**Concepts of Programming Languages**

**Implementation of Sudoku Solver in multiple languages**

Table of Contents

[Problem Statement 3](#_Toc135245141)

[Overview 3](#_Toc135245142)

[Sudoku Solver in Java 3](#_Toc135245143)

[Results: 6](#_Toc135245144)

[Flowchart: 7](#_Toc135245145)

[Sudoku Solver in Lisp 7](#_Toc135245146)

[Results: 9](#_Toc135245147)

[Flowchart: 10](#_Toc135245148)

[Sudoku Solver in Prolog 11](#_Toc135245149)

[Results 14](#_Toc135245150)

[Flowchart 15](#_Toc135245151)

[Comparison 15](#_Toc135245152)

[Imperative Programming: 15](#_Toc135245153)

[Functional Programming: 16](#_Toc135245154)

[Logic Programming: 16](#_Toc135245155)

[Conclusion 17](#_Toc135245156)

[Appendix 18](#_Toc135245157)

[Java 18](#_Toc135245158)

[Lisp 22](#_Toc135245159)

[Prolog 23](#_Toc135245160)

# Problem Statement

The goal of this assignment is to demonstrate an understanding of the different programming paradigms such as functional, imperative, and logic programming. This is accomplished by creating a sudoku solver in 3 languages, each from a different paradigm. For the imperative language, Java was selected, Lisp was used as a functional language and finally, Prolog was used as the logic programming language. Each paradigm and therefore language, has its own unique attributes and features ending in a unique implementation for each language. Overall, however, each language follows the same principle of using backtracking to solve the sudoku, making sure that 3 main rules are followed. Each row, column, and 3x3 square is a set of 9 distinct numbers.

# Overview

The sudoku solver here takes a naïve approach to solving the puzzle by checking every possible value. This results in a high algorithmic time complexity, but this is understandable as this is an NP problem. The complexity can be improved by using heuristics to lead the solver in the right direction, but this seems outside this project’s scope.

# Sudoku Solver in Java

|  |  |
| --- | --- |
| The sudoku function is the main function and takes no parameters. It simply must only be called. Once it is called, it calls the try\_number function with 0,0 as a arguments which is (0,0) in the board in terms of coordinates where try\_number is a recursive function where it starts filling the board by trying to find the appropriate number to put into the board and then continue if it is an acceptable value and backtrack if necessary. | (public void solve()  {  if (try\_number(0, 0))  {  printBoard();  }  else  {  System.out.print("No solution can be found.");  }  } |
| The try\_number function as stated before is a recursive function that goes through each board position one by one from left to right at each row where the arguments are row and column which are the coordinates of the position, we are trying to find the appropriate number for. Each time, we check the base case which is row = 9 so we have finished the board. If not, we check if the row has ended, if so we move on to the next row. Otherwise, we try every number from 1 to 9 in the corresponding placement and call the check function which checks if this value can be placed on the board or not. It runs through all values of 1-9 until it finds the correct value, places it in the spot, and moves on to the next position. If required, the recursive function will move backward to a previous position to backtrack if no value can be found at that position that will satisfy the conditions. | public boolean try\_number(int row, int column)  {  if (row == 9)  {  return true;  }  if (column == 9)  {  return try\_number(row + 1, 0);  }  if (this.board[row][column] != 0)  {  return try\_number(row, column+1);  }  for (int i = 1; i <= 9; i++)  {  if (this.board[row][column] == 0)  {  if (check(row, column, i))  {  this.board[row][column] = i;  if (try\_number(row, column+1)) return try\_number(row, column+1);  this.board[row][column] = 0;  }  }  }  return false;  } |
| The check function takes the number we are attempting to put in a certain position and the coordinates of the position. It first checks the number isn’t in the row or column of the coordinates. It then finds out the coordinates of the position relative to its 3x3 grid rather than the 9x9 grid position and checks that in its 3 x 3 square, it is the only number in that square. | public boolean check(int row, int column, int num) {  for (int i = 0; i < 9; i++)  {  if (this.board[row][i] == num || this.board[i][column] == num)  {  return false;  }  }  int boxStartRow = row - row % 3;  int boxStartColumn = column - column % 3;  for (int i = 0; i < 3; i++)  {  for (int j = 0; j < 3; j++)  {  if (this.board[boxStartRow + i][boxStartColumn + j] == num)  {  return false;  }  }  }  return true;  } |
| This function prints the board and is called at the end to show the answer | public void printBoard()  {  for (int i = 0; i < 9; i++)  {  if (i % 3 == 0)  {  System.out.println("++---+---+---++---+---+---++---+---+---++");  System.out.println("++---+---+---++---+---+---++---+---+---++");  }  System.out.print("|| ");  for (int j = 0; j < 9; j++)  {  System.out.print(this.board[i][j] + " ");  if (j % 3 == 2) System.out.print("|| ");  else System.out.print("| ");  }  System.out.println();  }  System.out.println("++---+---+---++---+---+---++---+---+---++");  System.out.println("++---+---+---++---+---+---++---+---+---++");  } |
| This example board can be used as the starting board for the sudoku function to solve. | int[][] board = {  {5, 3, 0, 0, 7, 0, 0, 0, 0},  {6, 0, 0, 1, 9, 5, 0, 0, 0},  {0, 9, 8, 0, 0, 0, 0, 6, 0},  {8, 0, 0, 0, 6, 0, 0, 0, 3},  {4, 0, 0, 8, 0, 3, 0, 0, 1},  {7, 0, 0, 0, 2, 0, 0, 0, 6},  {0, 6, 0, 0, 0, 0, 2, 8, 0},  {0, 0, 0, 4, 1, 9, 0, 0, 5},  {0, 0, 0, 0, 8, 0, 0, 7, 9}  }; |

## Results:

Input:

int[][] board = {

{5, 3, 0, 0, 7, 0, 0, 0, 0},

{6, 0, 0, 1, 9, 5, 0, 0, 0},

{0, 9, 8, 0, 0, 0, 0, 6, 0},

{8, 0, 0, 0, 6, 0, 0, 0, 3},

{4, 0, 0, 8, 0, 3, 0, 0, 1},

{7, 0, 0, 0, 2, 0, 0, 0, 6},

{0, 6, 0, 0, 0, 0, 2, 8, 0},

{0, 0, 0, 4, 1, 9, 0, 0, 5},

{0, 0, 0, 0, 8, 0, 0, 7, 9}

};

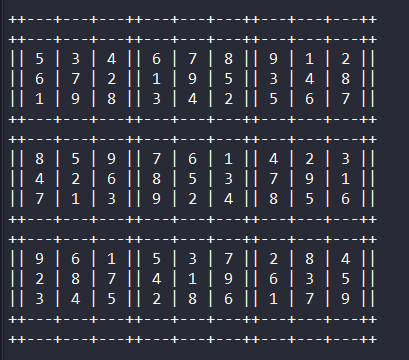


Figure 1: Java Output

## Flowchart:

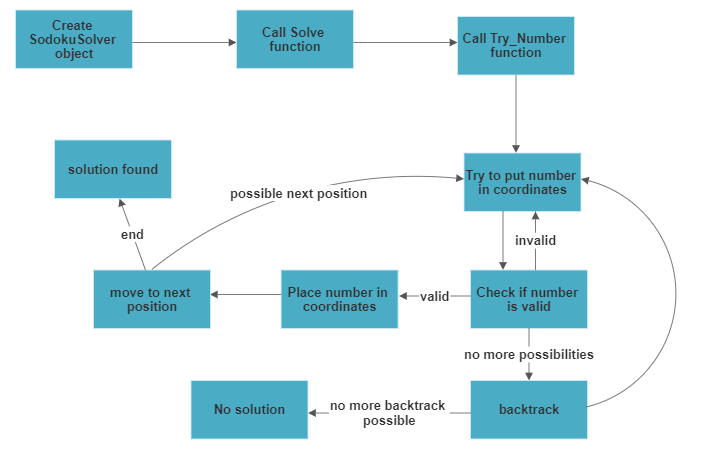


Figure 2: Flowchart of Java Code

# Sudoku Solver in Lisp

|  |  |
| --- | --- |
| The sudoku function is the main function and takes no parameters. It simply must only be called. Once it is called, it calls the try function with 0 as a parameter which means the 0th placement in the board which is (0,0) in the board in terms of coordinates which is a recursive function where it starts filling the board by trying to find the appropriate number to put into the board and then continue if it is an acceptable value. | (defun sodoku () (if (try 0) (show) (format t "This sodoku cannot be solved"))) |
| The try function as stated before is a recursive function that goes through each board position one by one from left to right at each row. The position is the parameter passed which is 9 x row + column (similar to how we do addressing or n-D arrays in a 1-D array). Each time, we take the position value and extract the row number by doing position / 9 with integer division, and the remainder which is position mod 9 is the column. Then we check that row and column are <= 9 so we know it is a valid coordinate on the board to avoid going to a coordinate that is not on the board. If it is not valid, we move along to the next iteration. Otherwise, we try every number from 1 to 9 in the corresponding placement and call the check function which checks if this value can be placed on the board or not. It runs through all values of 1-9 until it finds the correct value, places it in the spot, and moves on to the next position. | (defun try (position)  (let ((row (truncate position 9))  (column (mod position 9)))  (cond  ((or (>= row 9) (>= column 9)) t)  ((plusp (aref board row column)) (try (+ position 1)))  (t  (dotimes (i 9 (reset row column))  (if (check (+ i 1) row column)  (progn  (setf (aref board row column) (+ i 1))  (if (try (+ position 1)) (return t)))))))))) |
| The check function takes the number we are attempting to put in a certain position and the coordinates of the position. It then finds out the coordinates of the position relative to its 3x3 grid rather than the 9x9 grid position. This is stored in r and c. r is found by doing integer division on a row by 3 and then multiplying it by 3. c is found by doing integer division column by 3 and then multiplying it by 3. It then checks if the value is valid doing a series of calculations to compare the number with other numbers in its row, column and 3x3 box. If this number is not valid in that position, it returns nil, otherwise, it returns true. | (defun check (number row column)  (let ((r (\* (truncate row 3) 3))  (c (\* (truncate column 3) 3)))  (dotimes (i 9 t)  (when (or (= number (aref board row i))  (= number (aref board i column))  (= number (aref board (+ r (mod i 3))  (+ c (truncate i 3)))))  (return nil))))) |
| This function reset resets the position to empty/0 if a failure occurs. | (defun reset (row column)  (setf (aref board row column) 0)  nil) |
| This show function adds a visualization to the board and is called to show the answer at the end after the board is solved. | (defun show ()  (dotimes (row 9)  (if (zerop (mod row 3))  (format t "~%++---+---+---++---+---+---++---+---+---++~%++---+---+---++---+---+---++---+---+---++~%||")  (format t "~%++---+---+---++---+---+---++---+---+---++~%||"))  (dotimes (column 9)  (if (= 2 (mod column 3))  (format t " ~a ||" (aref board row column))  (format t " ~a |" (aref board row column)))))  (format t "~%++---+---+---++---+---+---++---+---+---++~%++---+---+---++---+---+---++---+---+---++~%")) |
| This example board can be used as the starting board for the sudoku function to solve. | (defvar board  #2a((5 3 0 0 7 0 0 0 0)  (6 0 0 1 9 5 0 0 0)  (0 9 8 0 0 0 0 6 0)  (8 0 0 0 6 0 0 0 3)  (4 0 0 8 0 3 0 0 1)  (7 0 0 0 2 0 0 0 6)  (0 6 0 0 0 0 2 8 0)  (0 0 0 4 1 9 0 0 5)  (0 0 0 0 8 0 0 7 9))) |

## Results:

Input:

(defvar board

#2a((5 3 0 0 7 0 0 0 0)

(6 0 0 1 9 5 0 0 0)

(0 9 8 0 0 0 0 6 0)

(8 0 0 0 6 0 0 0 3)

(4 0 0 8 0 3 0 0 1)

(7 0 0 0 2 0 0 0 6)

(0 6 0 0 0 0 2 8 0)

(0 0 0 4 1 9 0 0 5)

(0 0 0 0 8 0 0 7 9)))

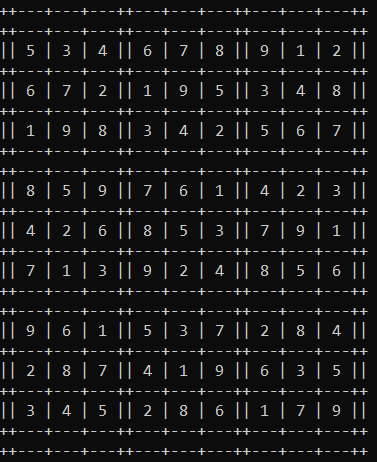


Figure 3: Lisp Output

## Flowchart:

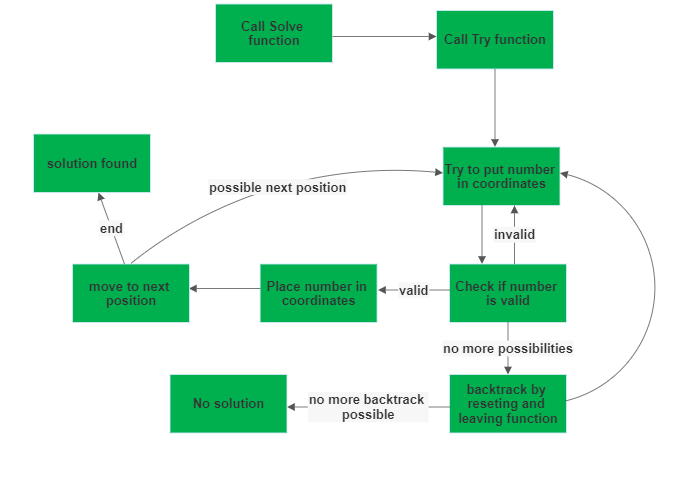


Figure 4: Flowchart of Lisp Code

# Sudoku Solver in Prolog

Below are all the functions/predicates used to implement the solver. Note that the function parameters have been exchanged with “…” or a shortened version for simplicity and to improve readability.

|  |  |
| --- | --- |
| The sudoku predicate is the main function of this solver. It takes in 81 variables which are passed in when the function is called. It then prints out “Solving…” to indicate that it is beginning the process of solving the sudoku and then it prints the input problem with the printProblem function and passed all the 81 variables to it.  It then prints out “Solution:” as it will begin calculating the solution with the solve function that will also take the 81 variables as parameters. The solve function takes care of the main computation, and afterward, it prints the solution with the printSolution function.  Each clause is only evaluated if the previous clause is true. Thus, by ANDing each instruction, we can create a sort of imperative code where we can sequentially call each predicate. This is how each predicate can act as a function. This logic is used in every function below. | sudoku(R1C1, R1C2, ..., R9C9) :-  write('Solving...'), nl,  printProblem(R1C1, R1C2, ..., R9C9),  write('Solution:'), nl,  solve(R1C1, R1C2, ..., R9C9),  printSolution(R1C1, R1C2, ..., R9C9). |
| The solve function is where the main computation occurs. For every row, column, and 3x3 square it calls the isValid function, passing in the 9 relevant variables. If any isValid fails and returns false, Prolog will backtrack to the most recent decision point.  isValid will keep testing possible values and will continue until a solution is found or all possible values have been tried.  Again, the clauses will be evaluated in order and thus each check of the rows or columns, or grids occurs in the order as it is defined. | solve(R1C1, R1C2, ..., R9C9) :-  isValid(...), % First row  isValid(...), % Second row  isValid(...), % Third row  isValid(...), % Fourth row  isValid(...), % Fifth row  isValid(...), % Sixth row  isValid(...), % Seventh row  isValid(...), % Eighth row  isValid(...), % Ninth row  isValid(...), % First column  isValid(...), % Second column  isValid(...), % Third column  isValid(...), % Fourth column  isValid(...), % Fifth column  isValid(...), % Sixth column  isValid(...), % Seventh column  isValid(...), % Eighth column  isValid(...), % Ninth column  isValid(...), % Top Left block  isValid(...), % Top Middle block  isValid(...), % Top Right block  isValid(...), % Left block  isValid(...), % Middle block  isValid(...), % Right block  isValid(...), % Bottom Left block  isValid(...), % Bottom Middle block  isValid(...). % Bottom Right block |
| The isValid function takes the 9 variables from either a row, column or 3x3 grid being passed and checks whether the variables are not reoccurring and are distinct.  This process first checks if every variable is instantiated and makes sure it is a number in the universe of discourse which is [1-9].  If the variable is uninstantiated in other words, unknown, it binds it to a value in the universe of discourse in the order of which the universe of discourse was defined. In the case where the isValid fails, solve will backtrack and Prolog will test the next value in the universe of discourse for the uninstantiated value.  Once it confirms all variables are proper and instantiated, it then checks each variable with every other variable to see if they are distinct. This process is called Constraint Satisfaction. | isValid(A, B, C, D, E, F, G, H, I) :- num(A), num(B), num(C), num(D), num(E), num(F), num(G), num(H), num(I),  A\=B, A\=C, A\=D, A\=E, A\=F, A\=G, A\=H, A\=I,  B\=C, B\=D, B\=E, B\=F, B\=G, B\=H, B\=I,  C\=D, C\=E, C\=F, C\=G, C\=H, C\=I,  D\=E, D\=F, D\=G, D\=H, D\=I,  E\=F, E\=G, E\=H, E\=I,  F\=G, F\=H, F\=I,  G\=H, G\=I,  H\=I. |
| This function is used to print the final solved grid by passing each row to the print function.  The print function then adds some borders and spacing for readability. | printSolution(R1C1, R1C2, ..., R9C9):-  print(R1C1, R1C2, ..., R1C9),  print(R2C1, R2C2, ..., R2C9),  print(R3C1, R3C2, ..., R3C9),  print(R4C1, R4C2, ..., R4C9),  print(R5C1, R5C2, ..., R5C9),  print(R6C1, R6C2, ..., R6C9),  print(R7C1, R7C2, ..., R7C9),  print(R8C1, R8C2, ..., R8C9),  print(R9C1, R9C2, ..., R9C9).  print(A, B, C, D, E, F, G, H, I) :-  write(' | '), write(A), write(' '), write(B), write(' '), write(C),  write(' | '), write(D), write(' '), write(E), write(' '), write(F),  write(' | '), write(G), write(' '), write(H), write(' '), write(I), write(' | '), nl. |
| This function is used to print the problem grid. Each row is passed to the printInit helper function.  The printInit helper function then adds some borders and spacing for readability, it also uses another helper function, printValueHelper to print ‘x’ when the variable is uninstantiated. | printProblem(R1C1, R1C2, ..., R9C9):-  printInit(R1C1, R1C2, ..., R1C9),  printInit(R2C1, R2C2, ..., R2C9),  printInit(R3C1, R3C2, ..., R3C9),  printInit(R4C1, R4C2, ..., R4C9),  printInit(R5C1, R5C2, ..., R5C9),  printInit(R6C1, R6C2, ..., R6C9),  printInit(R7C1, R7C2, ..., R7C9),  printInit(R8C1, R8C2, ..., R8C9),  printInit(R9C1, R9C2, ..., R9C9).  printInit(A, B, C, D, E, F, G, H, I) :-  write(' | '), printValueHelper(A), write(' '), printValueHelper(B), write(' '), printValueHelper(C),  write(' | '), printValueHelper(D), write(' '), printValueHelper(E), write(' '), printValueHelper(F),  write(' | '), printValueHelper(G), write(' '), printValueHelper(H), write(' '), printValueHelper(I), write(' | '), nl.  printValueHelper(X) :-  (var(X) -> write('x') ; write(X)). |
| This section of the code defines the universe of discourse. It creates facts for the interval [1-9]. | num(1). num(2). num(3). num(4). num(5). num(6). num(7). num(8). num(9). |

## Results

Input:

sudoku(1, \_, \_, 4, 8, 9, 3, 7, 6, 7, 3, 9, 2, 5, 6, 8, \_, 1, 4, \_, 8, 3, 7, 1, 2, 9, \_, 3, 8, 7, 1, 2, 4, \_, 5, 9, 5, 9, 1, 7, 6, 3, 4, 2, 8, \_, \_, 6, \_, 9, 5, 7, \_, \_, 9, 1, 4, 6, \_, 7, \_, \_, 2, \_, 2, \_, \_, 4, 8, \_, 3, 7, 8, \_, \_, 5, 1, 2, \_, 6, 4).

A picture containing text, font, black and white, screenshot

Description automatically generated

Figure 5: Prolog Output

## Flowchart

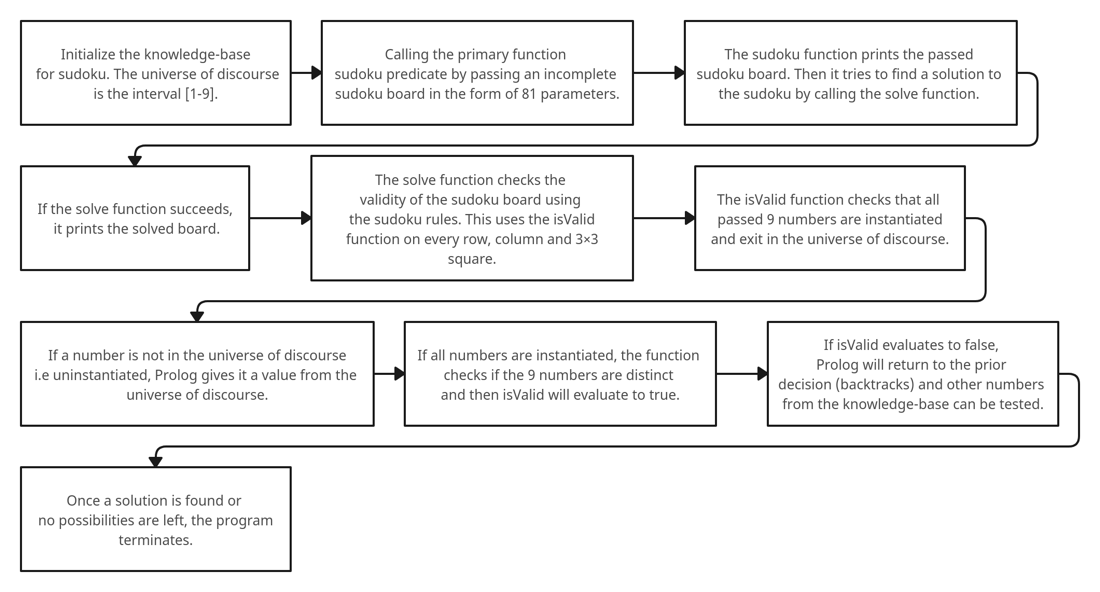


Figure 6: Flowchart of Prolog Code

# Comparison

Imperative, functional, and logic programming are the three most used programming paradigms in software development. Each programming paradigm has its own unique set of features and functionalities which makes them ideal for specific use cases. Moreover, each paradigm has its own strengths and weaknesses.

## Imperative Programming:

Imperative programming is the most widely used programming paradigm. The ideology of imperative programming believes that a program should be a sequence of instructions that are executed in a specific order. The programmer specifies how the program should be executed. Common examples of imperative programming languages include Python, C, C++, and Java.

**Strengths:**

1. Easy to learn: Imperative programming languages are easy to learn and understand.
2. Good for low-level programming: Imperative programming is ideal for low-level programming tasks where the programmer needs more control over the hardware.
3. Easy to debug: Imperative programming makes it easier to debug code since the program's state is clear at each step.
4. Computer architectures are designed using Von Neuman Architecture which are styled for imperative languages.

**Weaknesses:**

1. Difficult to parallelize: Imperative programming can be difficult to parallelize, which can limit performance in some situations.
2. Prone to errors: Imperative programming languages can be prone to errors, especially when dealing with large and complex code such as side effects with global variables.

## Functional Programming:

Functional programming is a programming paradigm that treats computation as mathematical functions such as Church’s lambda calculus. In functional programming, functions are the primary building blocks of the program, and they are designed to be pure, meaning that they do not have any side effects or variables. Examples of functional programming languages include Haskell, F#, Lisp, and ML.

**Strengths:**

1. Easy to reason with: Functional programming makes it easy to reason with code since functions are pure and do not have any side effects.
2. Easy to parallelize: Functional programming is easy to parallelize, which can lead to better performance in some situations.
3. Fewer bugs: Functional programming languages are less prone to bugs since they avoid side effects and mutable state.

**Weaknesses:**

1. Steep learning curve: Functional programming can be difficult to learn for programmers who are used to imperative programming.
2. Limited tooling: Functional programming languages often have limited tooling and fewer libraries than imperative programming languages.

## Logic Programming:

Logic programming is a programming paradigm that uses first-order predicate calculus to solve problems. In logic programming, the program is a set of logical statements that describe the problem called facts, and the program's execution involves finding a solution that satisfies these statements. It describes the form of a result and not how the result should be computed. Some examples of logic programming languages include Prolog and Mercury.

**Strengths:**

1. Good for solving complex problems: Logic programming is good for solving complex problems that are difficult to solve with imperative or functional programming.
2. Easy to reason with: Logic programming makes it easy to reason with code since programs are sets of logical statements. The program only knows about the facts and only has the knowledge base that the programmer adds.
3. Can be used for AI and machine learning: Logic programming is commonly used in artificial intelligence and machine learning applications such as expert systems. Some common issues with AI are illusions, where the AI learns incorrect information and treats it as true facts. For example, how ChatGPT says factually incorrect information with full confidence of what it says being correct. Logic programming can prevent these illusions.

**Weaknesses:**

1. Limited tooling: Logic programming languages often have limited tooling and fewer libraries than imperative or functional programming languages.
2. Limited to specific use cases: Logic programming is limited to specific use cases, such as artificial intelligence and machine learning applications.
3. Steep learning curve: Logic programming can be difficult to learn for programmers who are used to imperative programming.

# Conclusion

In conclusion, each programming paradigm has its own strengths and weaknesses. Imperative programming is widely used, easy to learn, and good for low-level and general-purpose programming tasks. Functional programming is easy to reason with, easy to parallelize, and less prone to bugs. Logic programming is good for solving complex problems and is commonly used in AI and machine learning applications such as expert systems. When choosing a programming paradigm for a project, it is essential to consider the specific requirements and constraints and the exact use case of the project to determine which paradigm to use.