PROLOG

Introduction to Prolog

Prolog: PROgramming in LOGic [1970]

- Prolog is a ***declarative language***
  - ***declarative*** meaning of a program defines WHAT the output should be
  - ***procedural*** meaning of a program defines HOW the output is obtained
- Prolog uses deduction in subset of first-order predicate logic
- Link for Prolog software: https://www.swi-prolog.org/

Introduction to Prolog

- Computer programming in Prolog consists of:
  - ***declaring*** some facts about objects and their relationships,
  - ***defining*** some rules about objects and their relationships,
  - ***asking*** questions about objects and their relationships.
- The PROLOG system enables a computer to be used as a store house of facts and rules, and it provides ways to make inferences.

Facts

- Facts in Prolog express relationships between objects.

```
likes(john, mary).
```

  - The names of the objects are called the ***arguments***; the name of the relationship is called the ***predicate***.
  - The names of all relationships and objects must begin with a lowercase letter
  - A fact must finishes with the "**.**" character.
- ***Examples:***

```
valuable(gold).
female(jane).
owns(john, gold).
```

```
father(john, mary).
king(john, france).
gives(john, book,mary).
```

- In Prolog, a collection of facts (and rules) that are used to solve a particular problem is called a ***database***.

Questions

When a question is asked to Prolog, it will search through the database. It looks for facts that match the fact in the question.

- If Prolog finds a fact that matches the question, Prolog will answer `yes`.
- If no such fact exists in the database, Prolog will respond `no`.

Questions

- ***Examples:***

```
likes(joe, fish).
likes(joe, mary).
likes(mary, book).
likes(john, book).

?- likes(joe, money).
no      -->  nothing matches the question; it doesn't mean 'false', but not provable

?- likes(mary,joe).
no

?- likes(mary, book).
yes
```

Variables

- Variables in Prolog stand for objects to be determined by Prolog (objects that we cannot name).
- Variables start with capital letters or an underscore character, e.g.

```
X, Object01, ShoppingList, _xyz
```

- A variable can be either ***instantiated*** or ***not instantiated***.
  - A variable is instantiated when there is an object that the variable stands for.
  - A variable is not instantiated when what the variable stands for is not yet known.

Satisfying a Goal (Question)

```
likes(john,flowers).
likes(john,mary).
likes(paul,mary).
?- likes(john,X).
X=flowers
```

- When Prolog is asked this question, the variable **`X`** is initially not instantiated. Prolog searches through the database, looking for a fact that ***matches*** the question. It searches in the database in the order it was typed in.

Satisfying a Goal (Question)

- If an uninstantiated variable appears as an argument, Prolog will allow that argument to match any other argument in the same position in the fact. When such a fact is found, then the variable **`X`** now stands for the second argument in the fact, (in this case flowers).

- We say that **`X`** is instantiated to `flowers`.

- Prolog now marks the place in the database where a match is found and prints out the object that the variable stands for.

Re-satisfying a Goal

- In response to Prolog's first answer we can ask it to satisfy the question in another way, i.e. to find another object that `X` could stand for.
  - This means that Prolog must *forget* that `X` stands for flowers, and resume searching with `X` uninstantiated again.
  - Because we are searching for an alternative solution, the search **is continued from the place-marker**.

```
likes(john,flowers).
likes(john,mary).
likes(paul,mary).
```

```
?- likes(john,X).
```
- -> our question

```
X = flowers;
```
- -> first answer. **We type ';' in reply**

```
X = mary;
```
- -> second answer. **We type ';'**

```
no
```
- -> no more answers

Conjunctions

```
likes(mary,food).
likes(mary,wine).
likes(john,mary).
likes(john, wine).
likes(mary, john).
```

Question: *Does John like Mary?* **and** *Does Mary like John?*

```
?- likes(john,mary), likes(mary,john).
```

Question: *Is there anything that John **and** Mary both like?*

- This question consists of two goals:
  - First, find out if there is some `X` that Mary likes.
  - Then, find out if John likes whatever `X` is.


```
?- likes(mary,X), likes(john, X).
```

Conjunctions

- When a sequence of goals is given, Prolog attempts to satisfy each goal in turn by searching for a matching fact in the database. All goals have to be satisfied in order for the sequence to be satisfied.
- If the first goal is in the database, then Prolog will mark the place in the database, and attempt to satisfy the second goal. If the second goal is satisfied, then Prolog marks that goal's place in the database, and we have found a solution that satisfies both goals.

```
likes(mary,food).
likes(mary,wine).
likes(mary,apple).
likes(john,mary).
likes(john, wine).
likes(mary, john).
likes(john,apple).

?- likes(mary,X), likes(john, X).
```

Conjunctions

- *Remember*: each goal keeps its own place-marker.
  If, the second goal is not satisfied, then Prolog will ***backtrack***, i.e. will attempt to re-satisfy the first goal.
- Prolog searches the database completely for each goal.
- When a goal needs to be re-satisfied, Prolog will begin the search from the goal's own place-marker, rather than from the start of the database.

```
likes(mary,food).
likes(mary,wine).
likes(john,mary).
likes(john, wine).
likes(mary, john).

?- likes(mary,X), likes(john, X).
X = wine;
no
```

Rules

- In Prolog, rules are used when you want to say that a fact ***depends*** on a group of other facts.
- Rules are also used to express definitions:
    - `X` is a bird if:
    - `X` is an animal, and
    - `X` has feathers.

    - `X` is a sister of `Y` if:
    - `X` is female, and
    - `X` and `Y` have the same parents.
- A variable stands for the same object wherever it occurs in a rule.
- A variable can *stand for a different object* in each different use of the rule.
- Rule is a ***general statement about objects and their relationships*****.**

Rules

In Prolog, a rule consist of a ***head*** and a ***body***, connected by the symbol ":-".

- The head of the rule describes what fact the rule is intended to define.
- The body of the rule is a goal or a conjunction/disjunction/negation of goals that must be satisfied, for the head to be true.

Examples:

- *John likes anyone who likes wine* ,

likes(john, X) :- likes(X, wine).

- *John likes X if X likes wine and food*

likes(john, X) :- likes(X, wine), likes(X, food).

- *John likes X if X likes wine or food*

likes(john, X) :- likes(X, wine);likes(X, food).

John likes X if X likes wine and food but not bread or X likes fruit

likes(john,X):-(likes(X,wine),likes(X,food),not(likes(X,bread)))
;likes(X,fruit).

Matching a Rule

A person may steal something if the person is a thief and the person likes the thing.

- **Consider the following database:**

```
thief(john).
likes(joe,food).
likes(joe,wine).
likes(john,X):-likes (X,wine).
kidnap(X, Y):-thief (X), likes (X, Y).
? - kidnap(john, X).
X  =  joe
```

Note that the definition of *likes* has three separate clauses: two facts and one rule.

Prolog Syntax

- Prolog programs are built from terms. A term is either a constant, a variable, or a structure.
- Constants: atoms, integers and floating point numbers.
- Atoms either begin with a lower-case letter and can include letters, digits and the underline character "_", or are made up from signs. If the atom has to begin with a capital letter or a digit, it should be enclosed in single quotes.

  Example: `likes, g_smith, 'George Smith', v123`

- Integers consist of digits and may not contain a decimal point.
- Variables have names beginning with a capital letter or an underline sign "_".

Structures

Much of the information that we want to represent in a program is compound, that is, it consists of **entities** which have a **number of different attributes**.

For example, the **person** entity might have a number of attributes such as **age, height, weight,** and so on**.**

General Form of a Structure

structure-name (attribute, ..., attribute)

- Note that structures look like predicates, but they work differently.
- predicates represent relationships; structures (and other terms) represent objects.

Structures

Suppose we want to represent cars with attributes make, age, price.

We might use a three-place structure called car; e.g.

car(ford, 3, 5000)

It represent a 3-year-old Ford selling for $5,000.

Structures of this type could be used in clauses such as:

```
% has(P,C) is true if P has a car matching C

has(joe, car(ford,3,5000)).

has(joe, car(opel,2,6000)).

has(mick, car(toyota,5,1000)).

has(mick, car(ford,2,2000)).
```

Structures-Continued

And we can pose queries like: "What kind of Ford does Mick have?"

Query: has(mick, car(ford, Age, Price))
Answer: Age=2, Price=2000

If we only want to get information about some fields we can use Prolog's "don't care" marker - the underscore character - to indicate this.

| ?- has(Person, car(ford,_,_)).

Person = joe ? ;

Person = mick

yes

Structures-Continued

If we wanted to know what make of car sold for under 5000, we might ask:

```
| ?- has(_, car(Make,_,Price)), Price < 5000.

Make = toyota
Price = 1000 ? ;

Make = ford
Price = 2000
```

Structures

- A ***structure*** is written by specifying its **functor,** and its components, for example:

```
book(wuthering_heights, bronte)
```

- Structures can be "nested" one inside another:

```
book(wuthering_heights, author(emily, bronte))
```

- Structures can appear inside facts and may participate in the process of question-answering.

```
owns(john,book(wuthering_heights,author(emily, bronte))).
?- owns (john, book (X, author (Y, bronte))).
```

(If John owns a book by any of the Bronte sisters).

- If this is true, `X` will be then instantiated to the title that was found,
  and `Y` will be instantiated to the first name of the author.
- The syntax for structures is the same as for facts. A predicate is actually the *functor* of a structure. The arguments of a fact or rule are actually the *components* of a structure. There are many advantages to representing Prolog programs as structures.

Arithmetic

- "Built-in" predicates for comparing numbers

`X = Y`	`X` and `Y` stand for the same number
`X < Y`	`X` is less than `Y`
`X > Y`	`X` is greater than `Y`
`X =< Y`	`X` is less than or equal to `Y`
`X >= Y`	`X` is greater than or to `Y`

The "`is`" operator:

- To satisfy an "`is`", Prolog first evaluates its right-hand argument according to the rules of arithmetic. The answer is then matched with the left-hand argument to determine whether the goal succeeds.

Arithmetic Operators

- "Built-in" arithmetic operators (that can be used in the right-hand side of the "`is`" operator):

`X + Y`	the sum of `X` and `Y`
`X - Y`	the difference of `X` and `Y`
`X * Y`	the product of `X` and `Y`
`X / Y`	the quotient of `X` divided by `Y`
`X // Y`	`integer division`
`X mod Y`	the reminder of `X` divided by `Y`

- Consider the following database about the population and area of different countries in 1976 (the population is given in millions of people, the area - in millions of square miles):

```
pop(usa, 203).
pop(india, 546).
pop(china, 800).
pop(brazil, 108).

area(usa, 3).
area(india, 1).
area(china, 4).
```

Arithmetic

- To find the population density of a country, we must use the rule that the density is the population divided by the area. A Prolog rule for this is:

```
density(X , Y) :-
        pop(X, P),
        area(X, A),
        Y is P/A.
```

- The divide operator " `/` " is integer division, which gives only the integer part of the quotient of the result.
- We use the "`is`" predicate any time we require to ***evaluate*** an arithmetic expression i.e. to calculate the result.

Arithmetic

```
pop(usa, 203).
pop(india, 546).
pop(china, 800).
pop(brazil, 108).

area(usa, 3).
area(india, 1).
area(china, 4).
area(brazil, 3).

density(X, Y)  :-
    pop (X, P),
    area (X, A),
    Y is P/A.

?-  density(china, X).
X = 200
yes
?-  density(turkey, X).
no
```

Operators

- This is a form of syntax that makes some structures easier to read. For example, arithmetic operations:

  `x + y * z` (instead of the normal way for structures: `+ (x, * (y,z) )`).

- A structure made up of arithmetic operators is like any other structure. No arithmetic is actually carried out until commanded by the Prolog "`is`" predicate. So, `3 + 4` does not mean the same thing as `7`. It is another way to write the term `+(3, 4)`.

Equality and Matching

- Equality predicate (built-in): an infix operator written as "=".
- When an attempt is made to satisfy the goal `X = Y` (where `X` and `Y` are any two terms), Prolog attempts to match `X` and `Y`, and the goal succeeds if they match.
- **Example:** The following question succeeds, causing `X` to be instantiated to the structure `rides(john, bicycle)`:


```
?- rides(john, bicycle) =  X.
```

Equality and Matching

- Integers and atoms are always equal to themselves.

Example:

```
policemen = policemen          succeeds
paper = pencil                 fails
1010 =1010                     succeeds
1010 = 1020                    fails
```

- Two structures are equal if they have the same ***functor*** and number of components, and all the corresponding components are equal. For example, the following goal succeeds and causes `X` to be instantiated to `bicycle`:

```
rides(john, bicycle) = rides(john, X).
```

- If we attempt to make two uninstantiated variables equal, the goal succeeds, and the two variables **share.** If two variables share, then whenever one of them becomes instantiated to some term, the other one automatically is instantiated to the same term.
- The "not equal" predicate: "`\=`".
- The goal `X \= Y` succeeds if `X = Y` fails, and it fails if `X = Y` succeeds.

Summary of Satisfying Goals

- Prolog performs a task in response to a question from the programmer. A question provides a ***conjunction*** of goals to be ***satisfied.*** Prolog uses the known clauses to satisfy the goals.
- A fact causes a goal to be satisfied immediately, whereas a rule can only reduce the task to satisfying a conjunction of ***subgoals.*** A clause can only be used if it matches the current goal. If a goal cannot be satisfied, **backtracking** will be initiated.

Summary of Satisfying Goals

- Backtracking consists of reviewing what has been done, attempting to re-satisfy the goals by finding an alternative way to satisfying them, You can initiate backtracking yourself by typing a semicolon when Prolog informs you of a solution.
- ***Matching***:
  - An uninstantiated variable will match any object. As a result, that object will be what the variable stands for.
  - An integer or an atom will match only itself.
  - A structure will match another structure with the same functor and number of arguments, and all the corresponding arguments must match

Looping in Prolog

To execute the instruction a fixed number of times, many programming languages provide 'for loop'.

In Prolog, there is no such facility available directly, but using recursion, we can obtain a similar effect, which is shown in the following programs:

```
loop(N):- N>0, write(N),nl,
S is N-1, loop(S).
```

Factorial Example

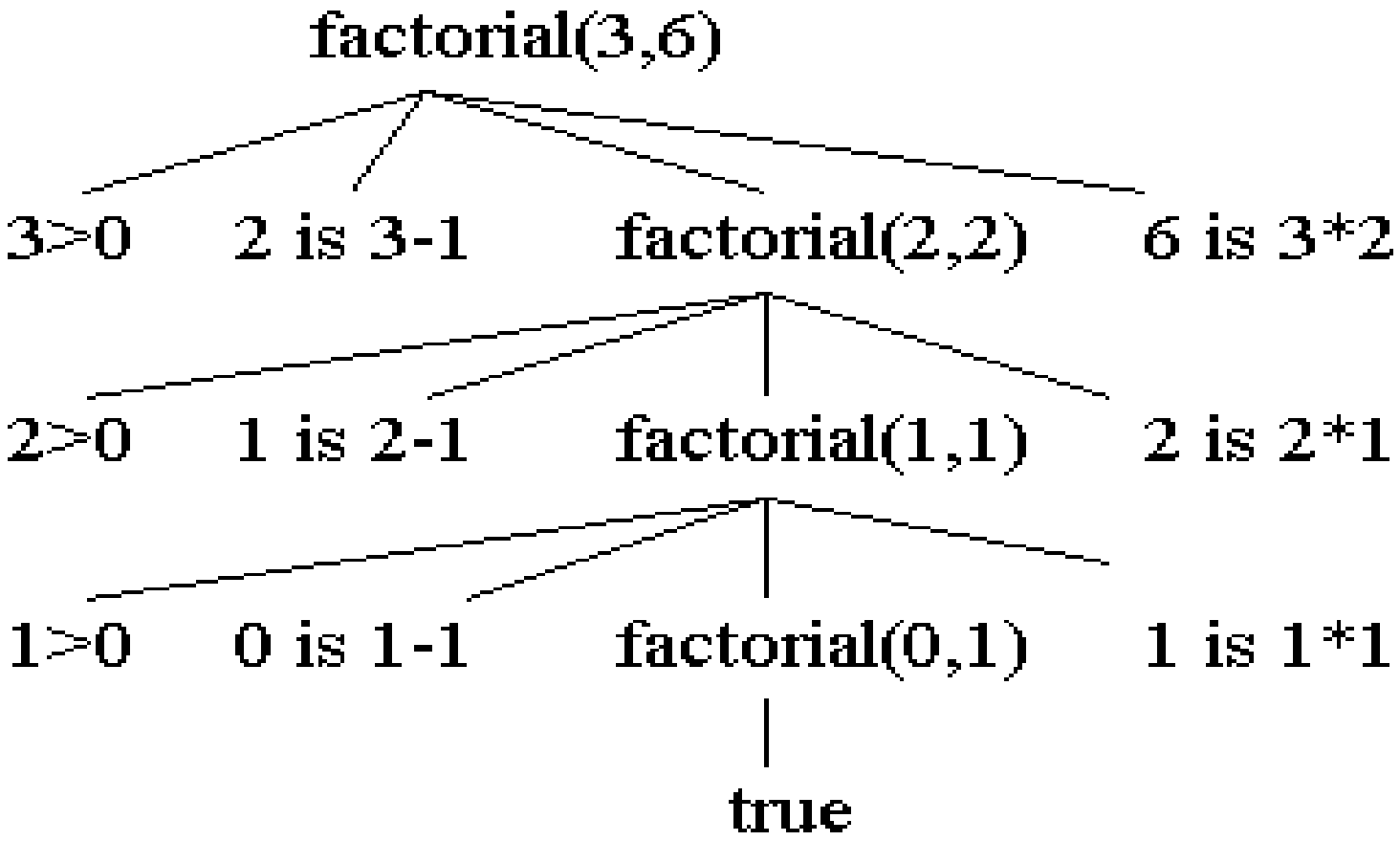
```
factorial(0,1).

factorial(N,F) :-
  N>0,
  N1 is N-1,
  factorial(N1,F1),
  F is N * F1.
```

The first clause is a unit clause, having no body.

The second is a rule, because it does have a body which consists of literals separated by commas ',' each of which can be read as "and". The head of a clause is the whole clause if the clause is a unit clause, otherwise the head of a clause is the part appearing to the left of the colon in ':-'.

A declarative reading of the first (unit) clause says that "the factorial of 0 is 1" and the second clause declares that "the factorial of N is F if N>0 and N1 is N-1 and the factorial of N1 is F1 and F is N*F1".


factorial(3,6)
3>0
2 is 3-1
factorial(2,2)
6 is 3*2
2>0
1 is 2-1
factorial(1,1)
2 is 2*1
1>0
0 is 1-1
factorial(0,1)
1 is 1*1
true

Built-in Predicates in PROLOG

- **TRACE**

- The effect of satisfying the goal trace is to turn on exhaustive tracing. As a result you will get to see every goal that your program generates at each of the four main ports (***Call, Exit, Redo*, and *Fail***).

- ***NOTRACE***

- The effect of satisfying the goal notrace is to stop exhaustive tracing from now on.

Built-in Predicates in PROLOG Contd…..

- **read/1 i.e. read(Term)**

  Read the next Prolog term from the current input stream and unify it with *Term*

- **write/1 i.e. write(Term)**

  Write *Term* to the current output

- **writeln/1 i.e. writeln(Term)**

  Equivalent to write(Term), nl.. The output stream is locked, which implies no output from other threads can appear between the term and newline.

Built-in Predicates in PROLOG Contd…..

- CUT Predicate

  Prolog is non-terminating in nature due to backtracking.

  Cut predicate is used for pruning the backtracking.

  It is represented as **!**.

  For example, consider the following program to find maximum of two numbers

  max(X,Y,X):- X>Y. // max of X and Y is X

  max(X,Y,Y):-X=<Y. // max of X and Y is Y

  The above program can be made efficient using cut as:

  max(X,Y,X):- X>Y,!.

  max(X,Y,Y):- X=<Y.

  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.)

Built-in Predicates in PROLOG Contd…..

Advantages of Cut:

The advantages of using Cut predicate is that program using cut operates faster because it does not waste time in satisfying those sub goals which will never contribute to the solutions. Program may occupy less of memory space because search tree is cut down and does not generate redundant branches.

Built-in Predicates in PROLOG Contd…..

Use of Cut

G:-A, B, C, ! , D, E, F. (1)

G :- P, Q. (2)

There are two possible rules for satisfying goal G. Using rule 1, subgoals A,B and C are tried to be satisfied for one time.

Case 1: If A,B and C succeed once, then Cut is said to be crossed and then rule 2 is never tried for backtracking for G. The result depends on subgoals D,E and F.

Built-in Predicates in PROLOG Contd…..

If any of the subgoal D,E or F fail, then there is no answer for G. Otherwise backtracking is done for suboals D, E and F in order to find all possible solutions of G.

Case 2: If any of the subgoals A, B or C fails, then Cut is not crossed and hence answer is tried for G thrugh rule 2 by satisfying subgoals P and Q.

Built-in Predicates in PROLOG Contd…..

Example of CUT

Consider the following Prolog program:

person(X):- **man(X), !.** (1)
person(X):- woman(X). (2)
man(john). (3)
man(smith). (4)
woman(kate). (5)
woman(mary). (6)

What will be the output of the query? –person(P).

Ans.

X= john

(only first successful instance of man(X)

No backtracking to (2) because ! is crossed)

Built-in Predicates in PROLOG Contd…..

Consider the following Prolog program:

person(X):- !, **man(X).** (1)

person(X):- woman(X). (2)

man(john). (3)

man(smith). (4)

woman(kate) (5)

woman(mary). (6)

What will be the output of the query? –person(P).

Ans.

X=john

X=smith

(All successful instances of man(X). No backtracking to (2) because ! is crossed)

Built-in Predicates in PROLOG Contd…..

- FAIL

The built in predicate *fail* is used to make the clause in which it appears fail explicitly. It is used to represent the negation. Predicate fail tells Prolog interpreter to fail a particular goal and subsequently forces backtracking. All the sub goals defined after fail will never be executed. So, it should be the last predicate in a rule.

Built-in Predicates in PROLOG Contd…..

FAIL

Example, the sentence “John does not like pasta” can be represented using fail predicate as

like(john, pasta):-fail.

?- ***like(john, pasta).***

No.

Recursion Contd….

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

A Recursive Rule Definition

- The predecessor relation can be defined as follow:

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

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

predecessor (X, Z) :-
        parent(X, Y1),
        parent(Y1, Y2),
        parent(Y2, Z).
...
```

- This program is lengthy and more importantly works to some extent.

Recursion Contd..

- There is an elegant and correct formulation of the predecessor relation in terms of itself:

  `X` is a predecessor of `Z` if:

  `X` is a parent of `Y` and

  `Y` is a predecessor of `Z`.

- Here is a Prolog clause with the same meaning:

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

- The complete program for the predecessor relation consists of two rules: one for *direct* predecessors and one for indirect predecessors.

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

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

Recursion

•Factorial of a number

```
factorial(0,1).
factorial(N,F) :- N>0,
                  N1 is N-1,
                  factorial(N1,F1),
                  F is N * F1.
```

Lists

- List is a data structure widely used in non-numeric programming.
- List consists of a number of items, such as
- A list can be empty or non empty.
    - [red,blue,green,white,red]
- An empty list is represented as [ ].
- A non-empty list consist of two things:
    - The first item called the head of the list.
    - The remaining part is called Tail and the tail must be another list.
    - The head and tail part of the list can be sepeartely written as [Head|Tail]

    •For example [a,b,c] can be written as:

    [a,b,c]= [a|[b,c]] = [a,b | [c]] = [a,b,c | [ ]]

Lists Operations

1) member(A,List)

Checks whether A is a member of the list or not where A can be a constant or another list.

For example: member(b,[a,b,c]) is true

member (b,[a,[b,c]]) is not true

member([b,c],[a,[b,c]]) is true

2) append(L1,L2,L3)

It appends list 1 and list 2 and places in list 3