A Puzzle

Knights and Liars: Knights always tell the truth; Liars always lie.

Zoe: "Mel is a liar"
Mel: "Neither I nor Zoe are liars"

Who's lying?

Why!? Logic!!

Zoe: "Mel is a liar"
Mel: "Neither I nor Zoe are liars"

$$z \Leftrightarrow \neg m \quad (1)$$
$$m \Leftrightarrow m \wedge z \quad (2)$$

z	m	(1)	(2)	(1) ∧ (2)
T	T	F	T	F
T	F	T	T	T
F	T	T	F	F
F	F	F	T	F

Why Programming?

It was a dark and stormy night. Three men, Alex, Bob & Carl, taking refuge from the rains, came to a hotel, one after another. One was a "knight": he always spoke the truth; another was a "liar": he always lied; and the third was a "knave": he alternated lying with speaking the truth.

When they arrived at the hotel, the manager was not at the desk. When she came to the front, she said she had only one room available, and will give it to the person who arrived first; the other two have to make do with the chairs in the lobby.

...

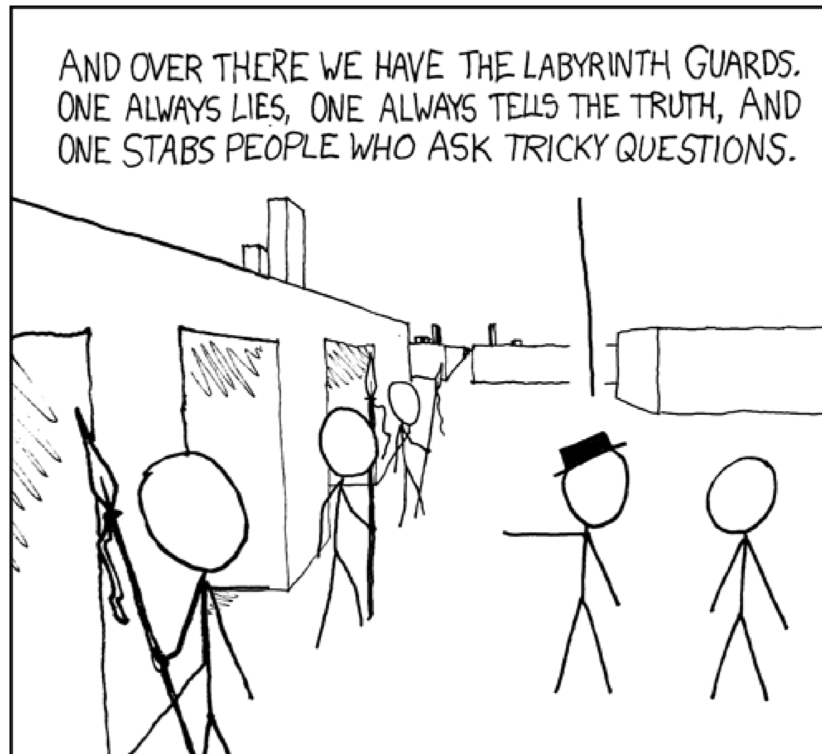
Why Programming? (Contd.)

...

The converation went like this:

Alex: *I came first.*
Bob: *No, he did not! I came first.*
Carl: *No, he did not! I came first.*
Alex: *No, he did not! I came first.*
Carl: *Well, Bob did not come second.*
Bob: *That's true!*

Who came first? Who came second? Who came third?



http://xkcd.com/246/

Logic and Programming

- Logic forms a formal foundation for describing relationships between entities.
- In many cases, we can infer interesting consequences from these relationships.
- When the inference procedure is simple enough, the descriptions of the relationships can be seen as *programs*.
- The same set of relationships can be described in many ways: each resulting in a different "program".
- Logic Programming: a framework for describing relationships such that inferences can be done *efficiently*.

Relations

- `parent(X, Y)`: `X` is a parent of `Y`.

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

- `male(X)`: `X` is a male.
  `female(X)`: `X` is a female.

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

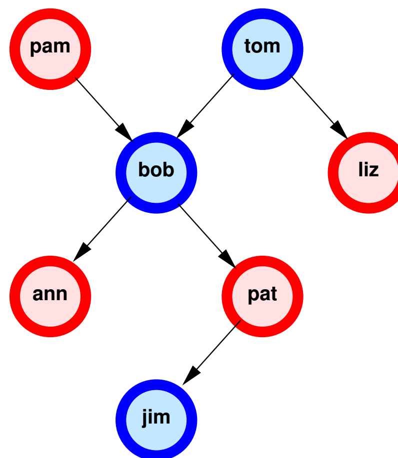
- `mother(X, Y)`: `X` is the mother of `Y`.

$$\forall X, Y. \quad \texttt{parent}(X, Y) \wedge \texttt{female}(X) \Rightarrow \texttt{mother}(X, Y)$$

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

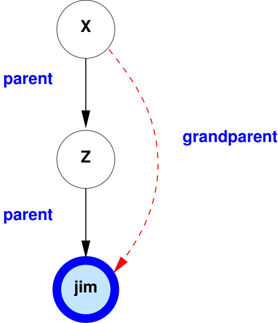
Representing relations

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

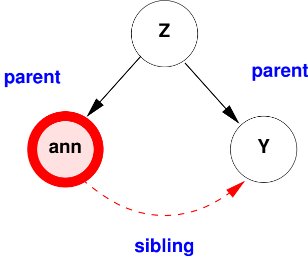


More Relations

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



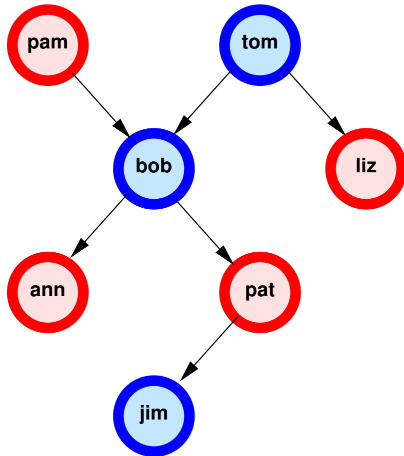
- `sibling(X,Y) :-  parent(Z,X), parent(Z,Y), X \= Y.`



More Relations

- `cousin(X,Y) :- .......`
- `greatgrandparent(X,Y) :- ........`
- `greatgreatgrandparent(X,Y) :- ........`
- `ancestor(X,Y) :- ...`

Computations in Prolog




```
|?- mother(M, bob).
    |?-  parent(M, bob), female(M).
         |?- M=pam, female(pam).
             M = pam

|?- father(M, bob)
    |?-  parent(M, bob), male(M)
         (i)   |?- M=pam, male(pam).
                   fail
         (ii) |?- M=tom, male(tom).
                   M = tom
```


Prolog Execution

Call: Call a predicate (invocation)
Exit: Return an answer to the caller
Fail: Return to caller with no answer
Redo: Try next path to find an answer

Syntax of Prolog Programs

- A program is a sequence of *clauses*.
- Each clause is of the form *head* **:-** *body* **.**
- Head is a *term*.
- Body is a comma-separated list of *terms*.
- Clause with an empty body is called a *fact*.
- A clause is also sometimes called a *rule*.

Terms

- Atomic data
- Variables
- Structures

Atomic Data

- **Numeric constants:** Integers, floating point numbers (e.g. `1024`, `-42`, `3.1415`, `6.023e23` ...)
- **Atoms:**
  - Strings of characters enclosed in single quotes (e.g. `'cram'`, `'Stony Brook'`)
  - Identifiers: sequence of letters, digits, underscore, beginning with a letter (e.g. `cram`, `r2d2`, `x_24`).

Variables

- Variables are denoted by identifiers beginning with an *Uppercase letter* or *underscore* (e.g. `X`, `Index`, `_param`).
- Underscore, by itself, represents an *anonymous variable*.
- Different occurrences of the same variable in a clause denote the same data.
- Each occurrence of an anonymous variable is treated as a different data.
- Variables are implicitly declared upon first use.
- Variables are not typed.
- Variables can be assigned only once, but that value can be further refined.

Structures

(We'll come to this topic later ...)

Meaning of Logic Programs

- Declarative Meaning: What are the logical consequences of a program?
- Procedural Meaning: For what values of the variables in the query can I prove the query?

Declarative Meaning

```
big(bear).                brown(bear).
big(elephant).            black(cat).
small(cat).               gray(elephant).

dark(Z) :- black(Z).      dark(Z) :- brown(Z).

dangerous(X) :- dark(X), big(X).
```

- Logical consequence of a program L is the smallest set such that
  - All facts of the program are in L
  - If $H :- B_1, B_2, \ldots, B_n$ is an instance of a clause in the program such that $B_1, B_2, \ldots, B_n$ are all in L,
    then H is also in L.
- For the above program we get `dark(cat)` and `dark(bear)` and consequently `dangerous(bear)`.

Procedural Meaning of Prolog

```
big(bear).        brown(bear).      dark(Z) :- black(Z).
big(elephant).    black(cat).       dark(Z) :- brown(Z).
small(cat).       gray(elephant).   dangerous(X) :- dark(X), big(X).
```

- A *query* is, in general, a conjunction of *goals*
- To prove $G_1, G_2, \ldots, G_n$:
  - Find a clause $H :- B_1, B_2, \ldots, B_k$ such that G_1 and H match.
  - Under that substitution for variables, prove $B_1, B_2, \ldots, B_k, G_2, \ldots, G_n$.

  If nothing is left to prove then the proof succeeds. If there are no more clauses to match, the proof fails.

Procedural Meaning of Prolog (Example)

```
big(bear).         brown(bear).       dark(Z) :- black(Z).
big(elephant).     black(cat).        dark(Z) :- brown(Z).
small(cat).        gray(elephant).    dangerous(X) :- dark(X), big(X).
```

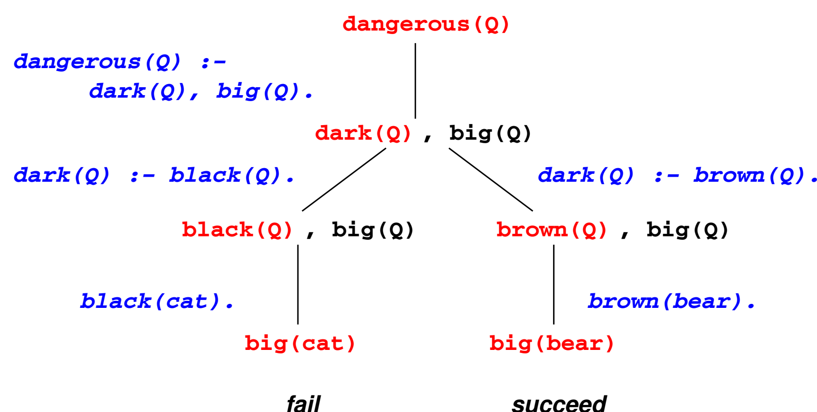
To prove `dangerous(Q)`:

1. Select `dangerous(X) :- dark(X), big(X)` and prove `dark(Q), big(Q)`.
2. To prove `dark(Q)` select the first clause of dark, i.e. `dark(Z) :- black(Z)`, and prove `black(Q), big(Q)`.
3. Now select the fact `black(cat)` and prove `big(cat)`. This proof fails!
4. Go back to step 2, and select the *second* clause of dark, i.e. `dark(Z) :- brown(Z)`, and prove `brown(Q), big(Q)`.
5. Now select `brown(bear)` and prove `big(bear)`.
6. Select the fact `big(bear)`.

   There is nothing left to prove, so the proof succeeds

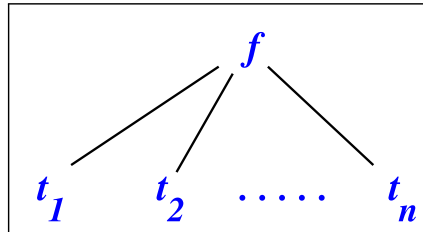
Derivations

```
big(bear).                brown(bear).
big(elephant).            black(cat).
small(cat).               gray(elephant).

dark(Z) :- black(Z).      dark(Z) :- brown(Z).

dangerous(X) :- dark(X), big(X).
```



Structures

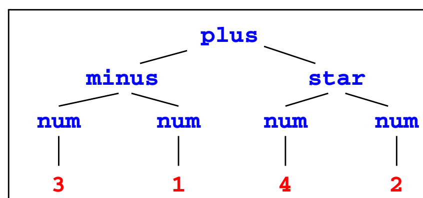
- If f is an identifier and $t_1, t_2, \ldots t_n$ are terms, then $f(t_1, t_2, \ldots t_n)$ is a term.



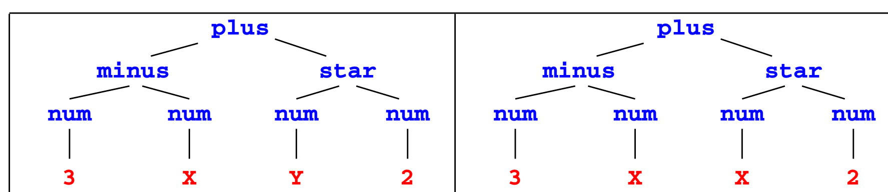
- In the above, f is called a *functor* and t_i is an *argument*.
- Structures are used to group related data items together (in some ways similar to `struct` in C and objects in Java).
- Structures are used to construct trees (and, as a special case, lists).

Trees

- Example: expression trees:
`plus(minus(num(3), num(1)), star(num(4), num(2)))`

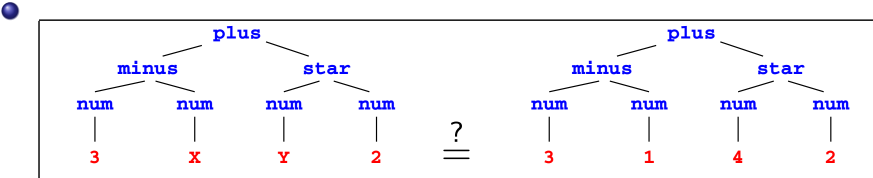


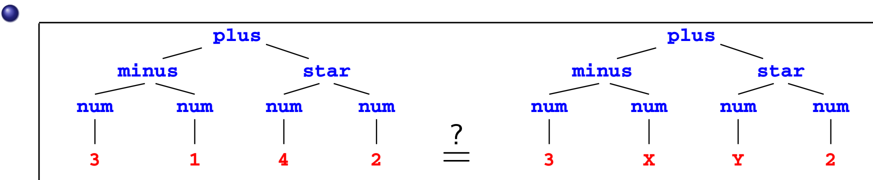
- Data structures may have variables. And the same variable may occur multiple times in a data structure.



Matching

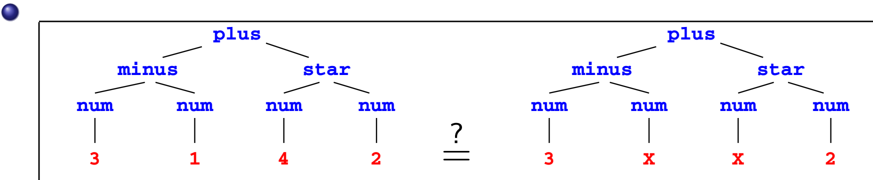
(We'll later use the word *unification* to make this more standard and rigorous)

- $t_1 = t_2$: find substitions for variables in t_1 and t_2 that make the two terms identical.
- 

Yes, with $X = 1$, $Y = 4$.
- 

Yes, with $X = 1$, $Y = 4$.

Matching (Contd.)

- 

No! X cannot be 1 and 4 at the same time.

Accessing arguments of a structure

- Matching is the predominant means for accessing a structures arguments.
- Let `date('Sep', 1, 2005)` be a structure used to represent dates, with the month, day and year as the three arguments (in that order).
- Then `date(M, D, Y) = date('Sep', 1, 2005)` makes $M =$ `'Sep'`, $D = 1$, $Y = 2005$.
- If we want to get only the day, we can write `date(_, D, _) = date('Sep', 1, 2005)`. Then we get $D = 1$.