UNIT-3

AI PROGRAMING LANGUAGES

Introduction to LISP

LISP (full form: LISt Processing) is one of the oldest and most important programming languages used in Artificial Intelligence (AI). It was developed by John McCarthy in 1958 at MIT. LISP was specially designed for symbolic computation, logical reasoning, and processing of lists, which makes it extremely suitable for building AI applications like expert systems, natural language understanding, planning, and problem solving.

Unlike traditional programming languages that work mainly with numbers, LISP focuses on symbols, lists, recursion, and dynamic computation, which are essential for AI algorithms. The central idea of LISP is that code and data are written in the same structure (lists), making the language highly flexible and able to modify itself during execution — a very powerful feature in AI.

Definition of LISP

LISP is a high-level, functional, symbolic programming language designed primarily for Artificial Intelligence applications. It is based on processing lists as the fundamental data structure, where programs and data share the same representation (S-expressions). LISP supports dynamic typing, automatic memory management (garbage collection), recursive functions, and symbolic computation. Its unique ability to represent knowledge through lists and manipulate them efficiently makes it one of the most powerful languages for AI research, expert systems, machine reasoning, and natural language processing.

Key Characteristics of LISP (In Detail)

1. Symbolic Processing

LISP can handle **symbols** (words, names, identifiers) rather than just numbers.
This is critical in AI because AI deals with **concepts, rules, knowledge, relationships**, etc.

Example:

```
(define x 'apple)
```

Here, 'apple is a symbol, not a number.

2. List-Based Structure

The name itself suggests that LISP is all about **list processing**.
Everything in LISP is a list:

- Data is stored as lists
- Code is written as lists
- Operations are applied on lists

Example of LISP list:

```
(1 2 3 4)
```

3. S-Expressions (Symbolic Expressions)

LISP uses a very simple and uniform syntax called **S-expressions**.

Example:

```
(+ 2 3)
```

This means "add 2 and 3".

S-expressions make the language:

- Easy to parse
- Easy to manipulate

- Suitable for AI reasoning tasks

4. Functional Programming

LISP supports **functions as first-class citizens**, meaning:

- Functions can be stored in variables
- Functions can be passed as arguments
- Functions can return other functions

This makes AI algorithms easier to implement.

5. Recursion

LISP relies heavily on **recursive functions** because recursion is a natural way to process lists and solve problems logically.

Example:

```
(defun factorial (n)
  (if (= n 0) 1
    (* n (factorial (- n 1)))))
```

6. Dynamic Typing

Variables in LISP do not require type declarations.
This flexibility is perfect for AI programs where the type of data can change dynamically.

7. Garbage Collection

LISP was one of the first languages to support **automatic memory management**, allowing programmers to focus on logic rather than memory.

8. Code-as-Data (Homoiconicity)

A unique feature of LISP is **code is written in the same structure as data** (both are lists).
This allows programs to:

- Modify their own code
- Generate new code
- Adapt during runtime

This self-modifying ability is very useful in expert systems and AI planning systems.

9. Macros

LISP provides a powerful macro system that allows users to create new syntactic forms.
Macros help in building custom AI languages or tools on top of LISP.

Applications of LISP in AI

1. Expert Systems

LISP was heavily used to build early expert systems like:

- MYCIN
- DENDRAL

2. Natural Language Processing (NLP)

Good for grammar processing, parsing, and language rules.

3. Machine Reasoning

Rule-based systems and logical inference systems use LISP.

4. Robotics

Used for path planning, symbol handling, and decision making.

5. AI Research

Universities and research labs use LISP for modeling intelligent behavior.

Advantages of LISP

- Highly flexible
- Excellent for symbolic and logical AI
- Supports rapid prototyping
- Dynamic and interactive
- Best for reasoning and knowledge representation

Disadvantages of LISP

- Unusual syntax (lots of parentheses)
- Slower than C/C++ in numerical tasks
- Not widely used in industry today (mostly academia)

Basic List Manipulation Functions

In many programming languages—especially LISP, functional languages, and some AI-based systems—lists are the most fundamental data structure. They store sequences of items such as numbers, symbols, or sub-lists.

To work with lists, several basic list manipulation functions are used. These functions allow you to:

- Create lists
- Access elements
- Add or remove items
- Combine or divide lists
- Modify list structure

Below is a very detailed explanation of the most important list manipulation functions.

1. CAR (or FIRST)

CAR is a list operation that returns the first element of a list. It is used to extract the “head” or “front” portion of a list. If the list is empty, CAR is undefined. In AI languages like LISP, CAR is essential because it helps in breaking down complex symbolic structures and processing lists recursively.

Example:

```
(car '(10 20 30 40))
```

```
Output: 10
```

2. CDR (or REST)

CDR returns the remainder of the list after removing the first element. It gives the “tail” of the list. CDR is critical in list processing because it allows iterative and recursive traversal of lists. Together with CAR, it enables decomposition of any list structure.

Example:

```
(cdr '(10 20 30 40))
```

```
Output: (20 30 40)
```

3. CONS

CONS is a fundamental list-construction function that adds a new element to the beginning of an existing list, returning a new list. CONS is used for creating lists during recursive algorithms and building symbolic expressions dynamically. It is a building block of all list structures in LISP-style languages.

Example:

(cons 5 '(10 20 30))

Output: (5 10 20 30)

4. LIST

LIST is a constructor function that creates a new list containing all the given elements. Unlike CONS, which adds a single element to the front, LIST creates an entire list in one step. It is used when you want to define a list explicitly.

Example:

(list 1 2 3 4)

Output: (1 2 3 4)

5. APPEND

APPEND joins two or more lists into a single combined list. It does not modify the original lists but returns a new list containing all elements from the input lists in order. APPEND is widely used in AI algorithms for combining partial solutions, building search paths, and merging data.

Example:

(append '(1 2) '(3 4 5))

Output: (1 2 3 4 5)

6. LENGTH

LENGTH returns the total number of elements in a list. This function is essential for iterative algorithms, list traversal, and determining stopping conditions in recursion.

Example:

(length '(5 10 15))

Output: 3

7. REVERSE

REVERSE returns a new list with all elements in the opposite order. This function is frequently used in algorithms for stack operations, backtracking, and list normalization.

Example:

(reverse '(1 2 3))

Output: (3 2 1)

8. NTH (or ELEMENT-AT in other languages)

NTH retrieves the element at a specified index (position) in a list. In many systems, indexing starts at 0. It is useful for accessing list items directly without traversing the whole list.

Example:

(nth 2 '(10 20 30 40))

Output: 30

9. MEMBER

MEMBER checks whether a particular value exists inside a list. If the element is found, it returns the sub-list starting from that element. If not, it returns NIL or false. It is used in pattern matching, rule checking, and search operation in AI.

Example:

(member 30 '(10 20 30 40))

Output: (30 40)

10. MAPCAR or MAP

MAPCAR applies a function to every element of the list, returning a new list of results. This operation is used for transforming lists, feature extraction, or applying uniform processing logic in AI tasks.

Example:

(mapcar 'square '(1 2 3 4))

Output: (1 4 9 16)

Why List Manipulation Functions Are Important

- They allow recursive data processing
- They help break down complex symbolic expressions
- Used heavily in expert systems
- Enable tree traversal, search, planning, and reasoning
- Fundamental in LISP, Prolog, Scheme, and AI languages
- Support functional and declarative programming styles

Input, Output, and Local Variables in AI

Artificial Intelligence programs—especially those written in **LISP**, **Prolog**, **Python**, or any rule-based or logic-based system—work by taking **input**, processing data using internal rules, and generating **output**.
During this processing, they often use **local variables** to store temporary values.

Let's understand each concept in detail.

1. INPUT in AI

Definition

In AI, **Input** refers to the data, facts, or instructions that an AI system receives from the user, from sensors, or from another program in order to start processing or decision-making.

- Input is the **starting point** of any AI computation.
- AI systems cannot generate intelligence unless they have **data to analyze**.
- Input can be:
    - a **question** asked by a user (e.g., “Find the shortest route”)
    - a **fact** (e.g., “Tom is a cat.”)
    - **sensor data** (temperature, images, videos)
    - **knowledge-base rules**
    - data stored in files, databases, or memory
- In symbolic AI (LISP, Prolog), inputs are often **expressions, predicates, or lists**.
- In machine learning, input is typically **numerical data**, **images**, **text**, or **audio signals**.

Purpose of Input

- To **provide the initial information** from which the AI system can derive conclusions.
- To **describe the problem** that AI must solve.
- To **trigger reasoning**, inference, learning, or computation.

2. OUTPUT in AI

Definition

Output is the **final result**, **decision**, **solution**, or **action** produced by an AI program after processing the given inputs.

- Output is the **end product** of an AI system's reasoning.
- It may be:
    - a **numeric answer**
    - a **list or data structure**
    - a **diagnosis**
    - the **best move** in a game (e.g., chess)
    - a **recognized speech text**
    - a **predicted class** in machine learning
    - a **recommended action**
- Output must be **meaningful**, **accurate**, and **related** to the given input.
- In symbolic AI, output can also be a **modified list**, a **logical truth value**, or a **derived conclusion**.

Purpose of Output

- To **return the solution** of the problem.
- To show the **effect of applying rules or algorithms**.
- To **communicate the AI system’s reasoning** to the user or another program.

3. LOCAL VARIABLES in AI

Local Variables are **temporary variables** used **inside a function, rule, or algorithm** in an AI program.
They **exist only during execution** and are **not accessible outside that specific function or rule**.

Long, Detailed Explanation

- Local variables store temporary data that the AI system needs **only for a short period**.
- They help the system:
    - match patterns,
    - hold intermediate results,
    - store temporary values during recursion,
    - represent placeholders in rules.
- Local variables are essential for **problem-solving**, **pattern matching**, and **search algorithms**.

Key Properties

1. **Scope is limited**
   They can only be accessed inside the block or rule where they are declared.
2. **Lifetime is temporary**
   They disappear after the function or rule finishes running.
3. **Used heavily in AI**
   Especially in:
    - LISP functions
    - Prolog rules
    - Search algorithms (DFS, BFS)

   - Machine learning functions (training loops)

Example (Conceptual)

If an AI function calculates a path:

```
path(start, end):
   temp_distance = ...   ← local variable
   best_route = ...      ← local variable
   return best_route
```

List and Array in Artificial Intelligence (AI) – Detailed Explanation

In AI, data must be stored, organized, and processed efficiently. Two fundamental data structures used widely in AI programming are:

- **List** (mostly in symbolic AI: LISP, Prolog)
- **Array** (mostly in numerical AI: ML, DL, Image processing)

Both have different purposes and advantages.

1. LIST in AI

A **List** in AI is an ordered collection of elements that can contain different types of data, including numbers, symbols, strings, and even other lists.
Lists are dynamic, flexible, and widely used in symbolic AI for knowledge representation, problem-solving, pattern matching, recursive processing, and representing complex structures like trees and graphs.

✔ 1. Ordered Structure

A list stores elements in a sequence:

(A B C D)

(1 2 3 4)

✔ 2. Heterogeneous Data

A list can store different types of elements:

(10 "RAM" TRUE (1 2 3))

✔ 3. Dynamic Size

The size of a list can grow or shrink at runtime.

✔ 4. Foundation of Symbolic AI

In LISP, nearly everything (programs, functions, data) is represented using lists.

✔ 5. Nested Lists

A list can contain sub-lists:

((A B) (C D) (E (F G)))

✔ 6. Supports Recursion

AI problems like tree search, parsing, and symbolic reasoning rely heavily on recursion, and lists are ideal for recursive operations.

📌 Common List Operations (Especially in LISP)

- **CAR** → returns the **first element**
- **CDR** → returns the **rest of the list**
- **CONS** → adds an element to the front of a list
- **APPEND** → joins two lists
- **LENGTH** → returns number of elements

📌 **Uses of Lists in AI**

- Knowledge representation
- Natural language processing
- Expert systems
- Search trees and state-space representation
- Logic and rule-based systems
- Symbolic computation

2. ARRAY in AI

An **Array** in AI is a data structure consisting of a fixed-size sequence of elements of the same data type stored in continuous memory locations.
Arrays allow fast, direct access to elements using indexes, making them essential for numerical computations, machine learning algorithms, deep learning models, and image processing.

✔ 1. Fixed Size

Arrays are usually fixed in size (unless using dynamic arrays).

✔ 2. Homogeneous Data

All elements in an array are of the same type (e.g., all integers, all floats).

✔ 3. Fast Random Access

You can quickly access any element using an index:

arr[0], arr[1], arr[2]

✔ 4. Continuous Memory

Data is stored in one block of memory, making computations faster.

✓ 5. Essential for Numerical AI

Machine learning and deep learning mostly depend on arrays:

- Vectors
- Matrices
- Tensors (multi-dimensional arrays)

📌 Uses of Arrays in AI

- Image processing (images are pixel arrays)
- Neural networks (weights stored as arrays)
- Mathematical operations (matrix multiplication)
- Numerical analysis
- Reinforcement learning grid environments
- Feature representation in ML models

Difference Between List and Array in AI

Feature	**List**	**Array**
Data Type	Mixed (heterogeneous)	Same type (homogeneous)
Size	Dynamic	Usually fixed
Memory	Linked or nested structure	Continuous block
Access Speed	Slower	Very fast

Feature	**List**	**Array**
AI Area	Symbolic AI	Numerical AI (ML/DL)
Flexibility	Very flexible	Less flexible
Usage	Hierarchical data, recursion	Vector/matrix operations

Simple program in LISP

Here are **simple and easy LISP programs** that beginners can understand quickly.
I will give **multiple basic examples** with clear explanations.

1. Print "Hello World"

```
(print "Hello World")
```

Explanation:
print is used to display text on the screen.

2. Add Two Numbers

```
(+ 10 20)
```

Explanation:
In LISP, the operator comes first → (operator operand1 operand2)
So (+ 10 20) means **10 + 20**.

3. Subtract Two Numbers

```
(- 50 25)
```

4. Multiply Two Numbers

```
(* 5 6)
```

5. Find Square of a Number

```
(defun square (x)
  (* x x))
```

Use it like:

```
(square 5)
```

6. Check if Number is Positive

```
(defun positive? (x)
  (if (> x 0)
      "Positive"
      "Not Positive"))
```

7. Factorial Program (Simple Recursion)

```
(defun fact (n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

Use:

```
(fact 5)
```

8. Simple List Program (Find First Element)

```
(car '(10 20 30 40))
```

Output:

```
10
```

9. Add element to front of list

```
(cons 5 '(10 20 30))
```

Output:

```
(5 10 20 30)
```

10. Sum of Elements in a List

```
(defun sum-list (lst)
  (if (null lst)
      0
      (+ (car lst) (sum-list (cdr lst)))))
```

Use:

```
(sum-list '(1 2 3 4))
```

If you want, I can also provide:
✓ LISP programs on **AI topics**
✓ List manipulation programs
✓ Pattern matching programs
✓ Short notes for exam

1. Knowledge Representation (Facts & Rules)

✓ Program: Define Facts and Query Them

```
(setq facts
      '((animal cat)
        (animal dog)
        (animal cow)
        (bird sparrow)
        (bird parrot)))
```

Query Function

```
(defun isa? (x type)
  (member (list type x) facts :test #'equal))
```

Use:

```
(isa? 'cat 'animal)
```

2. Simple Expert System Rule (AI Decision Making)

✓ Program: Diagnose Fever

```
(defun diagnose (temp)
  (if (> temp 99)
      "Patient has fever"
      "No fever"))
```

Use:

```
(diagnose 102)
```

3. Heuristic Search: Find Minimum Value (AI-type heuristic)

✓ Program: Best Node Selection

```
(defun find-min (lst)
  (if (null lst)
      nil
      (if (< (car lst) (car (cdr lst)))
          (car lst)
          (find-min (cdr lst)))))
```

4. Generate Successor States (AI State-Space)

✔ Program: Successors of a Number

(Example: each state generates +1 and -1)

```
(defun successors (n)
  (list (+ n 1) (- n 1)))
```

Use:

```
(successors 5)
```

5. Depth-First Search (DFS) on a Tree

✔ Graph Representation

```
(setq graph
      '((a b c)
        (b d e)
        (c f)
        (d)
        (e)
        (f)))
```

✔ DFS Function

```
(defun dfs (start goal)
  (cond ((eq start goal) (list start))
        (t (dfs-helper start goal '()))))

(defun dfs-helper (node goal visited)
  (if (member node visited)
    nil
    (let ((children (cdr (assoc node graph))))
      (if (null children)
          nil
          (dfs-children children goal (cons node visited))))))
(defun dfs-children (children goal visited)
  (cond
    ((null children) nil)
    (t (let ((path (dfs-helper (car children) goal visited)))
         (if path
             (cons (car children) path)
             (dfs-children (cdr children) goal visited))))))
```

6. Pattern Matching (Simple AI pattern matcher)

```
(defun match (pattern input)
  (equal pattern input))
```

Example:

```
(match '(i love ai) '(i love ai))
```

7. Natural Language Processing (Tokenization)

✔ Split a Sentence into Words

```
(defun tokenize (str)
  (remove "" (split-sequence #\space str)))
```

Use:

```
(tokenize "AI is powerful")
```

8. Rule-Based System (IF–THEN logic)

✓ Simple Rule: Weather Advice

```
(defun weather-advice (weather)
  (cond ((equal weather 'rain) "Take umbrella")
        ((equal weather 'sunny) "Wear sunglasses")
        ((equal weather 'cold) "Wear jacket")
        (t "Unknown weather")))
```

9. AI Recursion Example: Fibonacci

Used in dynamic programming and AI problem solving.

```
(defun fib (n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

10. AI List Processing (Remove duplicates)

```
(defun remove-duplicates-ai (lst)
  (if (null lst)
      nil
      (cons (car lst)
            (remove-duplicates-ai
        (remove (car lst) (cdr lst))))))
```

Introduction to PROLOG

PROLOG (short for **PROgramming in LOGic**) is a **logic-based, declarative programming language** widely used in Artificial Intelligence (AI), computational linguistics, expert systems, natural language processing, knowledge representation, and symbolic reasoning.
Unlike traditional programming languages where you write step-by-step instructions (procedural programming), PROLOG allows you to describe the **facts**, **rules**, and **relationships** of a problem, and the system uses **logical inference** and **automated reasoning** to find answers.

It is based on **Predicate Logic** and uses a computational mechanism known as **backtracking search** to automatically determine whether a query is true, and if so, what variable bindings satisfy that query.

PROLOG is fundamentally different from languages like C, Java, or Python.
Instead of telling the computer *how* to do something, you describe *what* you want, and PROLOG figures out the "how" logically.

1. Declarative Language

PROLOG is a **declarative** language.
You declare:

- **Facts** → basic truths
- **Rules** → logical conditions
- **Queries** → questions asked to the system

Example fact:

father(ram, shyam).

Example rule:

grandfather(X, Z) :- father(X, Y), father(Y, Z).

In PROLOG, you focus on logic, not on algorithm steps.

2. Based on Predicate Logic

PROLOG is built on **First-Order Predicate Logic**.
Programs are written as:

- **predicates**
- **arguments**
- **relations**

Example:
likes(ram, tea) means **Ram likes tea**.

This formal logic structure makes PROLOG ideal for AI applications involving reasoning.

3. Uses a search method called Backtracking

PROLOG uses **Depth-First Search (DFS)** + **Backtracking** to answer queries.

This means:

1. PROLOG tries to match a query with facts.
2. If a rule is used, it attempts to satisfy each condition.
3. If a path fails, PROLOG automatically backtracks and tries another path.

This automated search makes it extremely powerful for:

- puzzles
- pathfinding
- reasoning tasks
- expert systems

4. Knowledge is stored as Facts, Rules, and Queries

✓ Facts

Used to store basic information.

Example:

bird(sparrow).

✓ Rules

Used to represent logic or conditions.

Example:

can_fly(X) :- bird(X), not(penguin(X)).

✓ Queries

Questions asked to PROLOG.

Example:

?- can_fly(sparrow).

5. PROLOG uses unification

Unification is the process of matching patterns.

Example:
Query:

```
?- father(X, shyam).
```

If the fact is:

```
father(ram, shyam).
```

PROLOG unifies X = ram.

This automatic pattern matching is the heart of PROLOG.

6. No Traditional Flow Control

In languages like C or Java, you write loops, conditions, functions.

But in PROLOG:

- No loops
- No if-else blocks
- No step-by-step control

Instead, **logic drives the program**, and the PROLOG engine handles control flow automatically.

7. Used Mainly in AI

PROLOG is widely used for:

- Expert systems

- Natural Language Processing (NLP)
- Theorem proving
- Knowledge representation
- Intelligent tutoring systems
- Planning and scheduling
- Semantic web technologies

It is ideal for problems where the solution requires reasoning.

8. Simple Syntax, Powerful Logic

PROLOG programs are short but powerful because logic does most of the work.

Example (Family Relations):

```
mother(sita, ram).

father(dasharath, ram).


parent(X, Y) :- mother(X, Y).

parent(X, Y) :- father(X, Y).
```

Query:

```
?- parent(sita, ram).
```

Output:

```
Yes
```

9. Automatic variable binding

In PROLOG, variables start with a capital letter.

Example:

```
?- parent(X, ram).
```