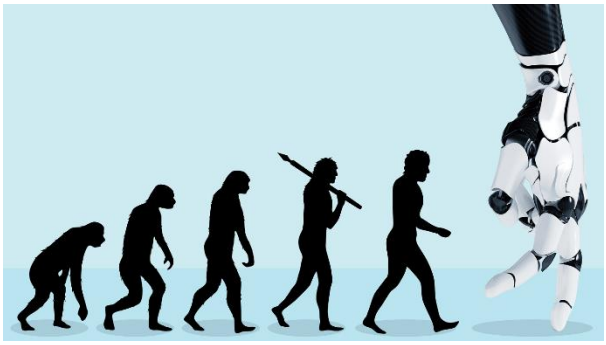
MCA 302 Artificial Intelligence

UNIT I

Contributed by: Om Shankar Mishra

MCA 302 ARTIFICIAL INTELLIGENCE

UNIT I Overview of AI:



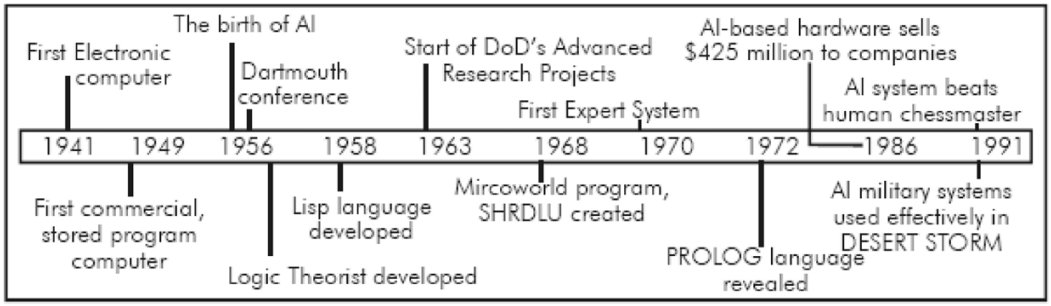
Artificial Intelligence is a branch of computer science that focuses on creating intelligent machines capable of performing tasks that typically require human intelligence. AI aims to develop systems that can reason, learn, perceive, understand natural language, and interact with the environment.

Artificial Intelligence एक computer science की branch है जो उन tasks को करने में capable है जो intelligent machines बनाने पर focused है जिनके लिए आमतौर पर human intelligence की आवश्यकता होती है। AI का aim ऐसे systems को develop करना है जो तर्क कर सकें, सीख सकें, अनुभव कर सकें, natural language को समझ सकें और environment के साथ बातचीत कर सकें।

History of AI

- Evidence of AI folklore dates back to ancient Egypt. (AI लोककथाओं के साक्ष्य प्राचीन Egypt से मिलते हैं।)
- Development of electronic computers in 1941 enabled the creation of machine intelligence. (1941 में electronic computers के विकास ने machine intelligence के निर्माण को सक्षम बनाया।)
- Term "artificial intelligence" coined in 1956 at the Dartmouth conference. ("artificial intelligence" शब्द 1956 में डार्टमाउथ conference में गढ़ा गया।)
- Slow progress initially, but AI continued to advance over the decades. (शुरू में धीमी प्रगति, लेकिन दशकों तक AI आगे बढ़ता रहा।)
- Various AI programs and their impact on technological advancements. (विभिन्न AI programs और technological advancements पर उनका प्रभाव हुआ।)

Figure 1.1: History of AI



The Beginnings of AI

- Link between human intelligence and machines observed in the early 1950s. (1950 के दशक की शुरुआत में human intelligence और machines के बीच Link देखा गया।)
- Norbert Wiener's feedback theory, exemplified by the thermostat. (नॉर्बर्ट वीनर का feedback theory, thermostat द्वारा उदाहरण दिया गया।)
- Feedback mechanisms as a foundation for intelligent behavior. (Intelligent behavior की नींव के रूप में Feedback mechanism है।)
- Development of The Logic Theorist in 1955 and its impact on AI. (1955 में Logic Theorist का विकास और AI पर इसका प्रभाव हुआ।)
- John McCarthy organized the Dartmouth conference in 1956, giving birth to the field of AI. (जॉन मैककार्थी ने 1956 में AI के क्षेत्र को जन्म देते हुए डार्टमाउथ conference का आयोजन किया।)

Major Issues in AI:

Ethics and Bias: AI systems training data से inherit biases कर सकते हैं, जिससे biased decisions और unfair outcomes हो सकते हैं।

Transparency and Explainability: Neural networks जैसे Complex AI models को समझना difficult हो सकता है, जिससे जवाबदेही और decision-making के बारे में चिंताएं बढ़ सकती हैं।

Job Displacement: Automation powered by AI can lead to job loss in certain industries, requiring strategies for upskilling and reemployment.

Privacy Concerns: AI systems often handle sensitive data, necessitating safeguards to protect user privacy and data security.

Autonomous Systems: As AI systems become more autonomous, questions arise about control and accountability in case of errors or accidents.

AI in Different Sectors:

Healthcare: AI aids in disease diagnosis, drug discovery, and personalized treatment plans.

Finance: AI is used for fraud detection, algorithmic trading, and customer service in the financial sector.

Transportation: Self-driving cars and intelligent traffic management systems are emerging applications of AI.

Retail: AI-powered recommendation systems and inventory management enhance customer experience.

The AI problems

Computing Power Demand: Machine Learning and Deep Learning require significant processing power, often unaffordable for developers. Cloud options help but come at a cost, limiting accessibility.

Trust Deficit and Awareness Gap: Understanding how AI models make predictions is complex, fostering skepticism. Many people are unaware of AI's integration into everyday items, hindering trust and awareness.

Limited Knowledge and Adoption: The potential of AI remains largely untapped outside tech circles. Small and Medium Enterprises (SMEs) miss out on benefits due to inadequate awareness and knowledge.

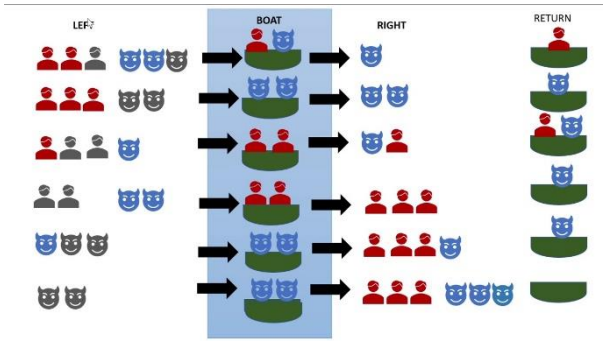
Human-Level Performance: Achieving human-level accuracy in AI tasks demands meticulous fine-tuning, massive datasets, and robust algorithms. Pre-trained models fall short of replicating human performance.

Data Privacy Concerns: AI's foundation is data, but breaches can lead to misuse. Innovative approaches, such as on-device training, aim to enhance data privacy and security.

Addressing Bias: Biased training data leads to biased AI outcomes. Algorithms must be developed to identify and mitigate biases, ensuring fairness and inclusivity.

Data Scarcity and Ethical Dilemmas: Misuse of user data prompted stricter regulations, resulting in data scarcity. AI advancements strive to deliver accurate results even with limited, locally sourced data while upholding ethical standards.

Three missionaries and three cannibals are on one side of a river along with a boat. They want to cross to the other side. The boat can carry at most two people at a time. If the cannibals ever outnumber the missionaries on either bank of the river (even for a brief moment), the cannibals will eat the missionaries. The goal is to find a sequence of boat trips that successfully transports all the missionaries and cannibals to the other side of the river without violating the cannibal-missionary ratio at any point.



State Space: In the context of the "Missionaries and Cannibals" problem, the state space represents all possible configurations of people and the boat on both sides of the river. Each state can be represented by a tuple of six values: (**M1, C1, B, M2, C2, Side**), where:

M1 is the number of missionaries on the starting side.

C1 is the number of cannibals on the starting side.

B indicates the position of the boat (0 for the starting side, 1 for the other side).

M2 is the number of missionaries on the other side.

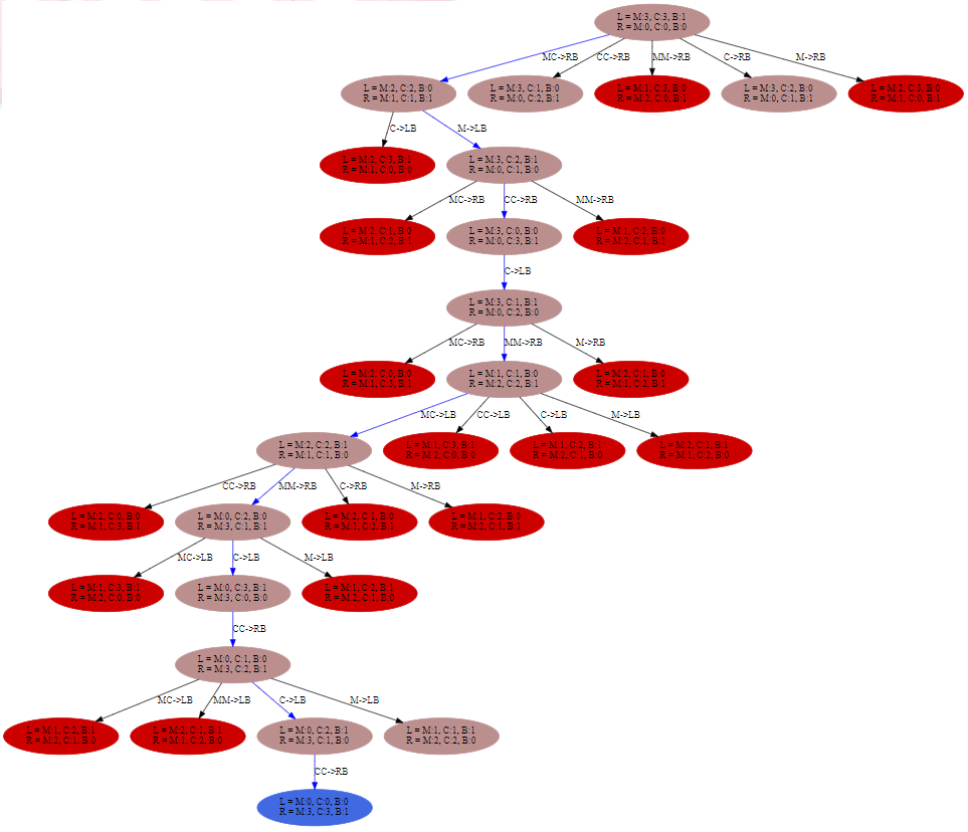
C2 is the number of cannibals on the other side.

Side indicates the current side of the river (0 for the starting side, 1 for the other side).

Third problem: Missionaries and cannibals

Transfer number	Starting shore[18]	Transfer	ending shore
Initial conditions	*MMM CCC*		
1	*MM CC*	*M C →*	
2	*MM CC*	*← C*	*M*
3	*MM C*	*CC →*	*M*
4	*MM C*	*← C*	*M C*
5	*M C*	*M C →*	*M C*
6	*M C*	*← C*	*MM C*
7	*M*	*CC →*	*MM C*
8	*M*	*← C*	*MM CC*
9		*M C →*	*MM CC*
Final conditions			*MMM CCC*

The initial state is (3, 3, 0, 0, 0, 0) (3 missionaries and 3 cannibals on the starting side with the boat), and the goal state is (0, 0, 1, 3, 3, 1) (all missionaries and cannibals on the other side).



State Transitions: Valid state transitions are determined by the rules of the problem:

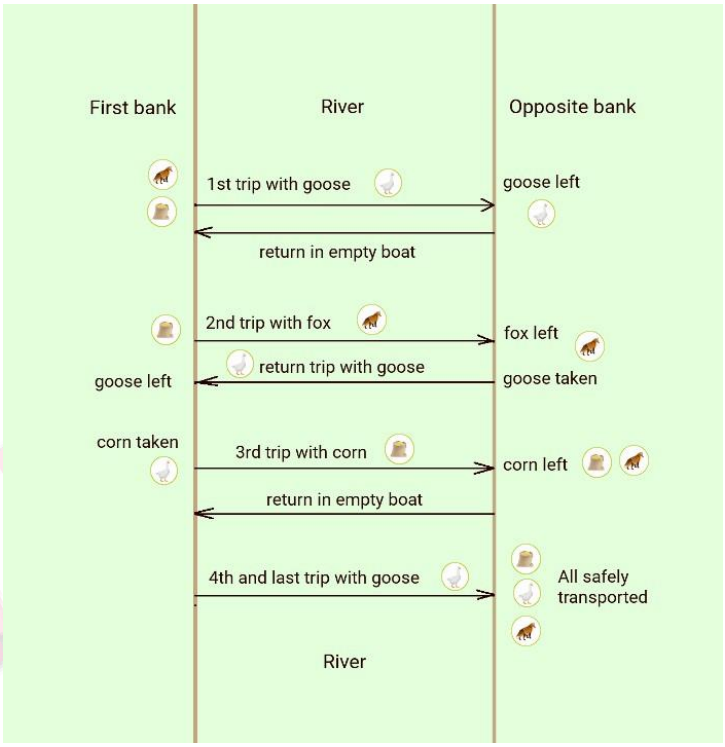
- The boat can carry either one or two people, either missionaries or cannibals.
- The number of cannibals on each side should not exceed the number of missionaries on that side, otherwise, the cannibals will eat the missionaries.
- The goal is to find a sequence of state transitions that leads from the initial state to the goal state while satisfying the constraints.

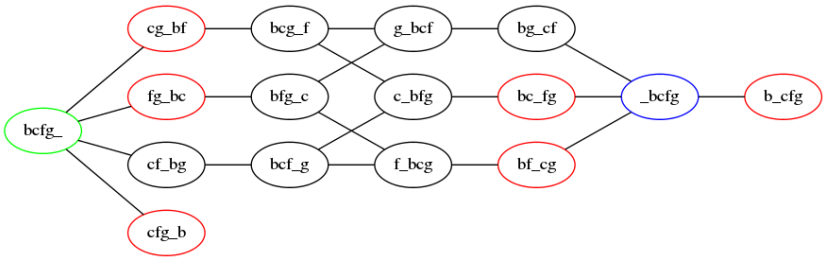
Consider the farmer, fox, goose, grain puzzle. In this puzzle a farmer wishes to cross the river taking his fox, goose and grain with him. he can use a boat which will accomodate only the farmer and one possession? If the fox is left alone with the goose , the goose will be eaten and if the goose is left alone with the grain, grain will be eaten. Draw a state space tree for this puzzle using left bank and right bank to denote the left and right banks respectively.

The goal is to find a series of valid moves that allow the farmer to safely transport all three items (the fox, the goose, and the grain) from one side of the river to the other. The puzzle can be solved using a specific sequence of moves. Here's one possible solution:

Let's label the two sides of the river as "A" and "B".

Initial setup: Farmer, Fox, Goose, and Grain are on side A.

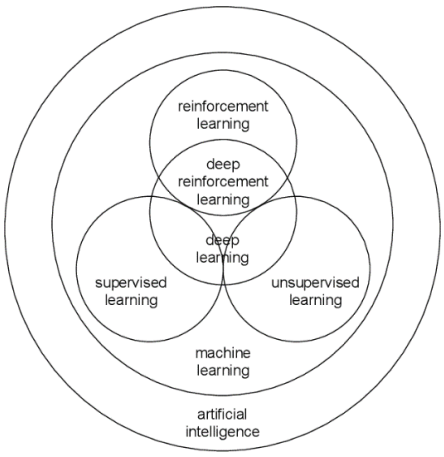





```
        Start
       /  |  \
      /   |   \
     F   G   G   F
    /\/    \/\
   /  X     X  \
  /   |     |   \
 FG   G     F   FG
 |    |     |   |
 X    F     FG  X
 |    |     |   |
 X    FG    F   X
 |    |     |   |
 G    X     X   G
 |    |     |   |
 X    X     X   X
```

What is an AI technique

An AI technique refers to a specific method, approach, or algorithm used to address problems and challenges in the field of artificial intelligence (AI). These techniques are designed to enable machines to simulate human cognitive functions and perform tasks that typically require human intelligence. AI techniques are the building blocks that power various AI applications and systems. Here are some common AI techniques:



1. Machine Learning (ML): Machine learning is a subset of AI that focuses on enabling machines to learn from data and improve their performance over time. There are several types of machine learning techniques, including:

- **Supervised Learning**: Training models on labeled data to make predictions or classifications.
- **Unsupervised Learning**: Discovering patterns and relationships in unlabeled data.
- **Reinforcement Learning**: Training agents to make decisions by interacting with an environment and receiving rewards or penalties.

2. Neural Networks and Deep Learning: Neural networks are computational models inspired by the human brain's structure and function. Deep learning is a subset of neural networks that involves training models with multiple layers (deep architectures) to automatically learn hierarchical features from data. These techniques have revolutionized tasks like image recognition, natural language processing, and more.

3. Natural Language Processing (NLP): NLP techniques focus on enabling computers to understand, interpret, and generate human language. These techniques include:

- **Text Classification:** Categorizing text into predefined categories or labels.
- **Named Entity Recognition**: Identifying entities like names, dates, and locations in text.
- **Sentiment Analysis**: Determining the sentiment or emotion expressed in text.

4. Computer Vision: Computer vision techniques enable machines to interpret and understand visual information from images and videos. These techniques include:

- **Object Detection**: Locating and identifying objects within images or video frames.
- **Image Segmentation**: Dividing an image into segments and assigning labels to each segment.
- **Facial Recognition**: Identifying and verifying individuals based on facial features.

5. Expert Systems: Expert systems utilize knowledge representation and reasoning techniques to capture human expertise in specific domains. They provide recommendations, solutions, or decisions based on predefined rules and knowledge.

6. Genetic Algorithms: Genetic algorithms are optimization techniques inspired by the process of natural selection. They involve generating a population of potential solutions and iteratively evolving them to find optimal or near-optimal solutions.

7. Reinforcement Learning: Reinforcement learning focuses on training agents to make sequential decisions by interacting with an environment. Agents receive rewards or penalties based on their actions and learn to maximize cumulative rewards over time.

8. Fuzzy Logic: Fuzzy logic is a mathematical framework for dealing with uncertainty and imprecision. It allows for gradual degrees of truth rather than binary true/false values.

Types of AI:

Narrow or Weak AI: Specialized in a specific task, like voice assistants or recommendation systems.

General or Strong AI: Possesses human-like intelligence and can perform any intellectual task that a human can.

Characteristics of AI applications

AI applications exhibit several distinct characteristics that differentiate them from traditional software applications. These characteristics stem from the ability of AI systems to simulate human cognitive functions and perform tasks that typically require human intelligence. Here are the key characteristics of AI applications:

1. **Adaptability:** AI applications can adapt and learn from new data and experiences. They can adjust their behavior based on changes in their environment or the data they receive, making them more flexible and responsive over time.
2. **Learning:** AI systems have the capacity to learn from historical data and improve their performance with experience. They can identify patterns, relationships, and trends in data, allowing them to make better decisions or predictions in the future.
3. **Reasoning:** AI applications can perform logical reasoning, inference, and deduction. They can follow a set of rules or logic to reach conclusions based on available information.
4. **Problem-Solving:** AI systems excel at solving complex problems that may involve a large number of variables, factors, and possible solutions. They can search through potential solutions, evaluate them, and choose the most suitable option.
5. **Perception:** AI applications can interpret and understand sensory data from the environment. This includes processing visual data (computer vision), audio data (speech recognition), and other forms of sensory input.
6. **Interaction:** AI systems can interact with humans or other systems using natural language, visual interfaces, or other modes of communication. This allows for more intuitive and user-friendly interactions.
7. **Autonomy:** Certain AI applications can operate autonomously, making decisions and taking actions without constant human intervention. Autonomous vehicles, drones, and industrial robots are examples of AI systems with a high degree of autonomy.
8. **Adaptation to Context:** AI applications can adapt their behavior based on the context of the situation. For instance, a virtual assistant might adjust its responses based on the user's mood, preferences, or the current task.
9. **Data-Driven Decision-Making:** AI applications heavily rely on data to make informed decisions. They analyze and process large volumes of data to derive insights and support decision-making.
10. **Continual Learning:** AI systems can continue to learn and improve over time as they encounter new data and experiences. This ability allows them to stay relevant and up-to-date in dynamic environments.
11. **Emulation of Human Intelligence:** Many AI applications aim to replicate or mimic specific aspects of human intelligence, such as understanding language, recognizing patterns, or making decisions.
12. **Parallel Processing:** Some AI techniques, such as neural networks, can perform parallel processing, which enables them to handle complex computations more efficiently.

Introduction to LISP programming

LISP (LISt Processing) is a programming language known for its distinctive syntax and its focus on symbolic processing and manipulation of lists. It has been widely used in various fields, especially in artificial intelligence (AI) research and development.

Contributed by: Om Shankar Mishra

LISP (LISt Processing) is a programming language known for its symbolic processing capabilities, making it well-suited for various artificial intelligence and symbolic computing applications. Here are some examples of **applications** that have been built using LISP:

Artificial Intelligence Research: LISP has been extensively used in AI research due to its flexibility in handling symbolic data. Early AI programs and systems, like the expert system framework called "DENDRAL," were implemented in LISP.

Symbolic Mathematics and Computer Algebra Systems: LISP is used to build computer algebra systems (CAS) that manipulate algebraic expressions symbolically. One prominent example is "Maxima," an open-source CAS.

Natural Language Processing (NLP): LISP has been used to develop systems for natural language understanding and generation. The SHRDLU program, created by Terry Winograd, was an early example of a NLP system implemented in LISP.

Game Development: LISP has been used to create video games and interactive simulations. One well-known example is the game "Sorcerer," an interactive fiction game written in LISP.

Expert Systems: LISP's flexibility in representing and manipulating symbolic knowledge makes it suitable for building expert systems, which are computer systems that emulate human expertise in a specific domain. "Expert System Shell" is an example of a LISP-based tool for developing expert systems.

Automated Reasoning: LISP has been used for automated theorem proving and logical reasoning systems. The "Cyc" project aimed to create a comprehensive ontology and knowledge base for general-purpose AI, and it heavily used LISP.

Robotics: LISP has been applied to control and programming of robots. The "Flavors" object-oriented programming system for LISP was used in robotics applications.

CAD/CAM: LISP has been used in Computer-Aided Design (CAD) and Computer-Aided Manufacturing (CAM) applications due to its ability to handle complex symbolic data and manipulate geometric objects.

Education: LISP has been used in educational contexts to teach programming concepts and AI principles due to its simplicity and expressiveness.

Emacs Text Editor: The Emacs text editor, a highly extensible and customizable editor, is implemented in a dialect of LISP called Emacs Lisp. It demonstrates LISP's ability to be used in practical software tools.

Music Composition: LISP has been used in generative music composition, where algorithms create music based on predefined rules.

Syntax and numeric functions:

Syntax in LISP: LISP's syntax is distinctive due to its reliance on parentheses and prefix notation. Expressions are written in the form (operator operand1 operand2 ...). Here's a breakdown of LISP syntax **elements**:

Operator: The first element in a list is the operator, which can be a function symbol or a special form.

Operands: The following elements are operands, which can be data values, function arguments, or nested expressions.
Lists: LISP treats code as lists, and lists can be nested within other lists to any level of complexity.
Atoms: These are symbols or numbers that are not enclosed in parentheses.
Numeric Functions: LISP provides a variety of numeric functions for performing arithmetic operations. Here are some common numeric functions:

- **+:** Adds numbers together: (+ 3 5) ; Result: 8
- **-:** Subtracts numbers: (- 10 4) ; Result: 6
- ***:** Multiplies numbers: (* 2 3) ; Result: 6
- **/:** Divides numbers: (/ 10 2) ; Result: 5
- **mod:** Computes the remainder of division: (mod 10 3) ; Result: 1
- **expt:** Computes the power of a number: (expt 2 3) ; Result: 8
- **sqrt:** Calculates the square root of a number: (sqrt 25) ; Result: 5
- **abs:** Computes the absolute value of a number: (abs -7) ; Result: 7
- **max and min:** Find the maximum and minimum of a list of numbers.
  (max 3 7 2) ; Result: 7 (min 3 7 2) ; Result: 2

- These numeric functions can be used to perform basic arithmetic operations in LISP programs. They take numeric arguments and return numeric results.
- LISP uses prefix notation, so the operator always comes before its operands. The parentheses help define the structure of expressions and ensure proper evaluation.

Basic list manipulation functions:

LISP provides a set of basic functions for creating, accessing, modifying, and combining lists.

car:

- **Syntax**: (car list)
- Returns the first element (head) of a list.
- **Example**: (car '(1 2 3)) ; **Result**: 1

cdr:

- **Syntax**: (cdr list)
- Returns the rest of the list after the first element (tail).
- **Example**: (cdr '(1 2 3)) ; **Result**: (2 3)

cons:

- **Syntax**: (cons element list)
- Constructs a new list by adding an element to the front of an existing list.
- **Example**: (cons 0 '(1 2 3)) ; **Result**: (0 1 2 3)

list:

- **Syntax**: (list &rest elements)
- Creates a new list containing the specified elements.
- **Example**: (list 1 2 3) ; **Result**: (1 2 3)

nth:

- **Syntax**: (nth index list)
- Retrieves the element at the specified index in the list (indexes are zero-based).
- **Example**: (nth 1 '(1 2 3)) ; **Result**: 2

append:

- **Syntax**: (append list1 list2)
- Combines two or more lists to create a new list.
- **Example**: (append '(1 2) '(3 4)) ; **Result**: (1 2 3 4)

length:

- **Syntax**: (length list)
- Returns the number of elements in a list.
- **Example**: (length '(1 2 3)) ; **Result**: 3

reverse:

- **Syntax**: (reverse list)
- Reverses the order of elements in a list.
- **Example**: (reverse '(1 2 3)) ; **Result**: (3 2 1)

member:

- **Syntax**: (member element list &key key test test-not)
- Checks if an element is present in a list and returns the tail starting from that element.
- **Example**: (member 2 '(1 2 3)) ; **Result**: (2 3)

subseq:

- **Syntax**: (subseq sequence start end)
- Returns a subsequence of a sequence (list or string) from the start index (inclusive) to the end index (exclusive).
- **Example**: (subseq '(1 2 3 4 5) 1 4) ; **Result**: (2 3 4)

Function Definitions:

Function in Lisp use the defun special form. It allows you to define named functions that can later be called with arguments. The basic syntax of a defun expression is:

```
(defun function-name (parameters)
  "Optional documentation string"
  body)
```

function-name: The name of the function being defined.

parameters: The list of input parameters that the function takes.

"Optional documentation string": A string that can provide information about the function's purpose and usage. It's optional but helpful for documentation.

body: The code that defines what the function does.

For example:

```
(defun add-nums (x y)
  "This function adds two numbers."
  (+ x y))
```

Lambda Expressions:

Lambda expressions are used to create anonymous functions in Lisp. These are functions that don't have a name and are usually created on-the-fly for specific purposes. The basic syntax of a lambda expression is:

```
(lambda (parameters)
  body)
```

Lambda expressions are often used when a small function is needed as an argument to another function or when a function is being returned from another function.

For **example**, let's say you want to square a list of numbers using the map function:

```
(mapcar (lambda (x) (* x x)) '(1 2 3 4 5))
```

In this example, the lambda expression (lambda (x) (* x x)) defines an anonymous function that squares its input.

Predicates and conditionals

Predicates and conditionals are essential concepts in LISP programming that allow you to make decisions and perform different actions based on certain conditions. Predicates are functions that return true or false based on the evaluation of a condition, and conditionals are constructs that help you control the flow of your program.

Predicates:

atom:

- **Syntax**: (atom expression)
- Returns true if the expression is an atom (a symbol or a number), otherwise false.
- **Example**:
  - (atom 'symbol) ; **Result**: T
  - (atom '(1 2 3)) ; **Result**: NIL

null:

- **Syntax**: (null expression)
- Returns true if the expression is nil (empty list), otherwise false.
- **Example**:
  - (null '()) ; **Result**: T
  - (null '(1 2 3)) ; **Result**: NIL

equal:

- **Syntax**: (equal object1 object2)
- Returns true if the two objects are structurally identical, otherwise false.
- **Example**: (equal '(1 2 3) '(1 2 3)) ; **Result**: T

Conditional Constructs:

if:

- **Syntax**: (if condition then-expression else-expression)
- Evaluates the condition. If true, evaluates and returns the then-expression; otherwise, evaluates and returns the else-expression.
- **Example**: (if (< 5 10) 'yes 'no) ; Result: YES

cond:

Syntax:

```
(cond
  (condition1 expression1)
  (condition2 expression2)
  ...
  (t default-expression))
```

Example:

```
(cond
  ((< x 0) 'negative)
  ((> x 0) 'positive)
  (t 'zero))
```

Evaluates a series of conditions and their corresponding expressions. Returns the value of the expression whose condition is true. If no condition is true, the default-expression is evaluated and returned.

case:

Syntax:

```
(case key
  (key1 expression1)
  (key2 expression2)
  ...
  (otherwise-expression))
```

Example:

```
(case x
  (1 'one)
  (2 'two)
  (otherwise 'other))
```

Compares the key against a set of keys and evaluates the expression associated with the first matching key. If no key matches, evaluates the otherwise-expression.

Conditionals allow you to control the flow of your program based on different conditions, enabling you to create dynamic and responsive behavior. Predicates play a crucial role in evaluating conditions to make informed decisions.

Input output and Local variables:

Input, output, and local variables are important aspects of programming, including in LISP. They allow you to interact with users, process data, and manage information within your programs. Here's how these concepts work in LISP:

Input and Output: LISP provides functions for input and output operations that allow you to interact with users through the terminal or console.

read:

Syntax: (read)

Reads a single expression from the user (in the form of a list) and returns it.

Example: (setq x (read))

write and print:

Syntax: (write expression) and (print expression)

Display the expression in a human-readable format on the output stream.

Example:

```
(write x)
(print "Hello, LISP!")
```

format:

Syntax: (format stream format-string &rest arguments)

Allows you to format and display data using a format string. Similar to the printf function in other languages.

Example: (format t "The value of x is ~a" x)

Local Variables: Local variables are used to store temporary values within a specific scope, such as a function. LISP uses the let construct to define local variables.

let:

Syntax:

```
(let ((variable1 value1)
     (variable2 value2)
     ...)
  body)
```

Example:

```
(let ((x 10)
      (y 20))
  (+ x y)) ; Result: 30
```

Defines local variables within the scope of the let expression and assigns initial values to them.

setq:

Syntax: (setq variable value)

Assigns a new value to an existing variable. This has a global effect, so use it with caution.

Example: (setq x 42)

Local variables help you manage data within specific blocks of code without affecting variables in other parts of your program. They promote modularity and reduce potential conflicts.

Iteraction and Recursion:

Interactivity and recursion are important concepts in programming, including in LISP. They allow you to create dynamic and flexible programs that can handle complex tasks.

Interaction: Interactivity involves allowing your program to communicate with users, receive input, and provide output. In LISP, you can achieve interaction through various input/output functions.

read-line:

Syntax: (read-line)

Reads a line of text input from the user and returns it as a string.

Example: (setq name (read-line))

princ:

Syntax: (princ expression)

Displays an expression without newline characters (similar to print).

Example: (princ "Enter a number: ")

terpri:

Syntax: (terpri)

Outputs a newline character, creating a line break in the output.

Example: (terpri)

Recursion: Recursion is a powerful technique where a function calls itself to solve a problem. In LISP, recursion is widely used due to its ability to handle complex and repetitive tasks.

Recursive Function:

Example:

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

In this example, the factorial function calculates the factorial of a number using recursion.

Base Case and Recursive Case:

Recursive functions typically have a base case (a condition that stops recursion) and a recursive case (where the function calls itself with a modified argument).

Example:

```
(defun fibonacci (n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (t (+ (fibonacci (- n 1)) (fibonacci (- n 2))))))
```

- The fibonacci function calculates the nth Fibonacci number using recursion and base cases for 0 and 1.
- Recursion allows you to solve problems that involve repetitive patterns or hierarchies. However, efficient recursive solutions require careful consideration of termination conditions and memory usage.

Property Lists and Arrays:

Property lists and arrays are data structures used in LISP to store and organize information. They provide different ways to manage and access data, depending on the requirements of your program.

Property Lists (Plists): A property list is an associative data structure that allows you to associate values with specific keys (symbols). Each key-value pair is called a property. Plists are useful for storing and retrieving information based on symbolic keys.

Creating a Plist:

```
(setq person-plist '(:name "Alice" :age 30 :city "Wonderland"))
```

Accessing Values in a Plist:

```
(getf person-plist :name) ; Returns "Alice"
(getf person-plist :age)  ; Returns 30
```

Adding or Modifying Properties:

```
(setq person-plist (plist-put person-plist :occupation "Engineer"))
```

Arrays: An array is an ordered collection of elements, each of which can be accessed using an index. Arrays are suitable for situations where you need to access elements by their position.

Creating an Array:

```
(setq my-array #(10 20 30 40 50))
```

Accessing Elements in an Array:

```
(aref my-array 2) ; Returns 30
```

Modifying Elements in an Array:

```
(setf (aref my-array 3) 45)
```

Multidimensional Arrays: LISP arrays can have multiple dimensions, allowing you to create matrices or more complex data structures.

Creating a Multidimensional Array:

```
(setq matrix #( #(1 2 3)
                #(4 5 6)
                #(7 8 9) ))
```

Accessing Elements in a Multidimensional Array:

```
(aref matrix 1 2) ; Returns 6
```

Both property lists and arrays have their own use cases:

- Use property lists when you need to associate properties (keys) with values in a flexible and symbolic way.
- Use arrays when you need ordered, indexed access to elements, especially in cases involving multi-dimensional data.

Write a LISP program to convert centigrade temperature to Fahrenhit

```
(defun celsius-to-fahrenheit (celsius)
  (* (+ celsius 32) 9/5))
(defun main ()
  (format t "Celsius to Fahrenheit Temperature Converter~%")
  (loop
    (format t "Enter temperature in Celsius (or 'exit' to quit): ")
    (let ((input (read-line)))
      (if (equal (string-trim input) "exit")
          (return)
          (let ((celsius (parse-integer input :junk-allowed t)))
            (if celsius
                (format t "Temperature in Fahrenheit: ~a~%" (celsius-to-fahrenheit celsius))
                (format t "Invalid input. Please enter a valid numeric temperature or 'exit'.~%")))))))

(main)
```