UNIT – VI Logic Programming Languages

- *Logic* programming systems allow the programmer to state a collection of axioms from which theorems can be proven.
- Express programs in a form of symbolic logic
- Use a logical inferencing process to produce results
- *Logic* programming languages are rule based languages
- *Declarative* rather that *procedural*:
  - Only specification of *results* are stated (not detailed *procedures* for producing them)

Introduction to predicate calculus

- Particular form of symbolic logic used for logic programming is ***predicate calculus.*** Symbolic logic provides basis for logic programming. predicate calculus contains Propositions

Propositions

- **Proposition** is logical statement that may or may not be true. It consists of objects and relationships of objects to each other

Objects

- Objects in propositions are represented by simple terms: either constants or variables
- *Constant*: a symbol that represents an object
- *Variable*: a symbol that can represent different objects at different times
  - Different from variables in imperative languages
- The simplest form of proposition is called **atomic proposition** consist of compound terms. A **compound term** is one element of a mathematical relation, written in a form that has the appearance of mathematical function

Compound term composed of two parts

- Functor: function symbol that names the relationship
- Ordered list of parameters (tuple)
- Examples:

  student(jon) --- jon is student

  like(rony, OS)

  like(nick, windows) ----- nick like windows

- like(jim, linux

Propositions are connected by operators

Name	Symbol	Example	Meaning
Negation	¬	¬ a	not a
Conjunction	∩	a ∩ b	a and b
Disjunction	∪	a ∪ b	a or b
Equivalence	≡	a ≡ b	a is equivalent to b
Implication	⊃ ⊂	a ⊃ b a ⊂ b	a implies b b implies a

.

Horn clauses

- Horn clauses Basis for logic programming
- Propositions can be stated as Horn clauses.
- propositions are written in a standard form known as Horn clause and it is consists of head (H) and body (B_1, …, B_n)

General form of horn clause is as follows

H :- B_1, …, B_n. read as B_1, …, B_n imples H
or
H $\subset B_1 \cap B_2 \cap \cdots\cdots \cap B_n$ read as $B_1 \cap B_2 \cap \cdots\cdots \cap B_n$ implies H

When B_i are all true, we can deduce that H is true *i.e Antecedent implies Consequent. (* B_1, …,B_n imples H). *Here comma (,) is taken as AND($\cap$) operator*

- right side (B_1, …, $B_{n)}$ is called *Antecedent*
- left side (H) is called *Consequent*
  Antecedent is called as if part and Consequent is called as then part

- ***Horn clause* - can have only two forms**

- ***Headed Horn clause***: single atomic proposition on left side
  Ex: parent(X,Y):- mother(X,Y).
  If X is a mother of Y then X is a parent of Y

  grandparent(X,Z):- parent(X,Y), parent(Y,Z)
  If X is a parent of Y and Y is a parent of Z then X is a grandparent of Z

- ***Headless Horn clause***: empty left side (used to state facts)

Ex: mammal(human) ------- human is a mammal
female(uma).

Resolution is an inference rule for horn clauses that allows inferred propositions to be computed from given propositions

- Resolution is the primary activity of a Prolog interpreter
- In order to derive new statements, logic programming system combines existing statements, cancelling like terms, through a process known as resolution
  Ex: C :- A , B
  D :- C
  ____
  D :- A , B

Unification is a process of finding values for variables in propositions that allows matching process to succeed. *Unification make two terms identical*

- ***Instantiation***: assigning temporary values to variables to allow unification to succeed
- After instantiating a variable with a value, if matching fails, may need to *backtrack* and instantiate with a different value
- Ex: **Prolog unifies man(X) with man(fred)** , thereby instantiating the variable X to fred .
- *Unification make two terms (X and fred) identical*
- After instantiating a variable with a value, if matching fails, may need to *backtrack* and instantiate with a different value

Programming with prolog

Prolog is a logic programming language, its programs consist of a list of facts and rules.

Prolog is used widely for artificial intelligence applications, and it is developed at University of Aix-Marseille and University of Edinburgh

Prolog is declarative: the program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a *query* over these relations.

The basic elements of prolog

Terms
Fact statements
Rule Statements
Goal Statements
Simple Arithmetic
List structures

Terms

All Prolog statements, as well as Prolog data, are constructed from **terms.**

Term is a constant, variable, or structure

- *Constant* is either an atom or an integer ex: fred, jake, 23 etc
- *Atom* is a symbolic value of Prolog
- Atom consists of either:
    - a string of letters, digits, and underscores beginning with a lowercase letter
    - a string of printable ASCII characters delimited by apostrophes
- *Variable*: any string of letters, digits, and underscores beginning with an uppercase letter
  Ex: X , Y , Z etc
- Structures represent the atomic propositions of predicate calculus, and their general form is functor(parameter list) `ex: parent(X,Y) where X and Y are term1 and term2`

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

Fact statements

Fact statements are used for the hypotheses

Headless Horn clauses are called Fact statements

The simplest form of headless Horn clause in Prolog is a single structure, which is interpreted as an unconditional assertion, or fact.

Examples of fact statements

mammal(human)
female(shelley).
male(bill).
father(bill, jake).
`mother(suma, shelley).`

Rule Statements

- Used for the hypotheses
- Headed Horn clause are called as Rule Statements
- 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

- Right side: *antecedent* (***if*** part)
  - May be single term or conjunction
- Left side: *consequent* (***then*** part)
  - Must be single term

Conjunction: multiple terms separated by logical AND operations

Examples of Rule Statements

```
ancestor(mary,shelley):- mother(mary,shelley).
parent(X,Y):- mother(X,Y).
parent(X,Y):- father(X,Y).
grandparent(X,Z):- parent(X,Y), parent(Y,Z).
```
if X is a prent of Y AND Y is a parent of Z **then** X is a grandparent of Z
```
sibling(X,Y):- mother(M,X), mother(M,Y),
               father(F,X), father(F,Y).
```

Goal Statements

- For theorem proving, theorem is in form of proposition that we want system to prove or disprove. These propositions are called *goal statement* s or **queries**
- Goal Statements are Same format as 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.

father(X,mike)

The approach of logic programming is to use as a database a collection of facts and rules that state relationships between facts and to use an automatic inferencing process to check the validity of new propositions, assuming the facts and rules of the database are true. This approach is the one developed for automatic theorem proving.

Simple Arithmetic

- Prolog supports integer variables and integer arithmetic
- is operator: takes an arithmetic expression as right operand and variable as left operand
  A is B / 17 + C

Trace

- Built-in structure that displays instantiations at each step
- *Tracing model* of execution - four events:
  - *Call* (beginning of attempt to satisfy goal)
  - *Exit* (when a goal has been satisfied)
  - *Redo* (when backtrack occurs)
  - *Fail* (when goal fails)

Inferencing Process of Prolog

It used to check to check the validity of new propositions

Queries are called goals. If a goal is a compound proposition, each of the facts is a subgoal

To prove a goal is true, must find a chain of inference rules and/or facts.

Process of proving a subgoal called matching, satisfying, or resolution. Prolog implementations use backward chaining (top-down resolution)

For goal Q:

```
B :- A
C :- B
…
Q :- P
```

- The following example illustrates the difference between forward (Bottom-up resolution) and backward chaining(top-down resolution).

- 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.
- Forward chaining starts with the known facts and asserts new facts

- **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. Backward chaining starts with goals, and works backward to determine what facts must be asserted so that the goals can be achieved

Backtracking in Prolog Example & Explanation

Prolog Program

```
/* Facts */
likes(mary, food).
// mary likes food.

likes(mary, wine).
// mary likes wine

    /* Rule */
    likes(john, X):- likes(mary,X).
    // john likes everything whatever mary like.

    /* Goal */
    ?-likes(john, What).

    What = food;

    What = wine ;
```

Explanation :

Here in query
? - likes(john, What).
'What' is a variable

Upon asking the query prolog first reads the query and find the match for it in the knowledge base in top - down manner. It finds the matching rule
likes(john, X).
But from that rule we can infer
likes(mary, X).
Now prolog w ill go back(backtrack) and will again read the knowledge base from the first line and finds likes(mary, food) and it will return answer as What = food .variable 'What' is instantiated with value 'food' . Here Prolog will keep the mark in database that for this goal What = food has been answered and it satisfies.

List Structures (lists)

In Prolog a very common data-structure is the list.

- **Definition**

List in a Prolog is a sequence of any number of elements

The elements are terms, i.e. constants, variables, structures, including other lists.

 Syntax: the elements are enclosed in square brackets, separated by commas.

Prolog also has a special facility to split the first part of the list (called the head) away from the rest of the list (known as the tail).

We can place a special symbol | (pronounced 'bar') in the list to distinguish between the first item in the list and the remaining list.

For example, consider the following.

```
[first,second,third] = [A|B]
```
where *A = first* and *B=[second,third]*

Examples:

[1, 3, 6, 2]

[john, mary, peter, 6,4]

[X, 3, 3, Z]

[noun(dog), verb(bites)]

[a, b, [a,b,c]]

[] *this is the empty list*

 Accessing the elements of a list

head of a list: its first element

tail of a list - the list containing all the elements except the first

- Elements can be atoms, atomic propositions, or other terms (including other lists)
  [apple, prune, grape, kumquat]
  [] *(empty list)*

Standard notation of prolog List: [H | T]

H - the head of the list, a single element

T - the list containing the remaining elements. Note that T is a list

Generally, left of | we can write explicitly individual elements, right of | stays the list of remaining elements

example
if [H | T] = [1, 2,3], then H = 1 and T = [2,3]

Matching variables to list's head and tail (comparisons of lists:)

[a,b,c] unifies with [Head|Tail] resulting in Head=a and Tail=[b,c]

[a] unifies with [H|T] resulting in H=a and T=[ ]

[a,b,c] unifies with [a|T] resulting in T=[b,c]

[a,b,c] doesn't unify with [b|T]

[] unifies with []. Two empty lists always match

Execution in Prolog

database contains facts and rules

Following is database

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

- **The following queries yield the specified answers.**

| ?- likes(mary,food).
the system will respond as
O/P : yes.
| ?- likes(john,wine).
yes.
| ?- likes(john,food).
no.

Applications of Logic Programming

- **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 Q0uery 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. The general-purpose language is used for processing
  the data and input and output functions. All of these functions can be done ina 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.

- 16.8.2 Expert Systems

Expert systems are computer systems designed to imitate 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.
One of the central problems for the designer of an expert system is dealing with the inevitable inconsistencies and incompleteness of the database. Logic programming appears to be well suited to deal with these problems. For example, default inference rules can help deal with the problem of incompleteness.
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. Missing from Prolog is the automatic ability of the system to query the user for additional information when it is needed.
One of the most widely known uses of logic programming in expert systems is the expert system construction system known as APES. The APES system includes a very flexible facility for gathering information from the user during expert system construction. It also includes a second interpreter for producing explanations to its answers to queries.
APES has been successfully used to produce several expert systems, including one for the rules of a government social benefits program and one for the British Nationality Act, which is the definitive source for rules of British citizenship.

16.8.3 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

Multiparadigm Languages

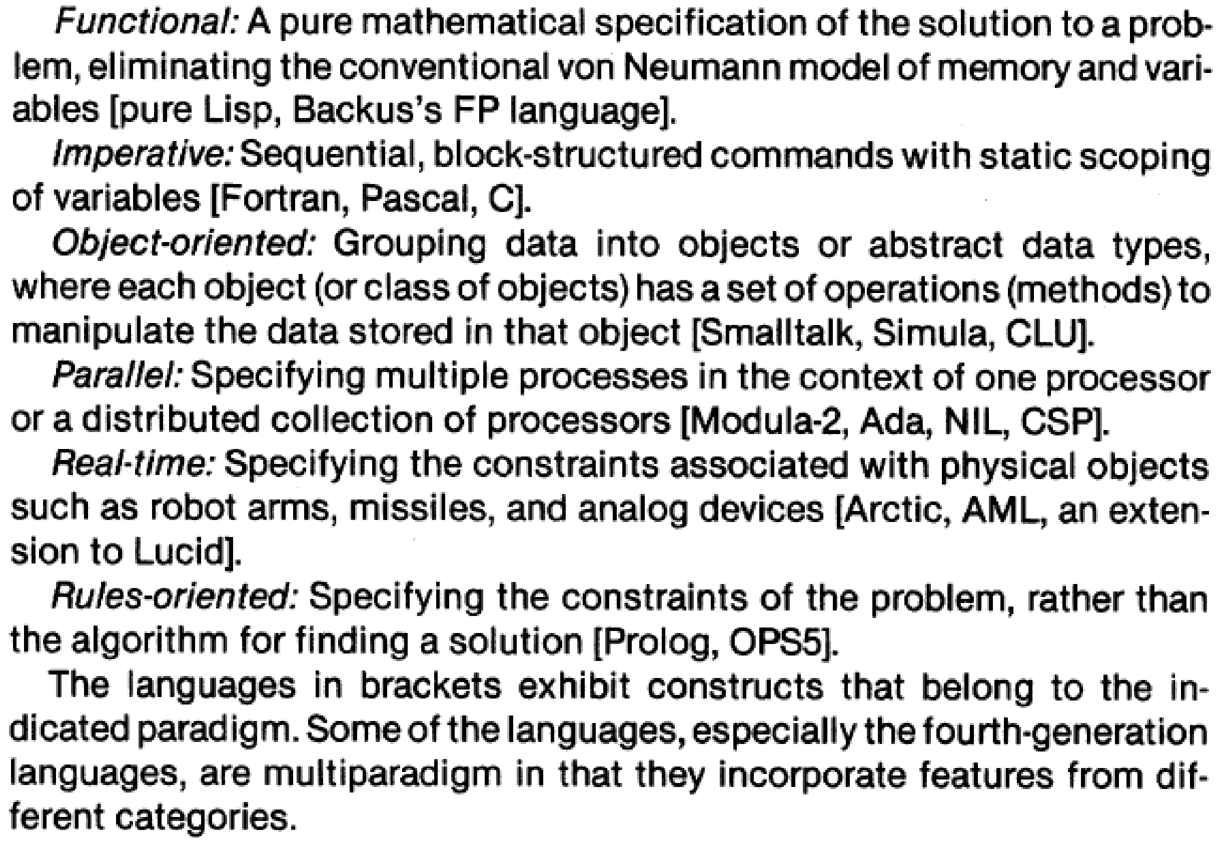
A new class of programming languages and environments is being developed to help solve this problem. They do not restrict the programmer to only one paradigm (see box on p. 8); incorporating two or more of the conventional program paradigms. For example, the Loops system, described in the first article in this issue, combines features of the Lisp, functional, rule oriented, and object-oriented paradigms. These multiparadigm systems are being created to give the programmer the right tool at the right time.

The most common multiparadigm system is the conventional operating system, which embodies several different programming languages.

Some programming paradigms

Access oriented: The specification of side-effects or demons attached to the manipulation of variables [an extension of Loops].

Data-structure-oriented: Approaching a solution by way of a single powerful data structure, such as lists (Lisp), arrays (APL), sets (SETL), relational databases (SQL).



Functional: A pure mathematical specification of the solution to a problem, eliminating the conventional von Neumann model of memory and variables [pure Lisp, Backus's FP language].

Imperative: Sequential, block-structured commands with static scoping of variables [Fortran, Pascal, C].

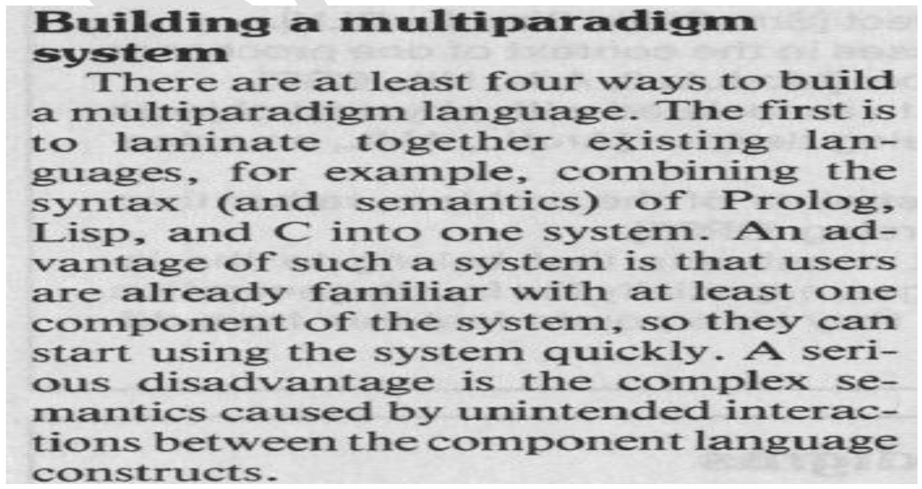
Object-oriented: Grouping data into objects or abstract data types, where each object (or class of objects) has a set of operations (methods) to manipulate the data stored in that object [Smalltalk, Simula, CLU].

Parallel: Specifying multiple processes in the context of one processor or a distributed collection of processors [Modula-2, Ada, NIL, CSP].

Real-time: Specifying the constraints associated with physical objects such as robot arms, missiles, and analog devices [Arctic, AML, an extension to Lucid].

Rules-oriented: Specifying the constraints of the problem, rather than the algorithm for finding a solution [Prolog, OPS5].

The languages in brackets exhibit constructs that belong to the indicated paradigm. Some of the languages, especially the fourth-generation languages, are multiparadigm in that they incorporate features from different categories.



Building a multiparadigm system

There are at least four ways to build a multiparadigm language. The first is to laminate together existing languages, for example, combining the syntax (and semantics) of Prolog, Lisp, and C into one system. An advantage of such a system is that users are already familiar with at least one component of the system, so they can start using the system quickly. A serious disadvantage is the complex semantics caused by unintended interactions between the component language constructs.