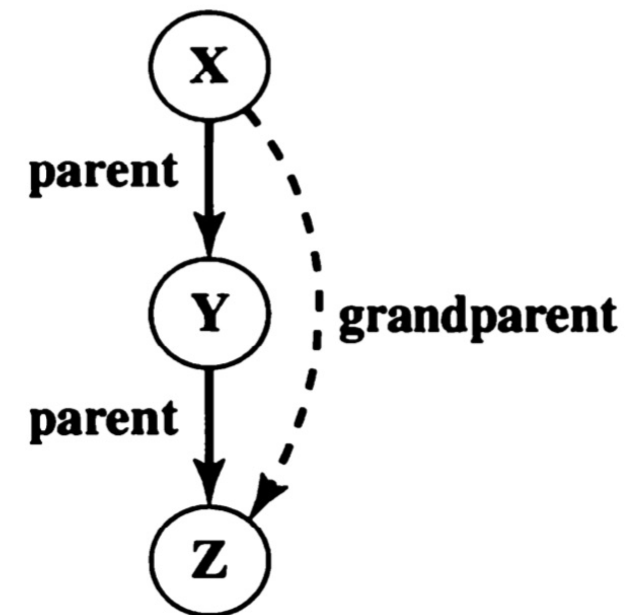
The two goals are trivial. They can be found as facts in the program.

This means that the conclusion part of the rule is also true, and Prolog will answer the question with **true.**

Graphically see relations

How? (We already anticipated that..)





mother and **grandparent** relations

Graphically see relations

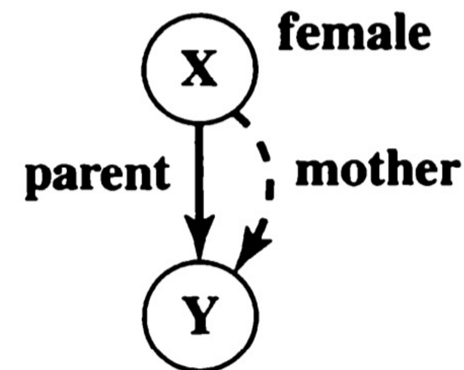
- Nodes in the graphs correspond to objects (arguments of the relations).

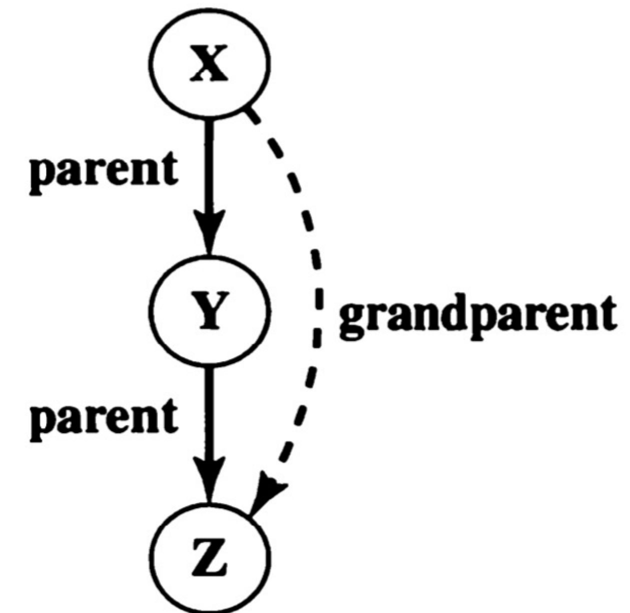
- Arcs between nodes correspond to binary relations.

- Arcs are oriented to point from the first argument of the relation to the second argument.

- Unary relations are indicated by simply labelling the object with the name of the relation

- Defined relations (by rules) are represented by dashed arcs.

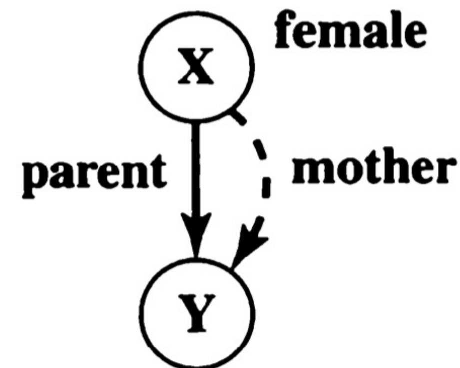


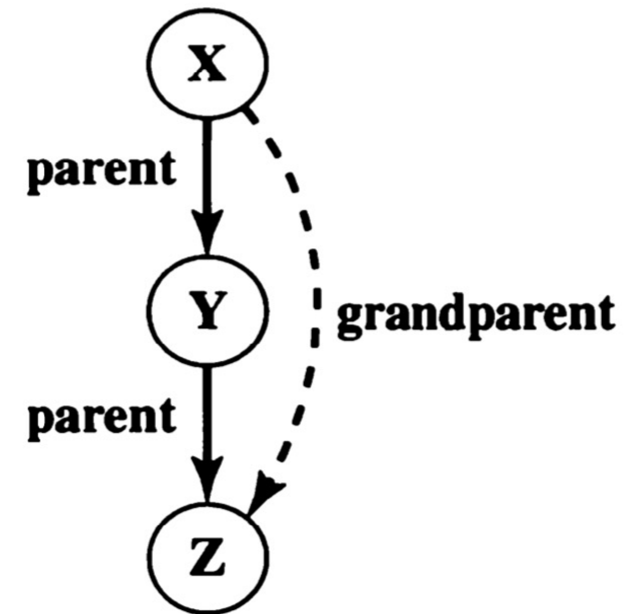


mother and **grandparent** relations

Graphically see relations

If the relations shown by solid arcs hold, then the relation shown by a dashed arc also holds.





Extend the program adding the **grandparent** relation

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

Ask Prolog: ?- grandparent(X, Z).

mother and **grandparent** relations

What does Prolog answer?

Convention

From:

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

To:

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

The head and the goals each on a separate line.

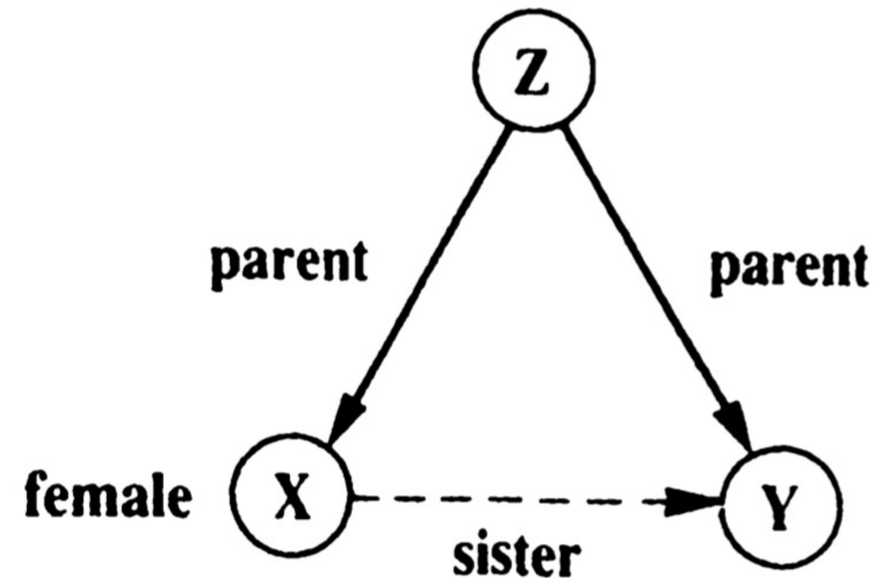
Exercise

Define the relation **sister**.

Exercise

Define the relation **sister**.

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

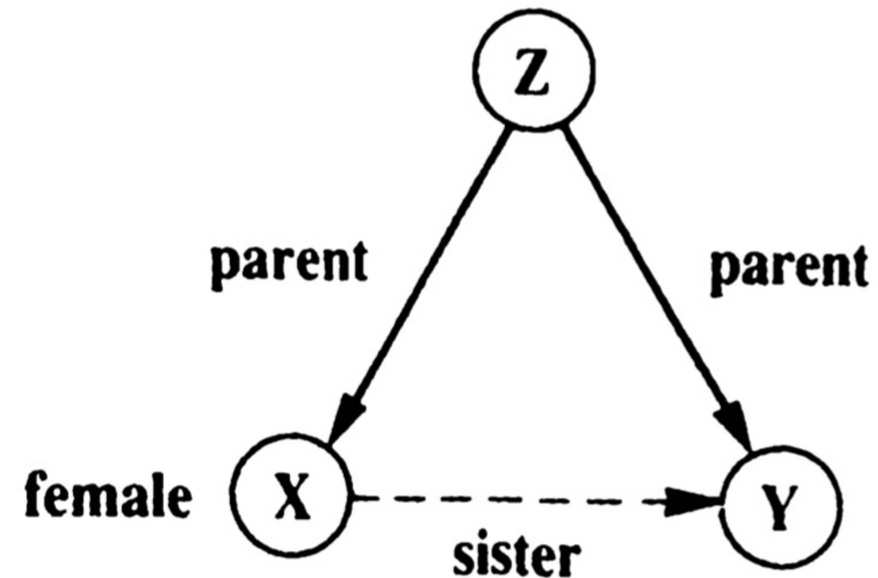


Some Z must be a parent of X, and this same Z must be a parent of Y

Exercise

Ask:
?- sister(ann, pat).

Ask:
?- sister(X, pat).



Is Anything weird about the answers?

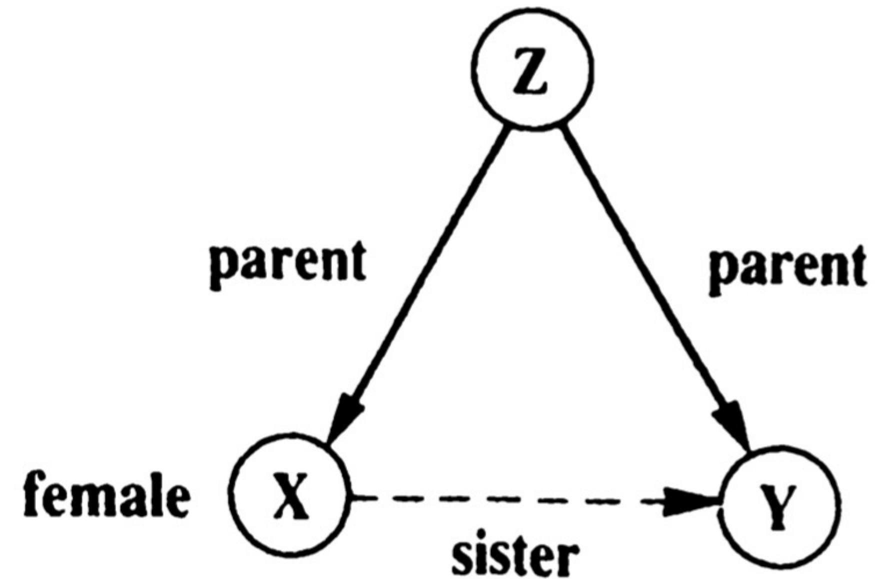
?- sister(X, pat).
X = ann ;
X = pat ;

Pat is a sister to herself!

Exercise

We can state X \= Y to express that X and Y must be different.

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



Recap

- Prolog programs can be extended by simply adding new clauses.
- Prolog clauses are of three types: *facts*, *rules* and *questions*.
- *Facts* declare things that are *always*, unconditionally, true.
- *Rules* declare things that are true depending on a given condition.
- By means of *questions* the user can ask the program what things are true.
- A Prolog clause consists of the *head* and the *body*. The body is a list of *goals* separated by commas. Commas between goals are understood as conjunctions.
- A fact is a clause that just has the head and no body. Questions only have the body. Rules consist of the head and the (non-empty) body.
- In the course of computation, a variable can be substituted by another object. We say that a variable becomes *instantiated*.
- Variables are assumed to be *universally quantified* and are read as ‘for all’.

Extend the “family” program

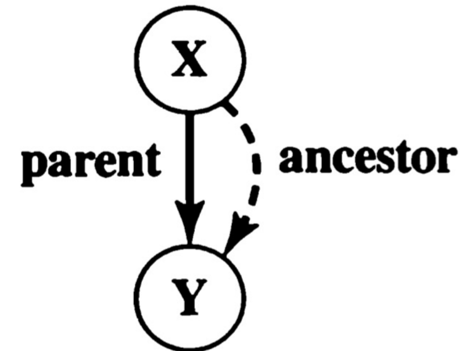
Let us add one more relation to our family program, the **ancestor** relation.

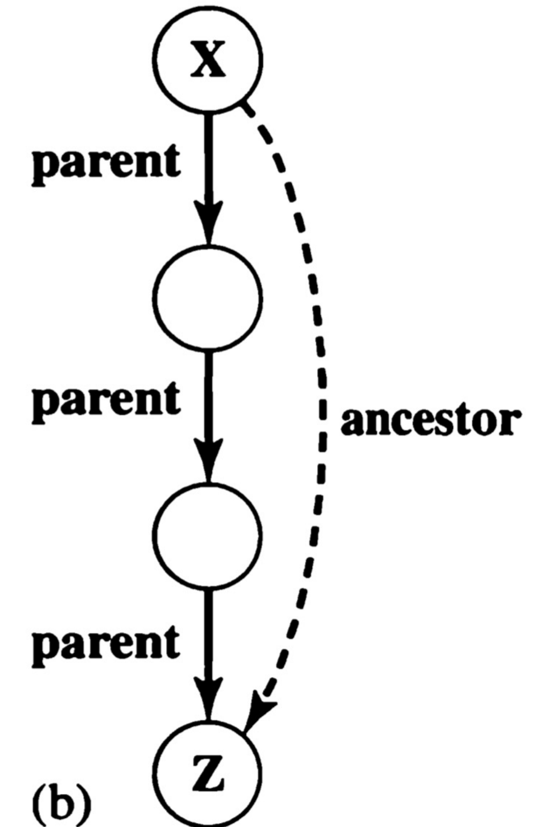
1st rule:

```
ancestor( X, Z) :-
    parent( X, Z).
```

2nd rule:

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





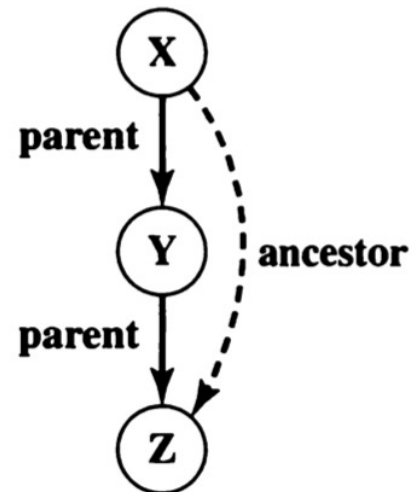
Extend the “family” program

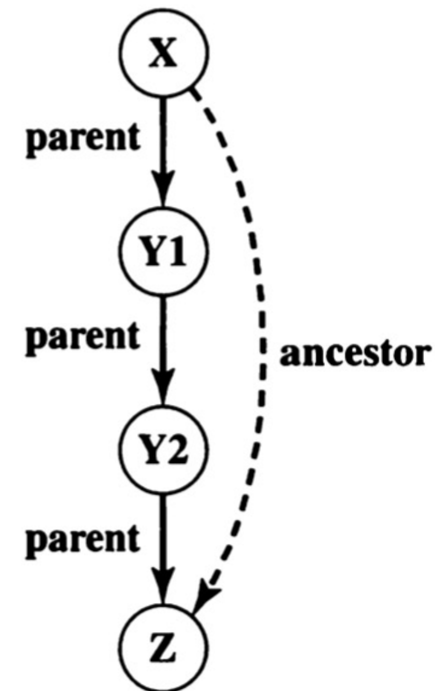
Let us add one more relation to our family program, the **ancestor** relation.

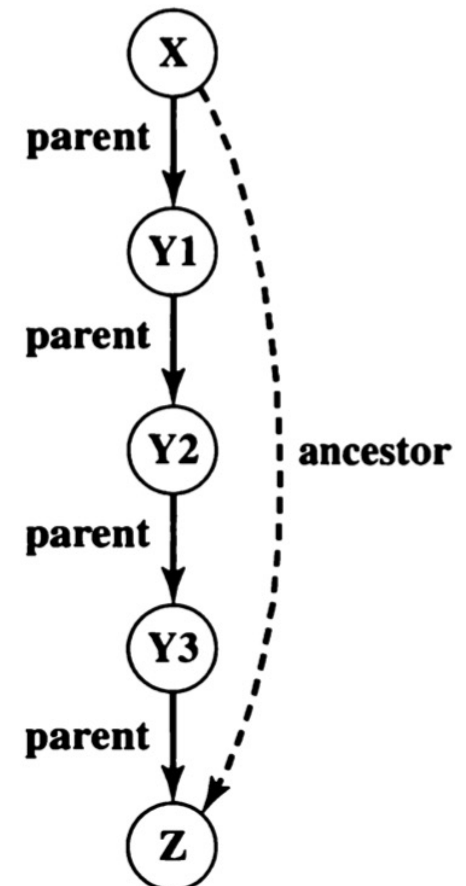
1st rule:
ancestor(X, Z) :-
    parent(X, Z).

2nd rule:
ancestor(X, Z) :-
    parent(X, Y),
    parent(Y, Z).

3rd rule:
ancestor(X, Z) :-
    parent(X, Y1),
    parent(Y1, Y2),
    parent(Y2, Z).







Extend the “family” program

Let us add one more relation to our family program, the **ancestor** relation.

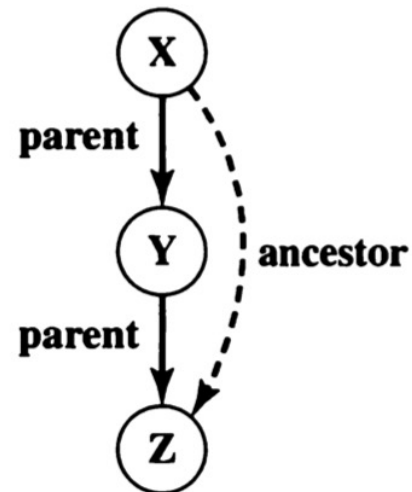
1st rule:
ancestor(X, Z) :-
  parent(X, Z).

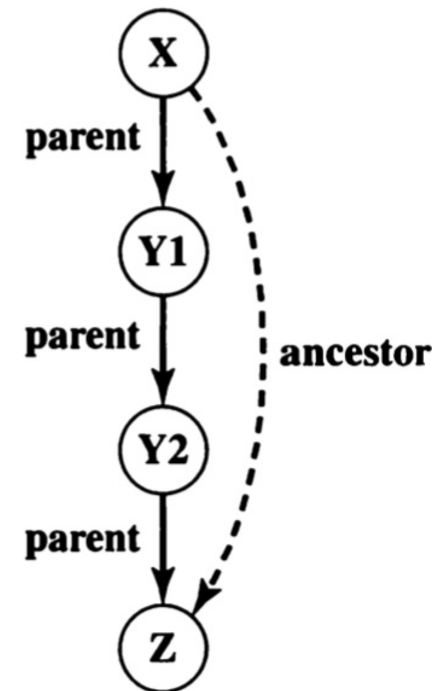
2nd rule:
ancestor(X, Z) :-
  parent(X, Y),
  parent(Y, Z).

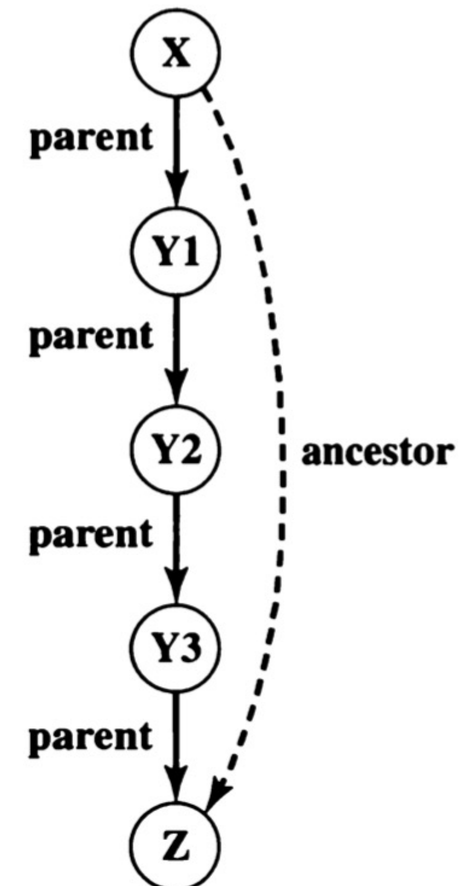
3rd rule:
ancestor(X, Z) :-
  parent(X, Y1),
  parent(Y1, Y2),
  parent(Y2, Z).

4th rule:
ancestor(X, Z) :-
  parent(X, Y1),
  parent(Y1, Y2),
  parent(Y2, Y3),
  parent(Y3, Z).

Nth rule:
…



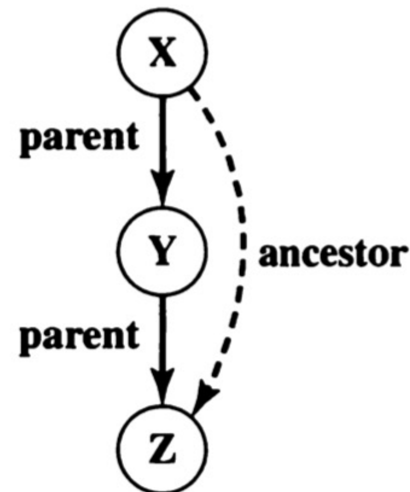


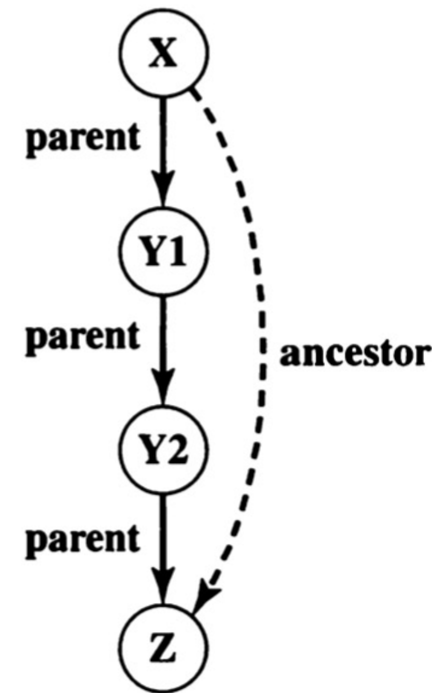


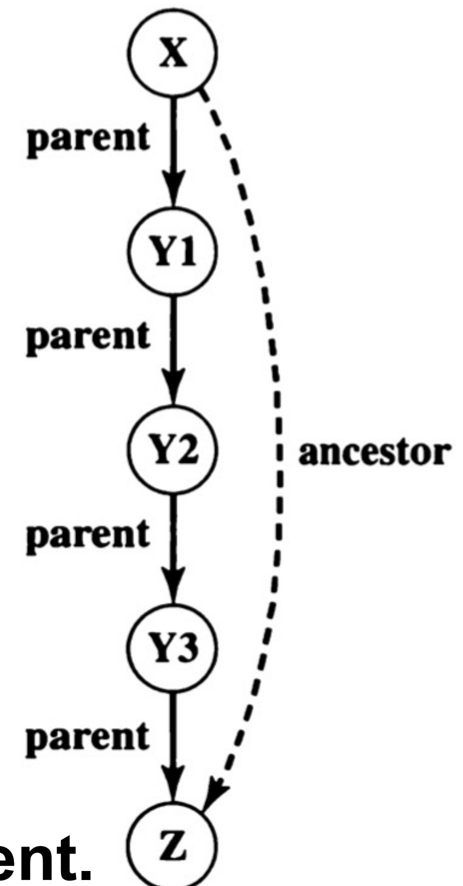
Extend the “family” program

Let us add one more relation to our family program, the **ancestor** relation.

1st rule:
```
ancestor( X, Z) :-
    parent( X, Z).
```
2nd rule:
```
ancestor( X, Z) :-
    parent( X, Y),
    parent( Y, Z).
```
3rd rule:
```
ancestor( X, Z) :-
    parent( X, Y1),
    parent( Y1, Y2),
    parent( Y2, Z).
```
4th rule:
```
ancestor( X, Z) :-
    parent( X, Y1),
    parent( Y1, Y2),
    parent( Y2, Y3),
    parent( Y3, Z).
```
Nth rule:
…



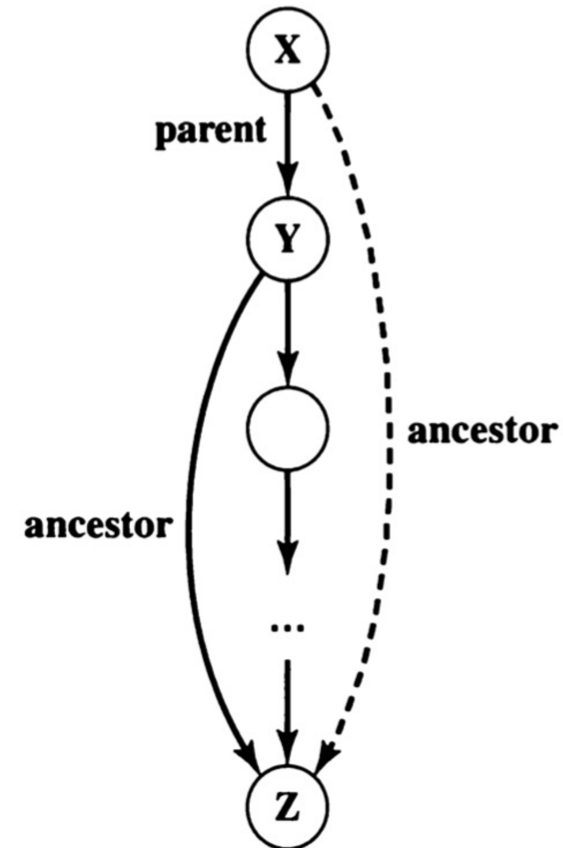




This program is lengthy and it only works to some extent.

rules

There is a much more elegant and correct formulation of the **ancestor** relation.

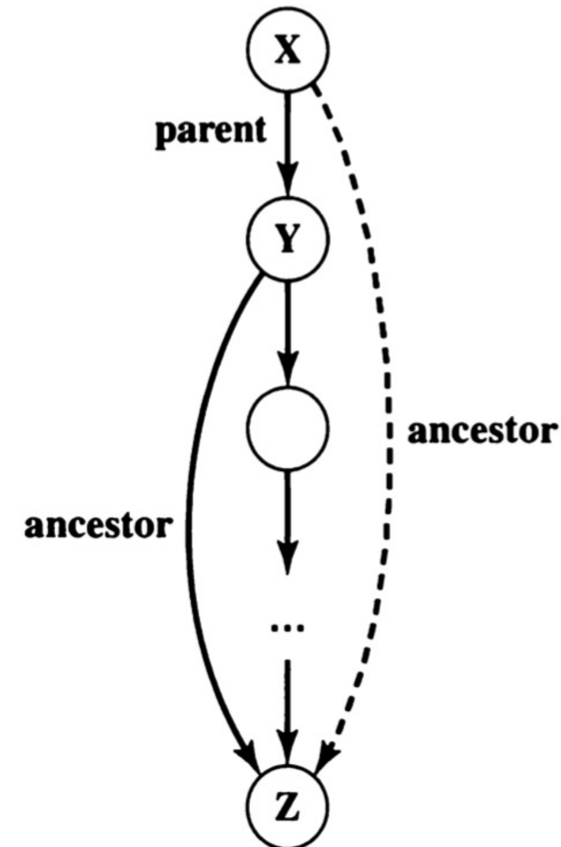


Recursive rules

There is a much more elegant and correct formulation of the **ancestor** relation.

For all X and Z,
X is an ancestor of Z if
there is a Y such that
(1) X is a parent of Y and
(2) Y is an ancestor of Z.

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



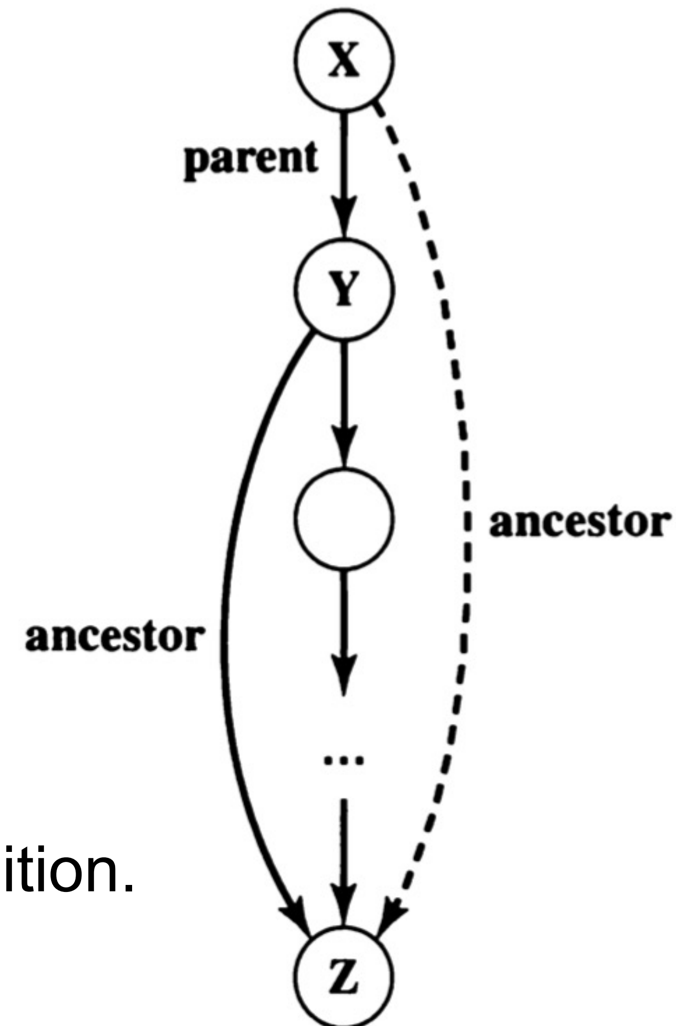
Recursive rules

Ancestor relation program

1st rule:
```
ancestor( X, Z) :-
    parent( X, Z).
```

2nd rule:
```
ancestor( X, Z) :-
    parent( X, Y),
    ancestor( Y, Z).
```

The key is the use of **ancestor** itself in its definition.
Such a definition is called ***recursive*** definition.



Recursive rules

Are logically correct and understandable.

Prolog can easily use recursive definitions.

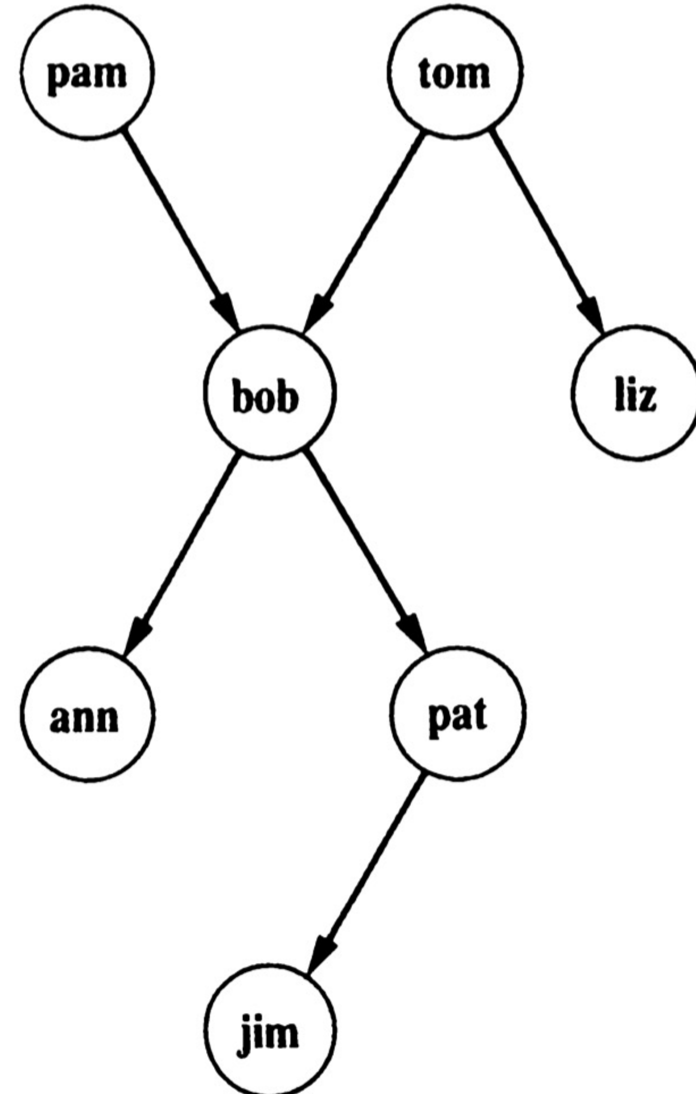
Recursive programming is, one of the fundamental principles of programming in Prolog.

It is necessary for solving task of significant complexity.

Ancestor program

Let us ask Prolog: Who are Pam's successors?

How can we formulate such a question?



Ancestor program

Let us ask Prolog: Who are Pam's successors?

?- ancestor(pam, X).

X = bob;
X = ann;
X = jim

Prolog's answers are correct and they logically follow from our definition of the **ancestor** and the **parent** relation.