MODULE 5
Logic Programming Languages

Logic Programming Languages

- Basic Elements of Prolog
- Applications of Logic Programming

The Basic Elements of Prolog

Terms

- Prolog programs consist of collections of statements. There are only a few kinds of statements in Prolog, but they can be complex. All Prolog statement, as well as Prolog data, are constructed from terms.
- A Prolog **term** is a constant, a variable, or a structure.
- A constant is either an atom or an integer.
- An atom is either a string of letters, digits, and underscores that begins with a lowercase letter or a string of any printable ASCII characters delimited by apostrophes.
- A variable is any string of letters, digits, and underscores that begins with an uppercase letter or an underscore (_).
  - The binding of a value, and thus a type, to a variable is called an **instantiation.** Instantiation occurs only in the resolution process.
  - A variable that has not been assigned a value is called **uninstantiated.**

- The last kind of term is called a **structure**.
- Structures represent the atomic propositions of predicate calculus, and their general form is the same:

  functor(parameter list)
- The functor is any atom and is used to identify the structure.
- The parameter list can be any list of atoms, variables, or other structures.

➢ Fact Statements

- Prolog has two basic statement forms; these correspond to the headless and headed Horn clauses of predicate calculus.
- The simplest form of headless Horn clause in Prolog is a single structure, which is interpreted as an unconditional assertion, or fact.
- Logically, facts are simply propositions that are assumed to be true.
- The following examples illustrate the kinds of facts one can have in a Prolog program.

```
female(shelley).
male(bill).
female(mary).
male(jake).
father(bill, jake).
father(bill, shelley).
mother(mary, jake).
mother(mary, shelley).
```

- Notice that every Prolog statement is terminated by a period.

- These simple structures state certain facts about jake, shelley, bill, and mary.
- For example, the first states that shelley is a female.
- The last four connect their two parameters with a relationship that is named in the functor atom; for example, the fifth proposition might be interpreted to mean that bill is the father of jake.

➢ Rule Statements

- The other basic form of Prolog statement for constructing the database corresponds to a headed Horn clause.
- The right side is the antecedent, or *if part*, and the left side is the consequent, or *then part*. If the antecedent of a Prolog statement is true, then the consequent of the statement must also be true.
- **Conjunctions** contain multiple terms that are separated by logical AND operations. In Prolog, the AND operation is implied. The structures that specify atomic propositions in a conjunction are separated by commas, so one could consider the commas to be AND operators. As an example of a conjunction,
- consider the following:

  female(shelley), child(shelley).

- The general form of the Prolog headed Horn clause statement is

  consequence :- antecedent_expression.
- It is read as follows: “consequence can be concluded if the antecedent expression is true or can be made to be true by some instantiation of its variables.”
- For example,

  ancestor(mary, shelley) :- mother(mary, shelley).
  - states that if mary is the mother of shelley, then mary is an ancestor of shelley.
- Headed Horn clauses are called **rules**, because they state rules of implication between propositions.

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

➢ **Goal Statements**

- The theorem is in the form of a proposition that we want the system to either prove or disprove.
- In Prolog, these propositions are called **goals, or queries**.
- The syntactic form of Prolog goal statements is identical to that of headless Horn clauses.
- For example, we could have

  man(fred).

  to which the system will respond either yes or no.
- The answer yes means that the system has proved the goal was true under the given database of facts and relationships. The answer no means that either the goal was determined to be false or the system was simply unable to prove it.

The Inferencing Process of Prolog

- This section examines Prolog resolution.
- Queries are called **goals.** When a goal is a compound proposition, each of the facts (structures) is called a **subgoal**. To prove that a goal is true, the inferencing process must find a chain of inference rules and/or facts in the database that connect the goal to one or more facts in the database.
- For example, if *Q* is the goal, then either *Q* must be found as a fact in the database or the inferencing process must find a fact *P1* and a sequence of propositions *P2, P3, c, Pn* such that

  P2 :- P1

  P3 :- P2

  . . .

  Q :- Pn
- Of course, the process can be and often is complicated by rules with compound right sides and rules with variables. The process of finding the *Ps,* when they exist, is basically a comparison, or matching, of terms with each other.

- There are two opposite approaches to attempting to match a given goal to a fact in the database.
- The system can begin with the facts and rules of the database and attempt to find a sequence of matches that lead to the goal. This approach is called **bottom-up resolution, or forward chaining**.
- The alternative is to begin with the goal and attempt to find a sequence of matching propositions that lead to some set of original facts in the database. This approach is called **top-down resolution, or backward chaining**.
- In general, backward chaining works well when there is a reasonably small set of candidate answers.
- The forward chaining approach is better when the number of possibly correct answers is large; in this situation, backward chaining would require a very large number of matches to get to an answer.
- Prolog implementations use backward chaining for resolution, presumably because its designers believed backward chaining was more suitable for a larger class of problems than forward chaining.

- The following example illustrates the difference between forward and backward chaining.
- Consider the query:

  man(bob).
- Assume the database contains

  father(bob).

  man(X) :- father(X).
- Forward chaining would search for and find the first proposition. The goal is then inferred by matching the first proposition with the right side of the second rule (father(X)) through instantiation of X to bob and then matching the left side of the second proposition to the goal.
- Backward chaining would first match the goal with the left side of the second proposition (man(X)) through the instantiation of X to bob. As its last step, it would match the right side of the second proposition (now father(bob)) with the first proposition.

- The next design question arises whenever the goal has more than one structure, as in our example.
- The question then is whether the solution search is done depth first or breadth first.
- A **depth-first search** finds a complete sequence of propositions—a proof—for the first subgoal before working on
- the others.
- A **breadth-first search** works on all subgoals of a given goal in parallel.
- Prolog's designers chose the depth-first approach primarily because it can be done with fewer computer resources. The breadth-first approach is a parallel search that can require a large amount of memory.

- The last feature of Prolog's resolution mechanism that must be discussed is **backtracking**.
- When a goal with multiple subgoals is being processed and the system fails to show the truth of one of the subgoals, the system abandons the subgoal it cannot prove.
- It then reconsiders the previous subgoal, if there is one, and attempts to find an alternative solution to it. This backing up in the goal to the reconsideration of a previously proven subgoal is called **backtracking.**
- Backtracking can require a great deal of time and space because it may have to find all possible proofs to every subgoal.

➢ Simple Arithmetic

- Prolog supports integer variables and integer arithmetic.
- Originally, the arithmetic operators were functors, so that the sum of 7 and the variable X was formed with

  +(7, X)
- Prolog now allows a more abbreviated syntax for arithmetic with the **is** operator.
- This operator takes an arithmetic expression as its right operand and a variable as its left operand.
- All variables in the expression must already be instantiated, but the left-side variable cannot be previously instantiated.
- For example, in

  A **is** B / 17 + C.

  if B and C are instantiated but A is not, then this clause will cause A to be instantiated with the value of the expression.

➢ List Structures

- So far, the only Prolog data structure we have discussed is the **atomic proposition**, which looks more like a function call than a data structure.
- Atomic propositions, which are also called structures, are actually a form of records.
- The other basic data structure supported is the **list**.
- Lists are sequences of any number of elements, where the elements can be atoms, atomic propositions, or any other terms, including other lists.
- The list elements are separated by commas, and the entire list is delimited by square brackets, as in

  [apple, prune, grape, kumquat]
- The notation [] is used to denote the empty list.

- Instead of having explicit functions for constructing and dismantling lists, Prolog simply uses a special notation.
- [X | Y] denotes a list with head X and tail Y, where head and tail correspond to CAR and CDR in LISP.
- A list can be created with a simple structure, as in

  new_list([apple, prune, grape, kumquat]).

  which states that the constant list [apple, prune, grape, kumquat] is a new element of the relation named new_list.
- That is, it states that [apple, prune, grape, kumquat] is a new element of new_list.
- Therefore, we could have a second proposition with a list argument, such as

  new_list([apricot, peach, pear]).

- In query mode, one of the elements of new_list can be dismantled into head and tail with

  new_list([New_List_Head | New_List_Tail]).
- If new_list has been set to have the two elements as shown, this statement instantiates New_List_Head with the head of the first list element (in this case, apple) and New_List_Tail with the tail of the list (or [prune, grape, kumquat]).
- If this were part of a compound goal and backtracking forced a new evaluation of it, New_List_Head and New_List_Tail would be reinstantiated to apricot and [peach, pear], respectively, because [apricot, peach, pear] is the next element of new_list.
- The | operator used to dismantle lists can also be used to create lists from given instantiated head and tail components, as in

  [Element_1 | List_2].
- If Element_1 has been instantiated with pickle and List_2 has been instantiated with [peanut, prune, popcorn], the sample notation will create, for this one reference, the list [pickle, peanut, prune, popcorn].

- The first two parameters to the append operation in the following code are the two lists to be appended, and the third parameter is the resulting list:
  - append([], List, List).
  - append([Head | List_1], List_2, [Head | List_3]) :-

  append(List_1, List_2, List_3).
- The first proposition specifies that when the empty list is appended to any other list, that other list is the result.
- One way to read the second statement of append is as follows: Appending the list [Head | List_1] to any list List_2 produces the list [Head| List_3], but only if the list List_3 is formed by appending List_1 to List_2.

- Suppose we need to be able to determine whether a given symbol is in a given list.
- A straightforward Prolog description of this is

```
member(Element, [Element | _]).
member(Element, [_ | List]) :- member(Element, List).
```

- The underscore indicates an “anonymous” variable; it is used to mean that we do not care what instantiation it might get from unification.

Applications of Logic Programming

1. **Relational Database Management Systems**

- Relational database management systems (RDBMSs) store data in the form of tables.
- Queries on such databases are often stated in Structured Query Language (SQL).
- SQL is nonprocedural in the same sense that logic programming is nonprocedural.
- The user does not describe how to retrieve the answer; rather, he or she describes only the characteristics of the answer.
- The connection between logic programming and RDBMSs should be obvious.
- Simple tables of information can be described by Prolog structures, and relationships between tables can be conveniently and easily described by Prolog rules.
- The retrieval process is inherent in the resolution operation.
- The goal statements of Prolog provide the queries for the RDBMS.
- Logic programming is thus a natural match to the needs of implementing an RDBMS.

- One of the advantages of using logic programming to implement an RDBMS is that only a single language is required.
- In a typical RDBMS, a database language includes statements for data definitions, data manipulation, and queries, all of which are embedded in a general-purpose programming language, such as COBOL.
- The general-purpose language is used for processing the data and input and output functions. All of these functions can be done in a logic programming language.
- Another advantage of using logic programming to implement an RDBMS is that deductive capability is built in.
- Conventional RDBMSs cannot deduce anything from a database other than what is explicitly stored in them.
- They contain only facts, rather than facts *and* inference rules.
- The primary disadvantage of using logic programming for an RDBMS, compared with a conventional RDBMS, is that the logic programming implementation is slower.
- Logical inferences simply take longer than ordinary table look-up methods using imperative programming techniques.

2. Expert Systems

- Expert systems are computer systems designed to emulate human expertise in some particular domain.
- They consist of a database of facts, an inferencing process, some heuristics about the domain, and some friendly human interface that makes the system appear much like an expert human consultant.
- In addition to their initial knowledge base, which is provided by a human expert, expert systems learn from the process of being used, so their databases must be capable of growing dynamically.
- Also, an expert system should include the capability of interrogating the user to get additional information when it determines that such information is needed.
- Prolog can and has been used to construct expert systems.
- It can easily fulfill the basic needs of expert systems, using resolution as the basis for query processing, using its ability to add facts and rules to provide the learning capability, and using its trace facility to inform the user of the "reasoning" behind a given result.

3. Natural-Language Processing

- Certain kinds of natural-language processing can be done with logic programming.
- In particular, natural-language interfaces to computer software systems, such as intelligent databases and other intelligent knowledge-based systems, can be conveniently done with logic programming.
- For describing language syntax, forms of logic programming have been found to be equivalent to context-free grammars.
- Proof procedures in logic programming systems have been found to be equivalent to certain parsing strategies.
- In fact, backward-chaining resolution can be used directly to parse sentences whose structures are described by context-free grammars.
- It has also been discovered that some kinds of semantics of natural languages can be made clear by modeling the languages with logic programming.