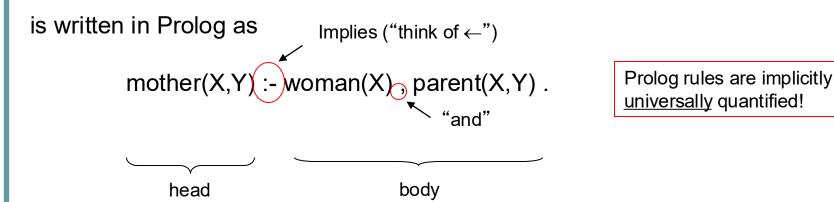
# Prolog Rules

Prolog <u>rules</u> are Horn clauses, but they are written "backwards", consider:

$$\forall X,Y[woman(X) \land parent(X,Y) \rightarrow mother(X,Y)]$$



You can think of a rule as introducing a new "fact" (the head), but the fact is defined in terms of a compound goal (the body). That is, predicates defined as rules are only true if the associated compound goal can be shown to be true.

# Prolog Rules

```
% a simple prolog program
woman(pam).
woman(liz).
woman(ann).
woman(pat).
                                                Queries:
man(tom).
                                                 ?- mother(pam,bob).
man (bob).
                                                ?- mother(Z,jim).
man(jim).
                                                 ?- mother(P,Q).
parent (pam, bob) .
parent (tom, bob) .
parent (tom, liz).
parent (bob, ann).
parent (bob, pat) .
parent(pat,jim).
mother(X,Y) := woman(X), parent(X,Y).
```

Demo of 'trace' predicate for mother.

# Prolog Rules

The same predicate name can be defined by multiple rules. Assume that our program looks like the following,

```
brother(fred, john).
sibling(X,Y) :- sister(X,Y).
sibling(X,Y) :- brother(X,Y).
```

Then our query,

```
?- sibling(fred,Q).
```

By trying the first rule and fail, backtracking to the second rule, trying that, and succeed.

# Another Simple Prolog Program

Consider the program relating humans to mortality:

```
mortal(X) :- human(X).
human(socrates).
```

We can now pose the query:

```
?- mortal(socrates).
```

True or false?

# Declarative vs. Procedural Meaning

When interpreting rules purely as Horn clause logic statement → <u>declarative</u>

When interpreting rules as "specialized queries" → procedural

Observation: We design programs with declarative meaning in our minds, but the execution is performed in a procedural fashion.

Consider:

mother(X,Y) := woman(X), parent(X,Y).

# Assignment

- Read Chap 20 in MPL
- Assignment #5 see BrightSpace

## Lists & Pattern Matching

- The <u>unification</u> operator: =/2
  - The expression A=B is true if A and B are terms and <u>unify</u> (look identical)

```
?- a = a.
true
?- a = b.
false
?- a = X.
X = a
?- X = Y.
true
```

### Lists & Pattern Matching

- Lists a convenient way to represent abstract concepts
  - Prolog has a special notation for lists.

```
[a]
[a,b,c]
[ ]
Empty
List
```

[ bmw, vw, mercedes ] [ chicken, turkey, goose ]

## Lists & Pattern Matching

Pattern Matching in Lists

But:

The Head-Tail Operator: [H|T]

```
?- [a,b,c] = [X|Y];
X = a
Y = [b,c]
?- [a] = [Q|P];
Q = a
P = []
```

#### Lists - the First Predicate

The predicate first/2: accept a list in the first argument and return the first element of the list in second argument.

```
first(List,E) :- List = [H|T], E = H;
```

#### Lists - the Last Predicate

The predicate last/2: accept a list in the first argument and return the last element of the list in second argument.

<u>Recursion</u>: there are always two parts to a recursive definition; the <u>base</u> and the <u>recursive step</u>.

```
last([A],A).
last([A|L],E) :- last(L,E).
```

### Lists - the Append Predicate

<u>The append/3 predicate:</u> accept two lists in the first two parameters, append the second list to the first and return the resulting list in the third parameter.

Hint: use recursion.

```
append([], List, List).
append([H|T], List, [H|Result]) :- append(T, List, Result).
```

#### Exercise: The halve/3 Predicate

 Design the predicate halve/3 that takes a list as its first argument and returns two lists each with half the elements of the original list (similar to the function halve we studied in Asteroid).

```
halve([1,2],[1],[2])
```

- halve([1],[1],[])
- halve([],[],[])