Logic Programming

1. Application Domains:

- Logic programming language application areas include natural language processing, expert systems, specifications checking, theorem proving, and control systems (among others).

2. Definitions

- Logic programming languages, also called ***declarative* languages**, differ substantially from the other programming languages we have discussed.

- Most importantly, logic programs do not specify the execution sequence as in most in the case of other language paradigms. User specifies the specifications of a solution and the computer derives the execution sequence for that solution:

Let us have airline flight information of the form:

flight(flight_number, from_city, to_city, departure_time, arrival_time)

Then all the flights from Washington DC to Santa Clara can be specified as either direct flights or as flights with an intermediate stop:

flight(flight_number, DC, Santa Clara, departure_time, arrival_time)

or

flight(flight_number, DC, **X**, depart1, arrive1)
flight(flight_number, **X**, Los Angles, depart2, arrive2)
depart2>=arrive1+30

- Unlike imperative and functional programming, where implementation is done using a ***mapping***, logic programming uses ***relations***:
    - In imperative and functional, we say
        - Given a value x compute mapping(a), i.e. square(4)
    - In logic programming, we have
        - Given a value x and y, determine whether x is related to y is true
        - Consider the ***relation*** ParentOf as follows:
            - ✓ ParentOf(X,Y) where X is parent of Y
            - ✓ ParentOf(john, Mary) - John is parent of Mary

3. Basic Definitions:

- **Horn** clause:

  A Horn clause is an IF clause and not IFF, formally,

  A0 if A1 and A2 and A3 and … and An

  Each Ai is a simple assertion of the form Ri(…) where Ri is a relation name.

  Informally, this clause means that, if A1, …, and An are all true, then we can infer that A0 is also true. But, we cannot conversely infer that A0 is false just because some Ai turns out to be false.

- ***A logic program is a collection of Horn clauses***.

4. Logic programs:

- A logic program consists of a database of the following:

  - Facts: Concrete relations among objects.

- Rules: A pattern of relationships among the database objects.

- User query:
  - Computation in a logic program consists of testing a given assertion (query) **A**. The facts and the rules of the program are used to determine which substitutions for variables in the query (called ***unification***) are consistent with the information in the database. If we can infer from the clauses of the program that **A** is true, then we say that **A** succeeds. Otherwise, we say that **A** fails. Note that this does not mean that **A** is false.

  - For example, Prolog as an interpreter,
    - Prompts the user for an input query and
    - Outputs:
      - The truth value ("yes") or falsity ("no") of that query, and
      - an assignment to the variables of the query that make the query true (that unifies the query).

- The testing of a query **A** can be implemented by a technique called ***resolution***:

  - ✓ If the program contains a fact ‘A0’, such that A0 matches **A**, then we immediately conclude that **A** succeeds.
  - ✓ If the program contains a clause ‘A0 if A1 and … and An’ such that A0 matches A, then we proceed by testing A1, ... and An separately as sub-queries. If all succeed, then we conclude that A succeeds. If one of the Ai fails, then we must backtrack, i.e., give up the attempt to use this particular clause and try another clause instead.
  - ✓ Only we have exhausted all clauses whose left-hand sides match **A** can we conclude that **A** fails.

- Resolution uses Unification:

  - ✓ Unification:

    To unify two expressions U and V, we need to find substitutions for the variables occurring in U and V that makes the two expressions identical.
    Generally, it is denoted by σ and written as $U\sigma = V\sigma$

- ✓ Example:
  - To unify the two expressions R1(X,John) with f(g(John), Z), bind X to g(John) and Z to John to get f(g(John), John) as the unified instance of both expressions.
  - If the database contains the fact ParentOf(John, Mary) then the unification of the following query

    ParentOf(X,Mary)=ParentOf(John,Y)

    Would be

    ParentOf(John,Mary)

    where X is substituted by John and Y is substituted by Mary.

- For example, the programmer supplies a list of facts and rules:

  - Rules:
    - ✓ Eagle has feather.
    - ✓ Eagle has beak.
    - ✓ Eagle has two legs.

        - ✓ Betty eats vegetables.
        - ✓ Chicken eats seeds.
        - ✓ Chicken has two legs.

    - Facts:
        - ✓ If someone eats meat then it is a carnivore.
        - ✓ If someone has feather then it is a bird.
        - ✓ If someone eats X then X is food.

- Finally, the programmer can ask questions based on those facts and rules, such as:
    - Is eagle a bird?
    - Is chicken a bird?

- The language engine then tries to apply the rules to answer the questions, e.g.:
    - Yes (meat is a food).
    - Yes (eagle and chicken are bird).

- Note that the answers are based only on the available information.

5. Programming Languages vs. IPL & FPL

- In a logic programming language, the programmer *does not* identify the computations or functions necessary to derive an answer.

- The great advantage of a declarative language is that the programmer only needs to supply the information that is known (facts and rules), and ask questions - *NOT* worry about the ways the rules must be applied to answer those questions.

- From a programmer's perspective, this is a much more high-level approach to problem solving than in imperative or functional languages.

6. Drawbacks of Logic programming:

- The query process can be quite slow and inefficient
- The programmer must supply sufficient facts
- Rules to allow the engine to answer the queries
- The programmer must understand how to phrase the facts, rules, and queries

7. Prolog

- Prolog (**PRO**gramming **LO**Gic) (1972): was the first widespread formal logic language. It is a non-procedural, goal-oriented language.
- Computation takes place by trying to infer the response to queries through accessing a database of facts and rules.
- The user supplies the facts, rules, and queries (expressed in predicate calculus), while the computer carries out resolution to try and infer the replies.
- By far the dominant logic programming language is Prolog, so that is the language we will examine in some detail.
- Prolog is based predominantly on the Horn clause,

  A :- B, C, D, ...

  This tells us A is true if B is true and C is true and ... etc. It does not tell us the circumstances under which a is false

🡻 Getting Started

- In fact, in Prolog all the queries are expressed like yes/no questions, and the answers come back either as
    - A simple "No" (meaning no way could be found to make the query true), or
    - "Yes", plus all the ways in which the query could be made true.

- Core components:

    - ✓ The basic elements of a Prolog "program" are:

        - **Entities:** the "things" we are discussing, such as "mars", "moon", "sun".
        - **Variables:** as with more traditional programming languages, we can declare a variety of variables in Prolog
        - **Properties:** we can assign properties to individual entities, for example fred would have the property of being a carnivore properties look like a C function call - the name of the property then the entity that has that property is given in brackets, e.g. **carnivore (fred)**, or **food (vegetables)**

        - **Relationships (facts) :** we can assign relationships between entities, for example fred eats meat, or wilma eats vegetables

relationships in Prolog again look like a C++ function call, we give the relationship name first, then in brackets the two entities that are related, e.g. eats(wilma,meat), eats(wilma,vegetables)

- **Rules:** we can establish rules that describe how one relationship or property is true if some other properties or conditions are true. For example, a rule could state that if someone eats meat and they eat vegetables then they are an omnivore.

- Rules in Prolog have three parts:
  - The result: this is the relationship or property which results from the rule, and goes on the left hand side of our rule, e.g. omnivore(Individual)
    Note that the entity here will usually be a variable.
  - The "if" symbol: **:-** which separates the **result** from the **clauses**
  - The clauses: the set of relationships and properties which determine if the rule is to be applied, this goes on the right hand side of the rule.

- There are several logic symbols which can be used in the right hand side:
  - The comma represents logical AND,

- The semi-colon represents logical OR,
- The word NOT in some ways functions like logical NOT, but there are some important differences that we'll consider later

8. Create a Prolog program:

- Create our set of properties, relationships, and rules. Typically we will enter these into a text file ending with the extension ".pl" (e.g. "flint.pl").
- Start Prolog and load our database from the file.
- Begin issuing queries based on the loaded information.
- *From this point on, everything you type is treated as a query, not a new fact or rule:*

  eats(fred,vegetables). is now *asking* whether or not fred eats vegetables, not stating it as a new fact.

- SYNTAX NOTE: all statements of properties, relationships, rules, and queries *must* end with a period -- the Prolog interpreter will not regard any of the above as completed until you enter a period.

- **Example:**

  - ✓ Create a file, called flint.pl, containing the following facts and rules:

```
eats(fred,meat).
eats(wilma,meat).
eats(betty,vegetables).
eats(wilma,vegetables).
eats(barney,meat).
eats(barney,vegetables).

carnivore(Individual) :- eats(Individual,meat).
omnivore(Individual) :-
eats(Individual,meat),eats(Individual,vegetables).
food(Thing) :- eats(Individual,Thing).
```

  - ✓ Once that is created, we can load the database of facts and rules from the file, then begin issuing queries and getting answers.

  - ✓ In the example below assume > is the prompt:

```
>['flint.pl'].
flint.pl compiled

Yes
>food(meat).
```

Yes
>carnivore(fred).

Yes
>food(dirt).

No
>omnivore(Individual).

Individual = wilma

Yes

- **How queries are answered:**

  - ✓ Suppose we are given the query "is fred an omnivore?"

    omnivore(fred).

  - ✓ In trying to solve this query, Prolog does the following:
    - search for all the relationships which match the goal, i.e. everything in our database that begins with omnivore(...)
    - if we encounter a fact in our database that explicitly states omnivore(fred). then we can quit and return Yes as soon as we encountered it.
    - if we encounter a rule of the form omnivore(...variable...) :- ... then we can say fred is an omnivore *if* we can substitute fred for the variable

(throughout the rule) and still satisfy all the clauses on the right side of the rule

- In the case of our example, the rule with this form is

  omnivore(Individual) :- eats(Individual,meat),eats(Individual,vegetables).

  and after substituting we want to check

  if eats(fred,meat) AND eats(fred,vegetables).

  Thus we have two new goals to try to satisfy to check out this rule, and we begin searching the database for facts and rules beginning with eats(..., trying to substitute "fred" and "meat" into the rules to continue the query.

  If we cannot prove eats(fred,meat) then that clause fails, and hence the rule fails as well, so we must continue on checking for other facts/rules defining omnivores.

- As soon as we satisfy one of the facts or rules that prove fred is an omnivore we can stop and return “Yes”.
- If we go through all our facts and rules and cannot prove fred is an omnivore then we return “no”.

- Another example: suppose our database is:
  - rainy(yesterday).
  - rainy(today).
  - cloudy(Day) :- rainy(Day).

Again, let us make the query cloudy(today).

In this case, the first applicable rule (i.e. that has cloudy as a head and one parameter) is cloudy(Day) :- rainy(Day).
Because Day is a variable, and we are looking for the specific value today, we can *instantiate* the variable Day with the value today, giving us the rule
cloudy(today) :- rainy(today).

From this point the process continues by trying to satisfy subgoal rainy(today) and succeed because of the applicable fact in the database.

- Another example:

Consider the following logic program:

mother (mary, sue).

father (john, sue).

mother (mary, bill).

father (john, bill).

mother (sue, nancy).

father (bob, nancy).

mother (sue, jeff).

father (bob, jeff).

mother (jane, ron).

Father(bill,ron).

Parent(A,B) :- father(A,B).

Parent (A,B) :- mother(A,B).

Grandparent(C,D) :- parent(C,E), parent(E,D).

- Show the trace of the processing of the following queries:

  ?- grandparent(Who, ron).

- Add a sibling relationship to the above program and answer the following question:

  ?- sibling(sue, X).