

# Prolog

Artificial Intelligence  
(Only for Reference)



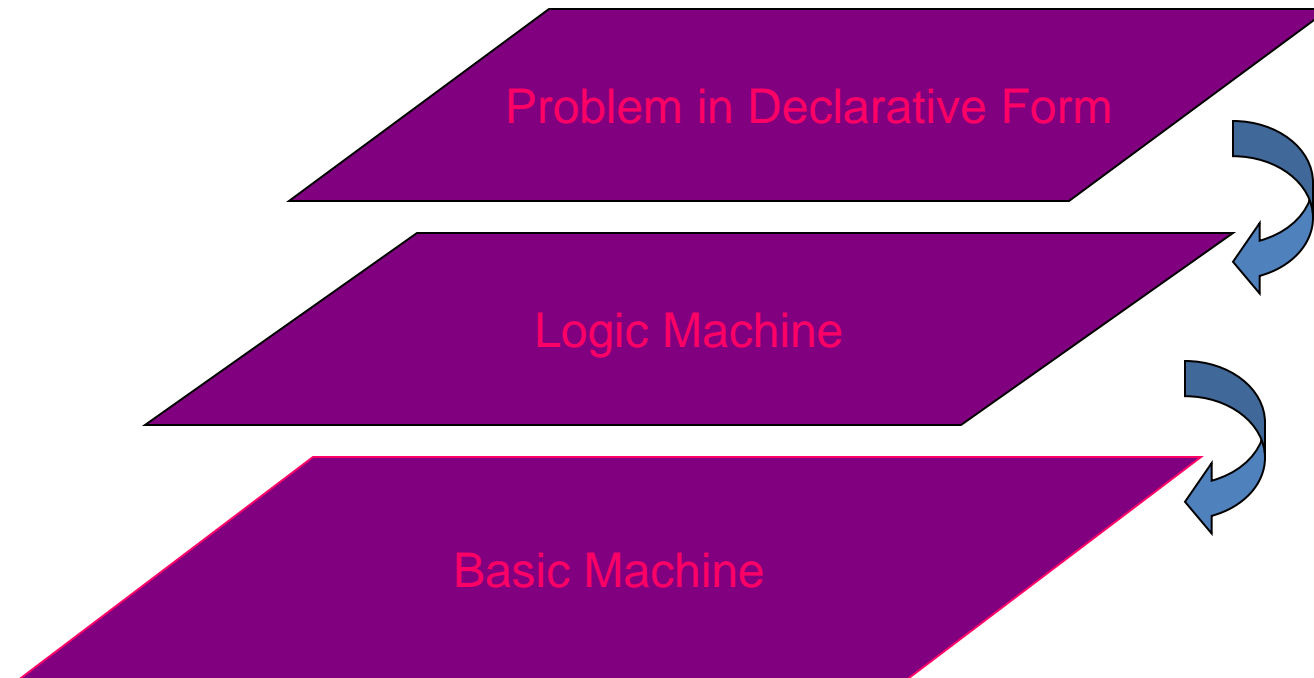
# Contents

- Introduction
- Syntax and meaning of Prolog Program
- Lists and its Operations
- Operators in Prolog
- Using structures in Prolog
- Controlling Backtracking
- I/O with Data file
- More built-in predicates



# Introduction

- PROgramming in LOGic
- Emphasis on *what* rather than *how*





## What is Prolog?

- Prolog (*programming in logic*) is a logic-based programming language: programs correspond to sets of logical formulas and the Prolog interpreter uses logical methods to resolve queries.
- Prolog is a *declarative* language: you specify *what* problem you want to solve rather than *how* to solve it.
- Prolog is very useful in *some* problem areas, such as artificial intelligence, natural language processing, databases, . . . , but pretty useless in others, such as for instance graphics or numerical algorithms.
- The objective of this first lecture is to introduce you to the most basic concepts of the Prolog programming language.



# Characteristics

- Prolog approximates first-order logic.
- Every program is a set of Horn clauses.
- Inference is by resolution.
- Search is by backtracking with unification.
- Basic data structure is term or tree.
- Variables are unknowns not locations.
- Prolog does not distinguish between inputs and outputs. It solves relations/predicates.

# Prolog's strong and weak points

- Assists thinking in terms of *objects* and *entities*
- Not good for *number crunching*
- Useful applications of Prolog in
  - *Expert Systems* (Knowledge Representation and Inferencing)
  - *Natural Language Processing*
  - *Relational Databases*

# What is Prolog?

- Prolog is the most widely used language to have been inspired by logic programming research. Some features:
- Prolog uses logical variables. These are not the same as variables in other languages. Programmers can use them as 'holes' in data structures that are gradually filled in as computation proceeds.

## ...More

- Unification is a built-in term-manipulation method that passes parameters, returns results, selects and constructs data structures.
- Basic control flow model is backtracking.
- Program clauses and data have the same form.
- The relational form of procedures makes it possible to define 'reversible' procedures.



## ...More

- Clauses provide a convenient way to express case analysis and nondeterminism.
- Sometimes it is necessary to use control features that are not part of 'logic'.
- A Prolog program can also be seen as a relational database containing rules as well as facts.



# What a program looks like

```
/* At the Zoo */
```

```
elephant(george).  
elephant(mary).
```

```
panda(chi_chi).  
panda(ming_ming).
```

```
dangerous(X) :- big_teeth(X).  
dangerous(X) :- venomous(X).
```

```
guess(X, tiger) :- stripey(X), big_teeth(X), isaCat(X).  
guess(X, koala) :- arboreal(X), sleepy(X).  
guess(X, zebra) :- stripey(X), isaHorse(X).
```

# Prolog is a 'declarative' language

- Clauses are statements about what is true about a problem, instead of instructions how to accomplish the solution.
- The Prolog system uses the clauses to work out how to accomplish the solution by searching through the space of possible solutions.
- Not all problems have pure declarative specifications. Sometimes extralogical statements are needed.

# Example: Concatenate lists a and b

In an imperative language

```
list procedure cat(list a, list b)
{
  list t = list u = copylist(a);
  while (t.tail != nil) t = t.tail;
  t.tail = b;
  return u;
}
```

In a functional language

```
cat(a,b) ≡
  if b = nil then a
  else cons(head(a),
    cat(tail(a),b))
```

In a declarative language

```
cat([], Z, Z).
cat([H|T], L, [H|Z]) :- cat(T, L, Z).
```



# A Prolog Program

- The program, sometimes called Database is a text file (\*.pl) that contain the facts and rules. It contains all the relations needed to define the problem.
- When you launch a program you are in query mode represented by the ? – prompt. In query mode you ask questions about the relations described in the program.
- When Prolog is launched the ?- should appear meaning you are in query mode. In *SWI-Prolog* you can load a program by typing the command **[file]**. when the file containing your program is file.pl. When you have done this you can use all the facts and rules that are contained in the program.

# SWI-Prolog

- We will use SWI-Prolog for the Prolog programming assignment
  - <http://www.swi-prolog.org/>
- After the installation, try the example program

```
?- [likes].
```

```
% likes compiled 0.00 sec, 2,148 bytes
```

Load example likes.pl

```
Yes
```

```
?- likes(sam, curry).
```

This goal cannot be proved, so it assumed to be false (This is the so called *Close World Assumption*)

```
No
```

```
?- likes(sam, X).
```

```
X = dahl ;
```

```
X = tandoori ;
```

```
X = kurma ;
```



Asks the interpreter to find more solutions



# Example

## Program

```
location(desk, office).  
location(apple, kitchen).  
location	flashlight, desk).  
location('washing machine', cellar).  
location(nani, 'washing machine').  
location(broccoli, kitchen).  
location(crackers, kitchen).  
location(computer, office).
```

## Query

```
?- location(apple, kitchen).  
yes
```



# Prolog Programming Model

- A program is a *database of (Horn) clauses* that are assumed to be true.
- The clauses in a Prolog database are either *facts* or *rules*. Each ends with a period.
- A *fact* is a clause without a right-hand side.
  - rainy(asheville).
- A *rule* has a right-hand side.
  - snowy(X) :- rainy(X), cold(X).
- The token :- is the implication symbol.
- The comma (,) indicates “and”



# Prolog Clauses

- A *query* or *goal* is a clause with an empty left-hand side. Queries are given to the prolog interpreter to initiate execution of a program.
- Each clauses is composed of *terms*:
  - *Constants* (atoms, that are identifier starting with a lowercase letter, numbers, punctuation, and quoted strings)
    - *E.g.* `curry`, `4.5`, `+`, `'Hi, Mom'`
  - *Variables* (identifiers starting with an uppercase letter)
    - *E.g.* `Food`
  - *Structures* (predicates or data structures)
    - *E.g.* `indian(Food)`, `date(Year,Month,Day)`



# Principle of Resolution

- Prolog execution is based on the *principle of resolution*
  - If  $C_1$  and  $C_2$  are Horn clauses and the head of  $C_1$  matches one of the terms in the body of  $C_2$ , then we can replace the term in  $C_2$  with the body of  $C_1$
- For example,
  - $C_1: \text{likes}(\text{sam}, \text{Food}) \text{ :- indian}(\text{Food}), \text{mild}(\text{Food}).$
  - $C_2: \text{indian}(\text{dahl}).$
  - $C_3: \text{mild}(\text{dahl}).$
  - We can replace the first and the second terms in  $C_1$  by  $C_2$  and  $C_3$  using the principle of resolution (after *instantiating variable Food to dahl*)
  - Therefore,  $\text{likes}(\text{sam}, \text{dahl})$  can be proved



# Another Example

```
takes(jane_doe, his201).  
takes(jane_doe, cs245).  
takes(ajit_chandra, art302).  
takes(ajit_chandra, cs254).  
classmates(X, Y) :- takes(X, Z), takes(Y, Z).
```

via resolution, yields the new rule:

```
classmates(jane_doe, Y) :- takes(Y, cs254).
```

# Unification

- Prolog associates variables and values using a process known as *unification*
  - Variable that receive a value are said to be *instantiated*
- Unification rules
  - A constant unifies only with itself
  - Two structures unify if and only if they have the same functor and the same number of arguments, and the corresponding arguments unify recursively
  - A variable unifies to with anything



# Equality

- Equality is defined as *unifiability*
  - An equality goal is using an infix predicate =
- For instance,
  - `?- dahl = dahl.`  
Yes
  - `?- dahl = curry.`  
No
  - `?- likes(Person, dahl) = likes(sam, Food).`  
Person = sam  
Food = dahl ;  
No
  - `?- likes(Person, curry) = likes(sam, Food).`  
Person = sam  
Food = curry ;  
No



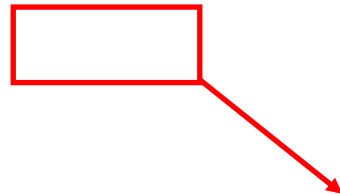
# Equality

- What is the results of

```
?- likes(Person, Food) = likes(sam, Food) .
```

```
Person = sam  
Food = _G158 ;
```

No



Internal Representation for an  
uninstantiated variable  
*Any instantiation proves the equality*



# Execution Order

- Prolog searches for a resolution sequence that satisfies the goal
- In order to satisfy the logical predicate, we can imagine two search strategies:
  - *Forward chaining*, derived the goal from the axioms
  - *Backward chaining*, start with the goal and attempt to resolve them working backwards
- Backward chaining is usually more efficient, so it is the mechanism underlying the execution of Prolog programs
  - Forward chaining is more efficient when the number of facts is small and the number of rules is very large

# Backward Chaining in Prolog

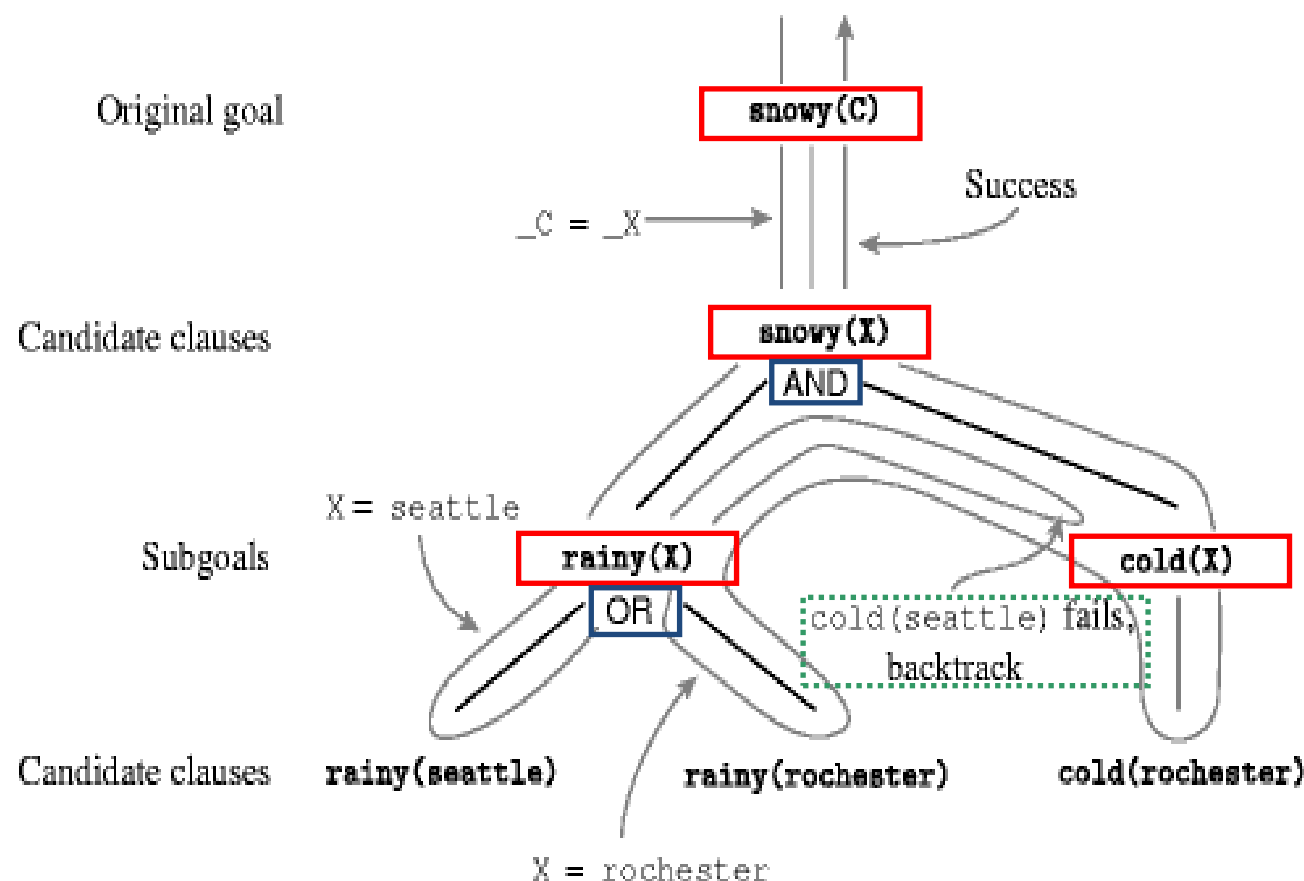
- Backward chaining follows a classic depth-first backtracking algorithm

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X)
```

- Example

– Goal:

`Snowy(C)`







# Depth-first backtracking

- The search for a resolution is ordered and depth-first
  - The behavior of the interpreter is predictable
- Ordering is fundamental in recursion
  - Recursion is again the basic computational technique, as it was in functional languages
  - Inappropriate ordering of the terms may result in non-terminating resolutions (infinite regression)
  - For example: Graph

```
edge(a,b) . edge(b, c) . edge(c, d) .  
edge(d,e) . edge(b, e) . edge(d, f) .  
path(X, X) .  
path(X, Y) :- edge(Z, Y), path(X, Z) .
```

Correct



# Depth-first backtracking

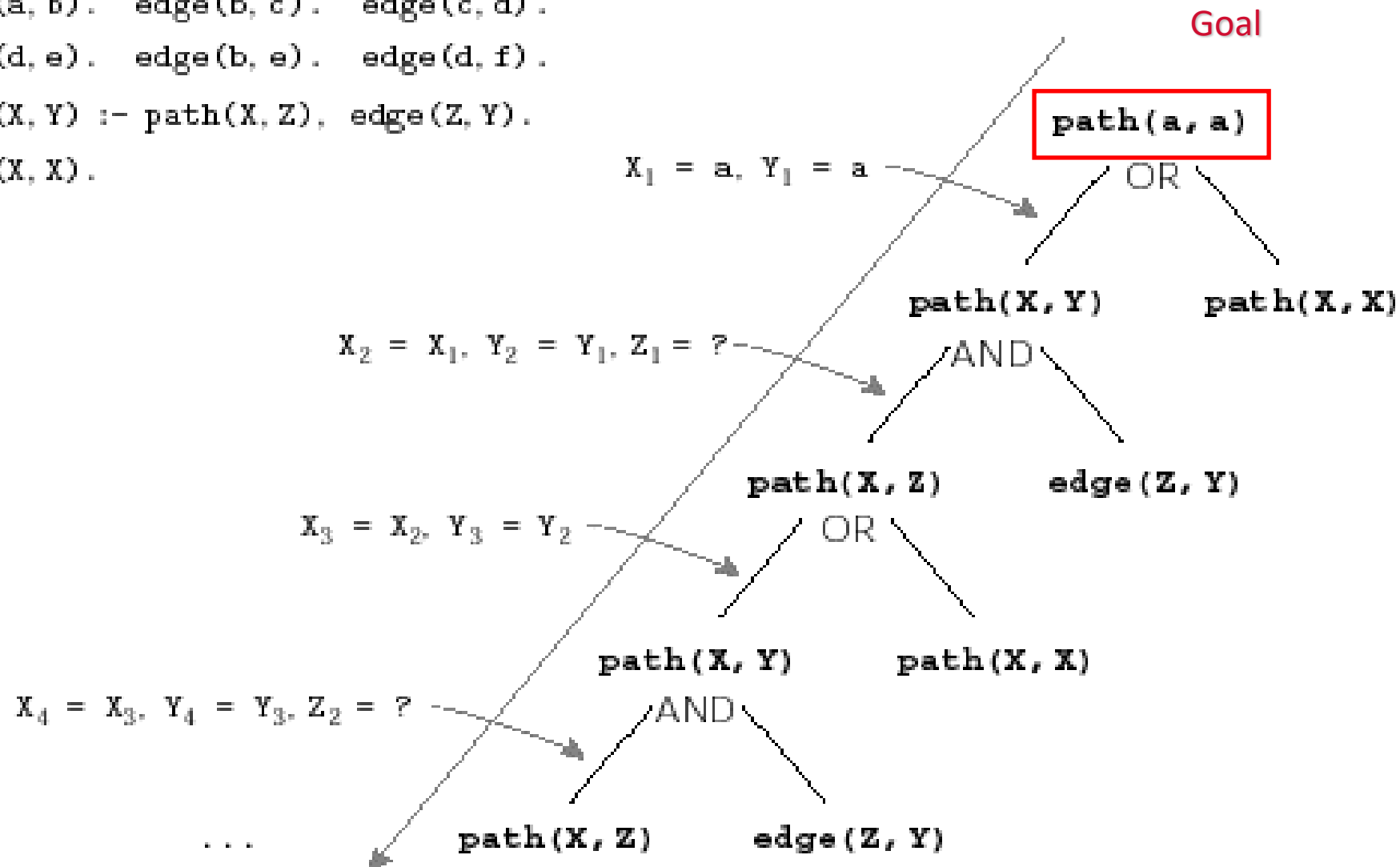
- The search for a resolution is ordered and depth-first
  - The behavior of the interpreter is predictable
- Ordering is fundamental in recursion
  - Recursion is again the basic computational technique, as it was in functional languages
  - Inappropriate ordering of the terms may result in non-terminating resolutions (infinite regression)
  - For example: Graph

```
edge(a,b) . edge(b, c) . edge(c, d) .  
edge(d,e) . edge(b, e) . edge(d, f) .  
path(X, Y) :- path(X, Z), edge(Z, Y) .  
path(X, X) .
```

Incorrect

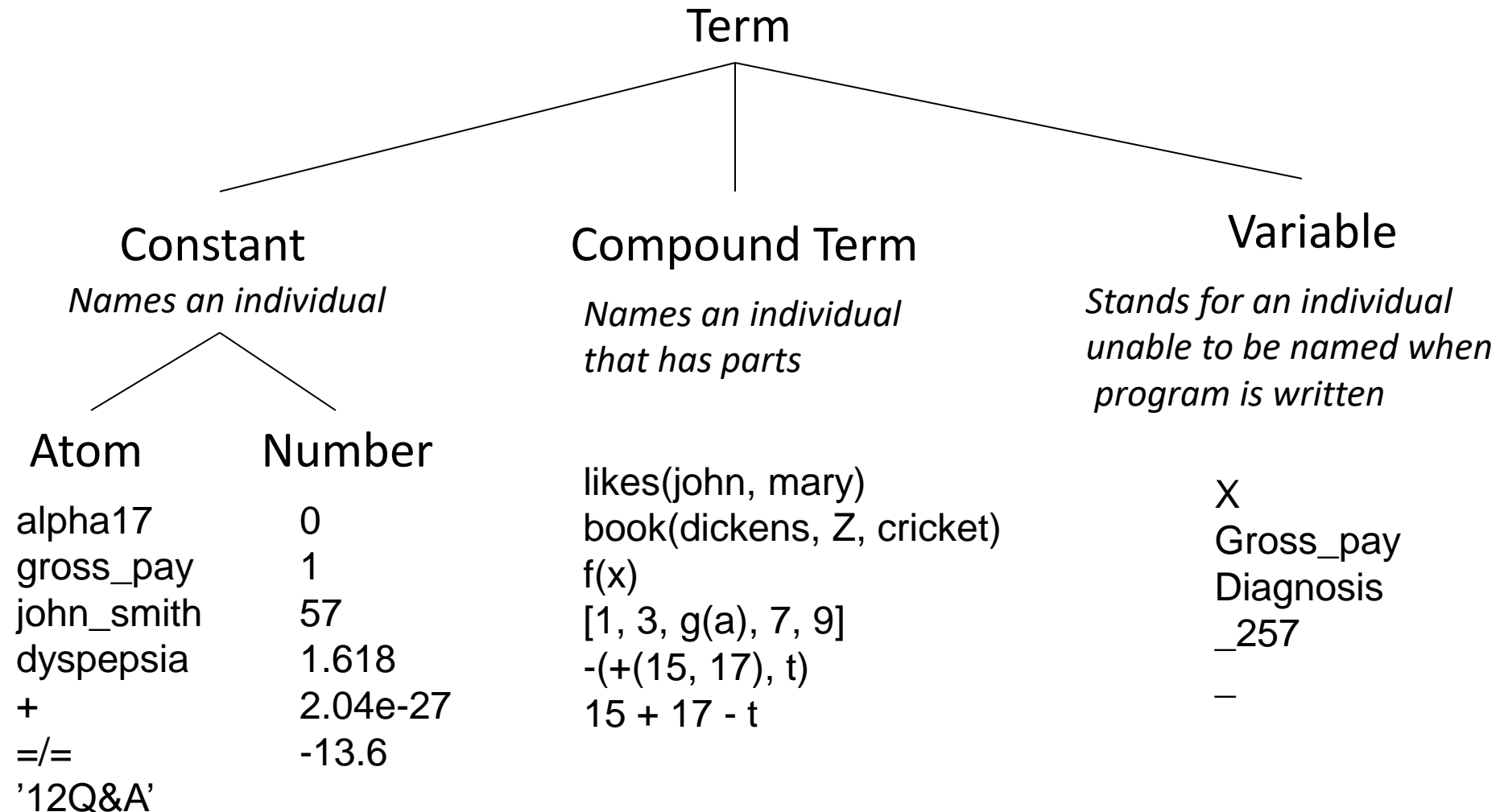
# Infinite Regression

```
edge(a, b).  edge(b, c).  edge(c, d).
edge(d, e).  edge(b, e).  edge(d, f).
path(X, Y) :- path(X, Z), edge(Z, Y).
path(X, X).
```





# Complete Syntax of Terms



# Anatomy of a Program

- Last week I told you that Prolog programs are made up of **facts** and **rules**.
- A **fact** asserts some property of an object, or relation between two or more objects.

e.g. `parent(jane, alan) .`

Can be read as “Jane is the parent of Alan.”

- **Rules** allow us to infer that a property or relationship holds based on preconditions.

e.g. `parent(X, Y) :- mother(X, Y) .`

= “Person X is the parent of person Y **if** X is Y’s mother.”

# Predicate Definitions

- Both facts and rules are **predicate definitions**.
- *'Predicate'* is the name given to the word occurring before the bracket in a fact or rule:

`parent(jane, alan) .`


**Predicate name**

- By defining a predicate you are specifying which information needs to be known for the property denoted by the predicate to be true.

# Clauses

- Predicate definitions consist of *clauses*.  
= An individual definition (whether it be a fact or rule).

e.g. `mother(jane, alan) . = Fact`  
`parent(P1, P2) :- mother(P1, P2) . = Rule`



- A clause consists of a *head*
- and sometimes a *body*.
  - Facts don't have a body because they are always true.

# Arguments

- A predicate head consists of a *predicate name* and sometimes some *arguments* contained within brackets and separated by commas.

`mother(jane, alan) .`

**Predicate name**      **Arguments**



- A body can be made up of any number of *subgoals* (calls to other predicates) and *terms*.
- Arguments also consist of *terms*, which can be:
  - *Constants* e.g. jane,
  - *Variables* e.g. Person1, or
  - *Compound terms* (explained in later lectures).



# Terms: Constants

Constants can either be:

- Numbers:
  - integers are the usual form (e.g. 1, 0, -1, etc), but
  - floating-point numbers can also be used (e.g. 3.0E7)
- Symbolic (non-numeric) constants:
  - always start with a lower case alphabetic character and contain any mixture of letters, digits, and underscores (but no spaces, punctuation, or an initial capital).
    - e.g. `abc`, `big_long_constant`, `x4_3t`).
- String constants:
  - are anything between single quotes e.g. 'Like this'.

# Terms: Variables

- Variables always start with an upper case alphabetic character or an underscore.
- Other than the first character they can be made up of any mixture of letters, digits, and underscores.

e.g. `X`, `ABC`, `_89two5`, `_very_long_variable`

- There are **no “types”** for variables (or constants) – a variable can take any value.
- All Prolog variables have a **“local”** scope:
  - they only keep the same value within a clause; the same variable used outside of a clause does not inherit the value (this would be a “global” scope).

# Naming tips

- Use real English when naming predicates, constants, and variables.

e.g. “John wants to help Somebody.”

Could be: `wants(john, to_help, Somebody) .`

Not: `x87g(j, _789) .`

- Use a **Verb Subject Object** structure:

`wants(john, to_help) .`

- **BUT** do not assume Prolog Understands the meaning of your chosen names!
  - You create meaning by specifying the body (i.e. preconditions) of a clause.

# Using predicate definitions

- Command line programming is tedious

e.g. | `?- write('What is your name?'), nl, read(X), write('Hello '), write(X).`

- We can define predicates to automate commands:

`greetings:-`

```
write('What is your name?'),  
nl,  
read(X),  
write('Hello '),  
write(X).
```

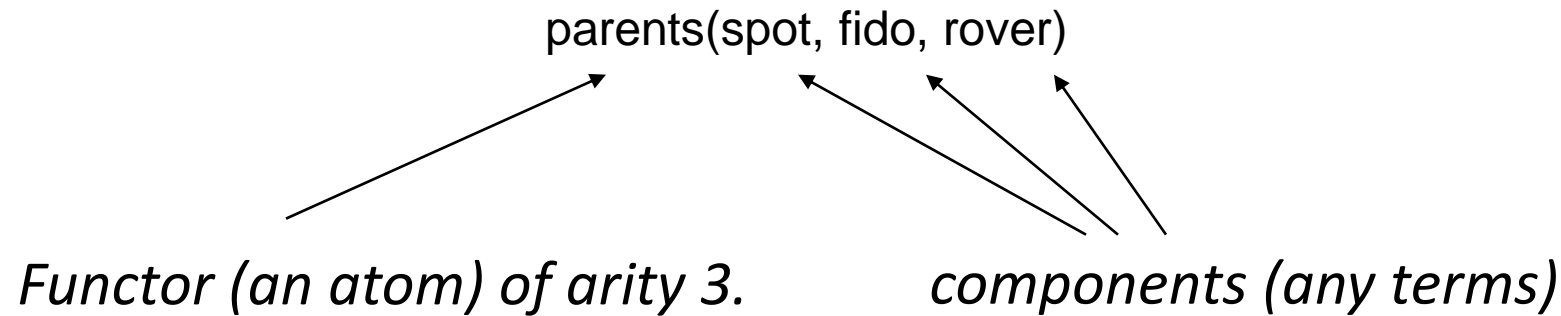
Prolog Code

```
| ?- greetings.  
What is your name?  
|: tim.  
Hello tim  
X = tim ?  
yes
```

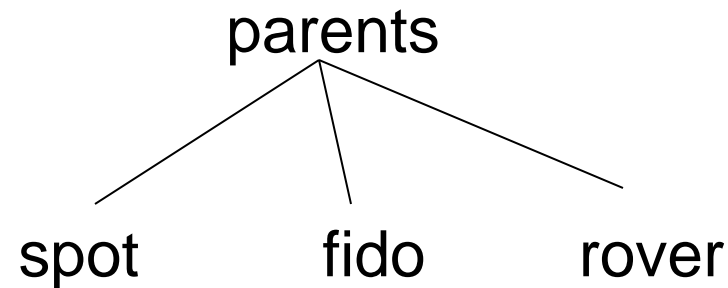
Terminal

# Compound Terms

*The parents of Spot are Fido and Rover.*



It is possible to depict the term as a tree:





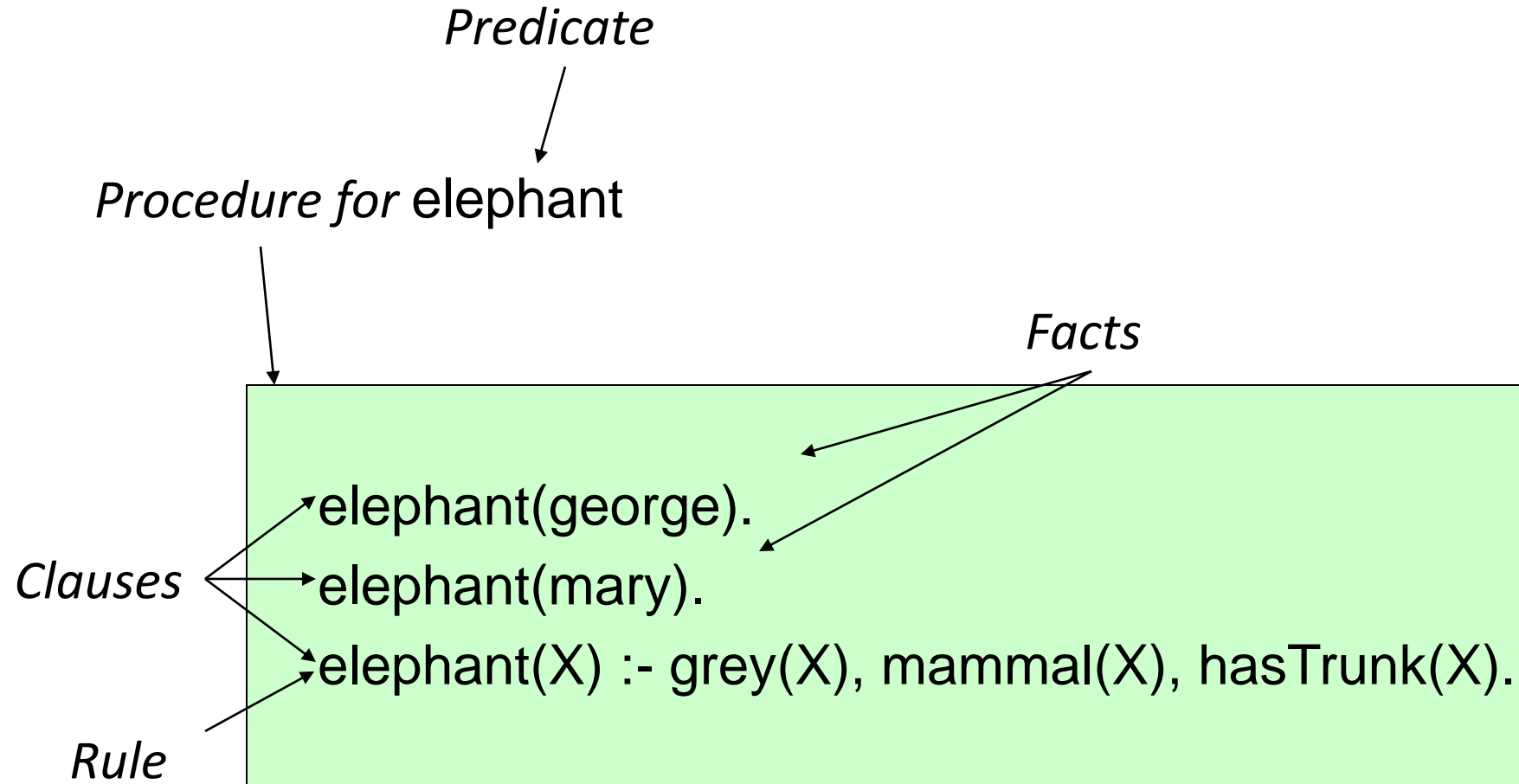
# Structure of Programs

- Programs consist of procedures.
- Procedures consist of clauses.
- Each clause is a fact or a rule.
- Programs are executed by posing queries.

*An example...*

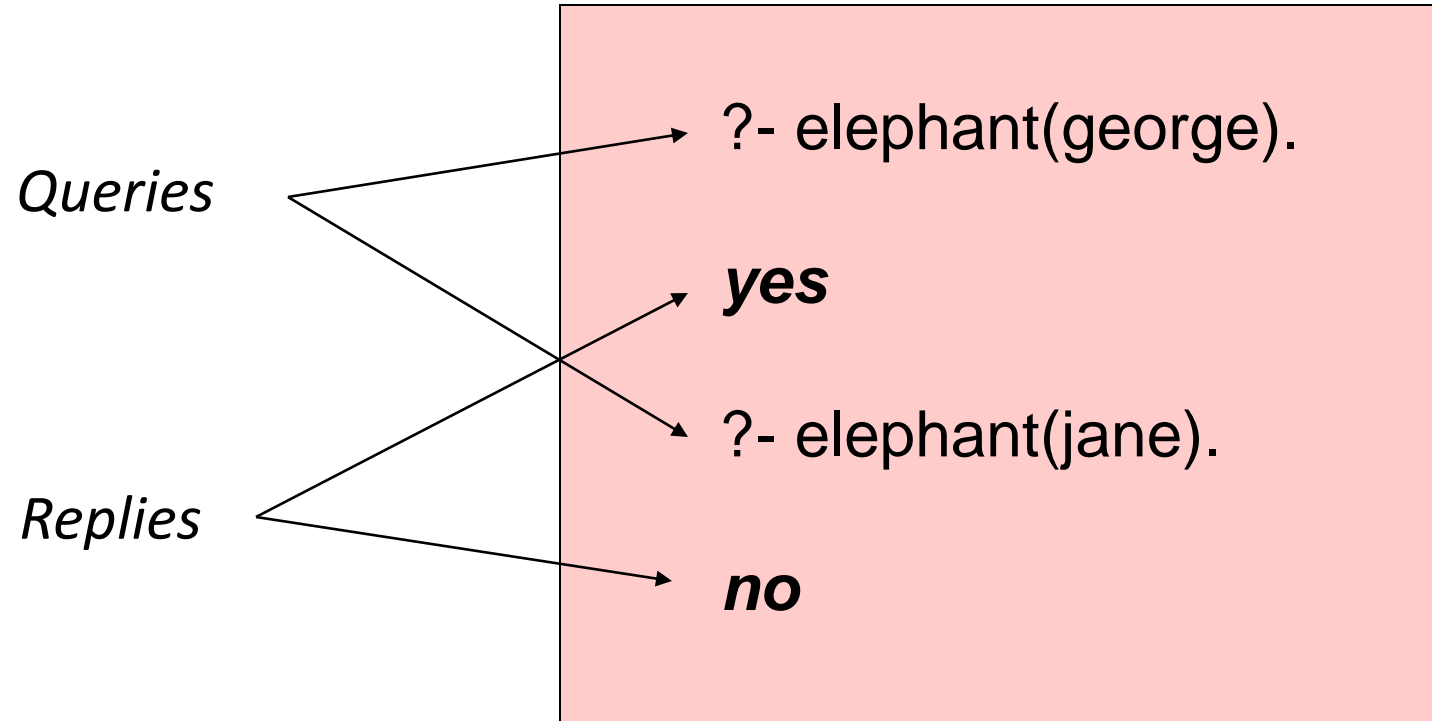


# Example





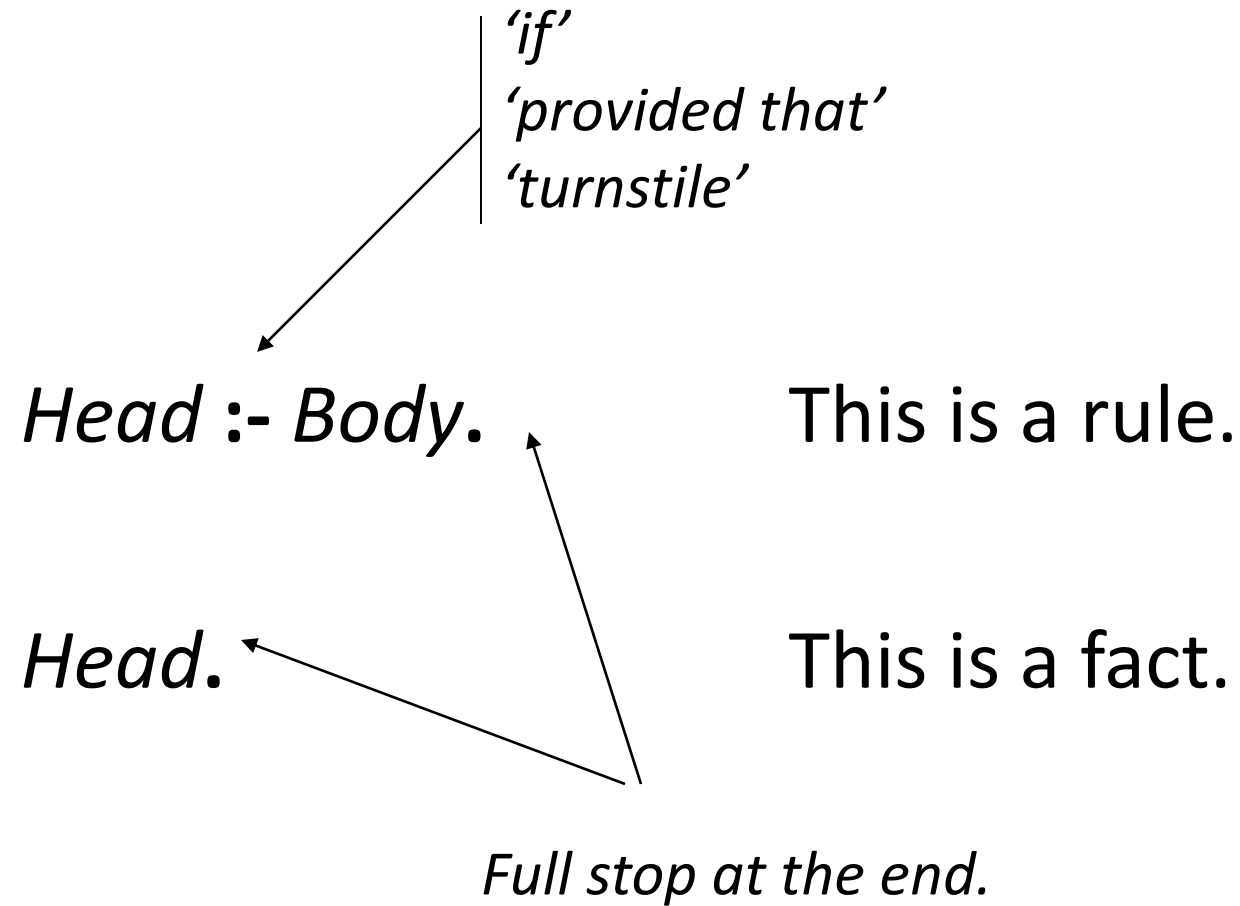
# Example



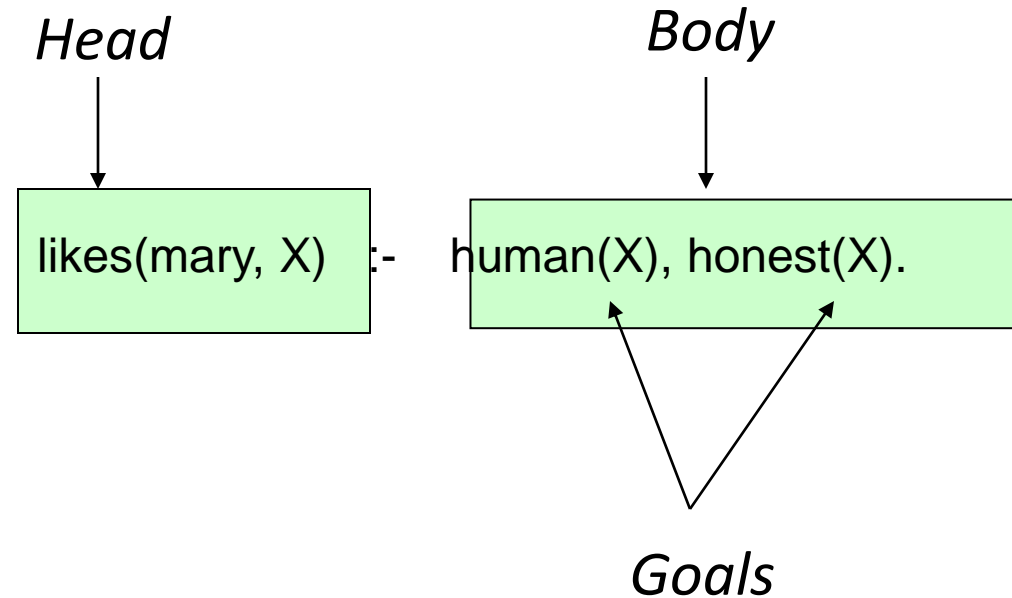




# Clauses: Facts and Rules



# Body of a (rule) clause contains goals.



Exercise: Identify all the parts of Prolog text you have seen so far.



# Interpretation of Clauses

Clauses can be given a declarative reading or a procedural reading.

*Form of clause:*

$H \text{ :- } G_1, G_2, \dots, G_n.$

*Declarative reading:*

“That  $H$  is provable follows from goals  $G_1, G_2, \dots, G_n$  being provable.”

*Procedural reading:*

“To execute procedure  $H$ , the procedures called by goals  $G_1, G_2, \dots, G_n$  are executed first.”



male(bertram).

male(percival).

female(lucinda).

female(camilla).

pair(X, Y) :- male(X), female(Y).

?- pair(percival, X).

?- pair(apollo, daphne).

?- pair(camilla, X).

?- pair(X, lucinda).

?- pair(X, X).

?- pair(bertram, lucinda).

?- pair(X, daphne).

?- pair(X, Y).

# Facts

- *John likes Mary*
  - *like(john,mary)*
- Names of relationship and objects must begin with a lower-case letter.
- Relationship is written *first* (typically the *predicate* of the sentence).
- *Objects* are written separated by commas and are enclosed by a pair of round brackets.
- The full stop character ‘.’ must come at the end of a fact.



## More facts

<b>Predicate</b>	<b>Interpretation</b>
valuable(gold)	Gold is valuable.
owns(john,gold)	John owns gold.
father(john,mary)	John is the father of Mary
gives (john,book,mary)	John gives the book to Mary



# Questions

- *Questions* based on facts
- Answered by *matching*

Two facts *match* if their predicates are same (spelt the same way) and the arguments each are same.

- If matched, prolog answers *yes*, else *no*.
- *No* does not mean falsity.

## Facts

A little Prolog program consisting of four *facts*:

```
bigger(elephant, horse).
```

```
bigger(horse, donkey).
```

```
bigger(donkey, dog).
```

```
bigger(donkey, monkey).
```

---



## Queries

After compilation we can *query* the Prolog system:

```
?- bigger(donkey, dog) .
```

Yes

```
?- bigger(monkey, elephant) .
```

No

---

## A Problem

The following query does not succeed!

```
?- bigger(elephant, monkey).
```

No

The *predicate* bigger/2 apparently is not quite what we want.

What we'd really like is the transitive closure of bigger/2. In other words: a predicate that succeeds whenever it is possible to go from the first animal to the second by iterating the previously defined facts.

---



# Programming in Prolog

- In Prolog program, we declare **facts** describing explicit relationships between objects and properties objects might have (e.g. Tulika likes ice-cream, Hair is black, Maruti is a company, Mini is a cat, Zahir teaches Salim)
- We define **rules** defining implicit relationships between objects (e.g. brother relationship) and/or rules defining implicit object properties (e.g. A is a child of B if B is parent of A).
- One then uses the system to generate queries by asking questions about relationships between objects and/or about object properties (e.g. does Tulika like ice-cream? Rohit is parent of whom?)

# Facts

- Facts are properties of objects, or relationships between objects
- “Mahi has phone number 1122334433”, is written in prolog as:  
phoneno(mahi,1122334433).

# Facts

- Names of properties/relationships begin with lower case letters.
- The relationship name appears as the first term.
- Objects appear as comma separated arguments within parenthesis.
- A period “.” must end a fact.
- Objects also begin with lower case letters. They also can begin with digits (like 1234) and can be strings of characters enclosed in quotes. e.g., color(rose,'red')
- phoneno(mahi,1122334433). is also called as predicate or clause.

# Facts about a hypothetical Department of CE/IT

- %teaches(X,Y): person X teaches in course Y
  - teaches(sudhir,course001).
  - teaches(tapas,course002).
  - teaches(pravin,course003).
  - teaches(pranab,course001).
- %studies(X,Y): student X studies in course Y
  - studies(suparna,course001).
  - studies(santanu,course003).
  - studies(subir,course002).
  - studies(swarup,course003).
  - studies(jay,course003).
  - studies(jayvir,course001).
- Together these facts will form Prolog's database.

# Cut

- The cut, in Prolog, is a goal, written as !, which always succeeds, but cannot be backtracked past. It is used to prevent unwanted backtracking, for example, to prevent extra solutions being found by Prolog.
- This is a way of saying: "if the first rule succeeds, use it and don't try the second rule. (Otherwise, use the second rule.)"



# Maximum/Minimum

- $\text{max2}(X, Y, X) :- X > Y.$
- $\text{max2}(X, Y, Y) :- X \leq Y.$
- $\text{Max}(X, Y, X) :- X \geq Y, !$
- $\text{Max}(X, Y, Y).$
- Reformulation:
- $\text{Max\_cal}(X, Y, Z) :- X \geq Y, !, Z = X; Z = Y.$



# Recursion

---

```
factorial(0,1) .  
factorial(N,F) :-  
    N>0,  
    N1 is N-1,  
    factorial(N1,F1),  
    F is N * F1.  
fib(0, 0) .  
fib(X, Y) :-  
    X > 0,  
    fib(X, Y, _) .  
fib(1, 1, 0) .  
fib(X, Y1, Y2) :-  
    X > 1,  
    X1 is X - 1,  
    fib(X1, Y2, Y3),  
    Y1 is Y2 + Y3.
```

# Inference Engine

- A program's protocol for navigating through the rules and data in a knowledge system in order to solve the problem. The major task of the **inference engine** is to select and then apply the most appropriate rule at each step as the expert system runs, which is called rule-based reasoning.



# Inference Engine

- **Inference engine** is one of the basic components of an expert system that carries out reasoning whereby the expert system reaches a solution. It matches the rules provided in the rule base with the facts contained in the database.

# Inference Engine

- An **inference engine** is a computer program that applies artificial intelligence to try to obtain answers or responses to queries from a knowledge base.



# Inference Engine

- An **Inference Engine** is a tool from artificial intelligence.
- The first **inference engines** were components of expert systems.
- The typical expert system consisted of a knowledge base and an **inference engine**.
- The **inference engine** applied logical rules to the knowledge base and deduced new knowledge.
- **Inference engines** work primarily in one of two modes:
- forward chaining and backward chaining.
- **Forward chaining** starts with the known facts and asserts new facts.
- **Backward chaining** starts with goals, and works backward to determine what facts must be asserted so that the goals can be achieved.



# Prolog-Inference Engine

- Prolog has a built-in backward chaining inference engine which can be used to partially implement some expert systems.
- Prolog rules are used for the knowledge representation, and the Prolog inference engine is used to derive conclusions.
- Other portions of the system, such as the user interface, must be coded using Prolog as a programming language.
- The Prolog inference engine does simple backward chaining.
- Each rule has a goal and a number of sub-goals.
- The Prolog inference engine either proves or disproves each goal. There is no uncertainty associated with the results.

# SWI-Prolog

- We will use SWI-Prolog for the Prolog programming assignments
  - <http://www.swi-prolog.org/>
- After the installation, try the example program

```
?- [likes].
```

```
% likes compiled 0.00 sec, 2,148 bytes
```

Load example likes.pl

```
Yes
```

```
?- likes(sam, curry).
```

This goal cannot be proved, so it assumed to be false (This is the so called *Close World Assumption*)

```
No
```

```
?- likes(sam, X).
```

```
X = dahl ;
```

```
X = tandoori ;
```

```
X = kurma ;
```



Asks the interpreter to find more solutions



# Principle of Resolution

- Prolog execution is based on the *principle of resolution*
  - If  $C_1$  and  $C_2$  are Horn clauses and the head of  $C_1$  matches one of the terms in the body of  $C_2$ , then we can replace the term in  $C_2$  with the body of  $C_1$
- For example,
  - $C_1$ : `likes(sam, Food) :- indian(Food), mild(Food).`
  - $C_2$ : `indian(dahl).`
  - $C_3$ : `mild(dahl).`
  - We can replace the first and the second terms in  $C_1$  by  $C_2$  and  $C_3$  using the principle of resolution (after *instantiating* variable `Food` to `dahl`)
  - Therefore, `likes(sam, dahl)` can be proved





# Equality

- Equality is defined as *unifiability*
  - An equality goal is using an infix predicate =
- For instance,
  - `?- dahl = dahl.`  
Yes
  - `?- dahl = curry.`  
No
  - `?- likes(Person, dahl) = likes(sam, Food).`  
Person = sam  
Food = dahl ;  
No
  - `?- likes(Person, curry) = likes(sam, Food).`  
Person = sam  
Food = curry ;  
No

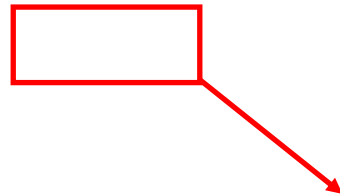


# Equality

- What is the results of

`?- likes(Person, Food) = likes(sam, Food) .`

`Person = sam`  
`Food = _G158 ;`



No

Internal Representation for an  
uninstantiated variable  
*Any instantiation proves the equality*



# Execution Order

- Prolog searches for a resolution sequence that satisfies the goal
- In order to satisfy the logical predicate, we can imagine two search strategies:
  - *Forward chaining*, derived the goal from the axioms
  - *Backward chaining*, start with the goal and attempt to resolve them working backwards
- Backward chaining is usually more efficient, so it is the mechanism underlying the execution of Prolog programs
  - Forward chaining is more efficient when the number of facts is small and the number of rules is very large

# Backward Chaining in Prolog

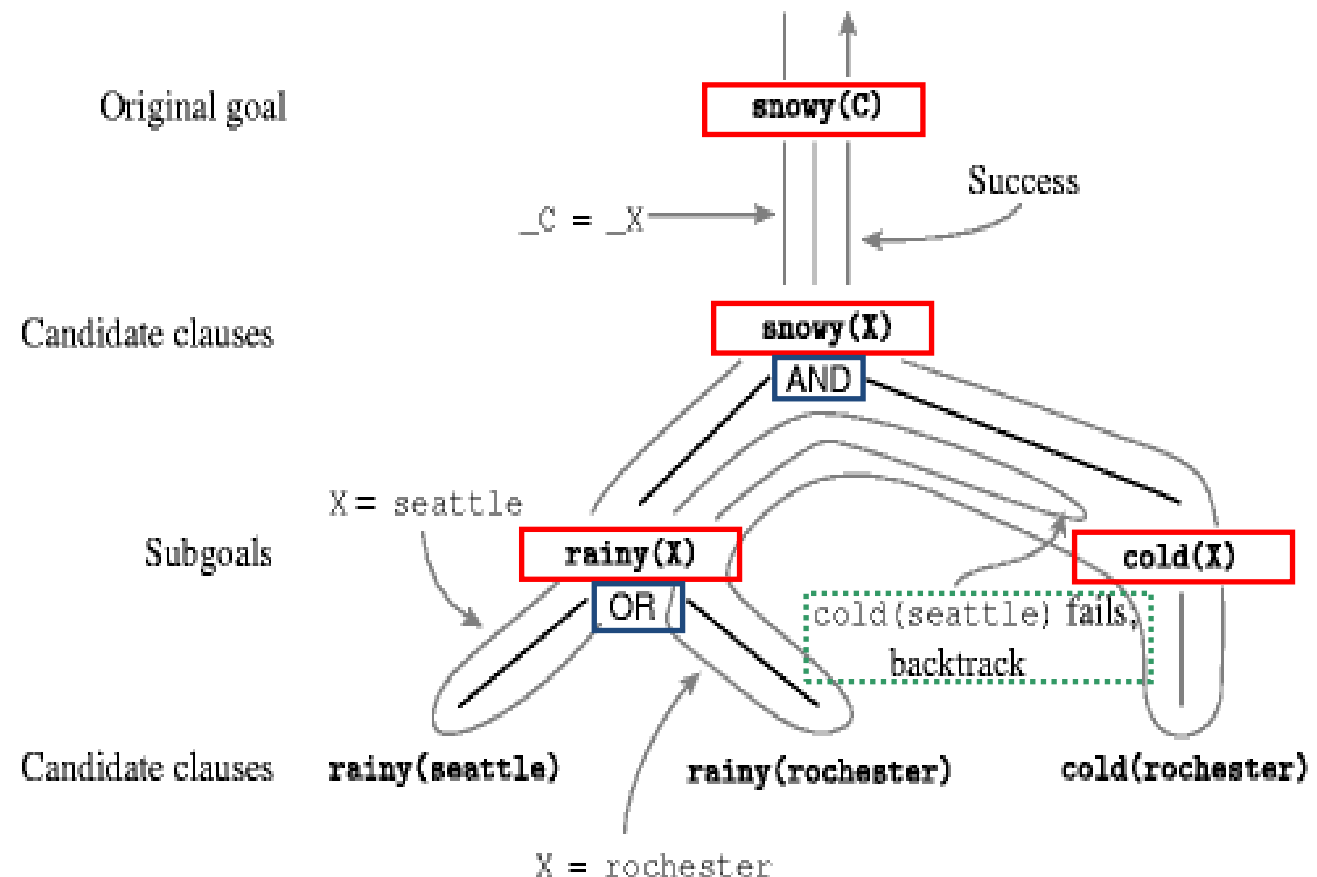
- Backward chaining follows a classic depth-first backtracking algorithm

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X)
```

- Example

– Goal:

Snowy (C)





# Depth-first backtracking

- The search for a resolution is ordered and depth-first
  - The behavior of the interpreter is predictable
- Ordering is fundamental in recursion
  - Recursion is again the basic computational technique, as it was in functional languages
  - Inappropriate ordering of the terms may result in non-terminating resolutions (infinite regression)
  - For example: Graph

```
edge(a,b) . edge(b, c) . edge(c, d) .  
edge(d,e) . edge(b, e) . edge(d, f) .  
path(X, X) .  
path(X, Y) :- edge(Z, Y), path(X, Z) .
```

Correct



# Depth-first backtracking

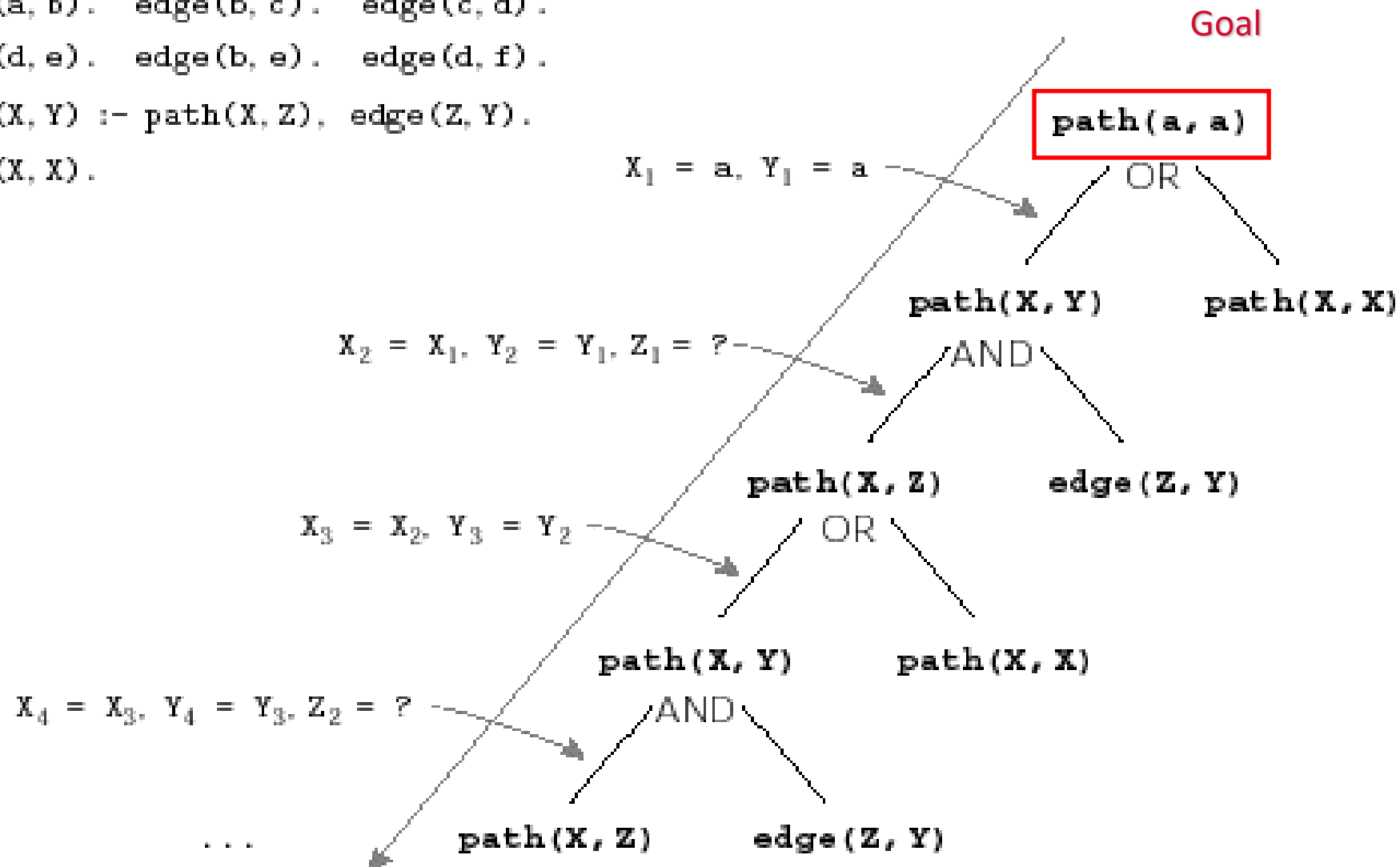
- The search for a resolution is ordered and depth-first
  - The behavior of the interpreter is predictable
- Ordering is fundamental in recursion
  - Recursion is again the basic computational technique, as it was in functional languages
  - Inappropriate ordering of the terms may result in non-terminating resolutions (infinite regression)
  - For example: Graph

```
edge(a,b) . edge(b, c) . edge(c, d) .  
edge(d,e) . edge(b, e) . edge(d, f) .  
path(X, Y) :- path(X, Z), edge(Z, Y) .  
path(X, X) .
```

Incorrect

# Infinite Regression

```
edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, Y) :- path(X, Z), edge(Z, Y).
path(X, X).
```





# Lists

```
memberH(X, [X|_]). /*to find the member*/  
memberH(X, [_|L]):-  
    member(X, L).  
concatenation([], L, L). /*to concatenate two lists*/  
concatenation([H|T], L1, [H|L2]):-  
    concatenation(T, L1, L2).  
  
add(X, L, [X|L]). /*to add an element in a list*/  
insert(X, L, LX):- del(X, LX, L).  
perm([], []).  
perm([H|T], P):-  
    perm(T, T1),  
    insert(H, T1, P).  
del(X, [X|L], L).  
del(X, [Y|L], [Y|L1]):-  
    del(X, L, L1).
```