Pushdown Automata (PDA) – Projects 1 to 10

1. PDA Simulator (CLI-based)

Objective: To simulate the behavior of a Pushdown Automaton using command-line interface.

The user inputs a PDA description (states, transitions, stack symbols).

The simulator reads an input string and traces PDA transitions.

It shows the stack operations in each step (push/pop).

Highlights final acceptance/rejection of input.

Useful for students to understand how PDAs process strings.

Covers both deterministic and simple non-deterministic PDAs.

Output is shown in the form of transition logs.

Code is modular to support custom PDA inputs.

Can be extended to accept PDA configuration via file.

2. PDA for Balanced Parentheses

Objective: Build a PDA that accepts strings with correctly nested brackets.

It uses stack operations to ensure each opening bracket is closed.

Every '(' or '[' is pushed onto the stack.

For each closing bracket, the top of the stack is popped and matched.

The PDA reaches final state if the input is consumed and the stack is empty.

Helps understand the use of stack in parsing nested structures.

Can be tested on strings like (([])), ([])[], etc.

Rejects unbalanced or mismatched expressions.

Can be extended to support multiple types of delimiters.

Includes graphical trace of stack operations.

3. PDA for Palindromes (with separator)

Objective: Create a PDA that accepts palindromes in the format $w\#w^R$.

It pushes the first half w of the string onto the stack.

When it reads the # separator, it changes mode.

Then for each symbol in the second half, it pops and compares with stack top.

If all match and stack becomes empty at end, accept.

Demonstrates the power of stack memory for mirroring input.

Use cases in DNA palindrome detection and code analyzers.

Works with both even and odd-length input.

Non-deterministic behavior can be visualized in trace.

Can be extended to handle more general palindrome forms.

4. Interactive Stack Visualizer

Objective: A GUI tool that shows how stack changes during PDA execution.

Users input a PDA configuration and a string.

As the PDA processes input, the tool animates stack operations.

Each transition is shown with current state, input symbol, and stack symbol.

Great learning tool for visualizing push/pop operations.

Highlights active transitions and non-deterministic choices.

Includes pause/play and step-by-step modes.

Can show multiple stack traces in parallel if PDA is non-deterministic.

Students can practice debugging PDA transitions visually.

Built using Python tkinter or JavaScript for portability.

5. PDA to CFG Converter

Objective: Automatically convert a PDA into its equivalent Context-Free Grammar.

Inputs a PDA description in standard format.

Constructs variables and production rules based on transitions.

Uses standard conversion approach (A_{pq} → rules).

Outputs a CFG that accepts the same language as the PDA.

Useful for verifying equivalence of PDA and CFG.

Supports deterministic and simple non-deterministic PDAs.

Outputs grammar in readable form (S → aSb | ε).

May include visualization of grammar derivation tree.

Helpful for compiler design and theoretical verification.

6. Nondeterministic PDA Emulator

Objective: Emulate a non-deterministic PDA and explore multiple computation paths.

Processes an input string and shows all possible transition branches.

Implements depth-first or breadth-first search on configurations.

Rejects if no branch leads to acceptance; accepts if any does.

Visualizes branching structure as a tree of configurations.

Helps students understand power and complexity of nondeterminism.

Can limit recursion depth to avoid infinite loops.

Supports epsilon transitions and stack branching.

Includes trace viewer and state debugger.

Logs decision paths taken during execution.

7. PDA for $a^nb^nc^n$

Objective: Accept the language $a^nb^nc^n$ using a PDA, which requires a complex strategy.

Traditional PDA with one stack cannot accept this, so uses tricks.

Implements a nondeterministic PDA that guesses mid-point.

Pushes 'a', matches 'b' with pops, then 'c' with count.

Another approach marks transitions and uses multi-pass parsing.

Illustrates the limitation of PDA and power of nondeterminism.

Used in theoretical analysis and complex language modeling.

Includes transition logs with input pointers and stack state.

Rejects incorrect orders like aaabbbccc with unmatched symbols.

Built using Python or Java, extensible to simulate TMs.

8. PDA with Stack Alphabet Editor

Objective: Allow users to define custom stack alphabets and PDA rules.

Interface to input stack symbols, states, transitions.

Supports dynamic PDA generation based on user input.

Stack visualization updates as input is parsed.

User can run custom strings and test stack behaviors.

Helps learners build custom PDAs for any language.

Includes default templates (balanced brackets, a^nb^n, etc.).

Can be used in competitions or classrooms for design tasks.

Error-checks invalid configurations or rules.

Logs each input and its trace in structured format.

9. Game-based PDA: Stack Battle

Objective: Build a gamified learning platform for PDA simulation.

Players choose actions like push, pop, transition based on rules.

Each level has a target string that must be accepted.

Visual stack grows/shrinks as players make moves.

Wrong moves cause stack errors or rejection.

Levels increase in complexity (ε-transitions, branching).

Leaderboard and score system for competitive learning.

Built using React or Unity for interactive UI.

Can integrate PDA puzzles from real course content.

Engages students in automata theory through game logic.

10. PDA Input Expression Validator

Objective: Use PDA to validate arithmetic expressions with nested parentheses.

Accepts expressions like ((a+b)c) or a(b+c)/(d-e).

PDA pushes '(' onto stack and pops for ')'.

Also handles operator precedence with additional stack rules.

Rejects malformed expressions (e.g., unbalanced parentheses).

Can extend to check syntactical correctness (variable names, operators).

Used in compiler frontend tools for syntax validation.

Shows expression as tree or evaluation trace.

Real-world application: parsing code or mathematical input.

Includes error messages and diagnostic output.

Context-Free Grammar (CFG) – Projects 11 to 20

11. CFG Parser for Arithmetic Expressions

Objective: Build a CFG-based parser for arithmetic expressions with +, -, , /.

Define grammar rules like E → E+E | EE | (E) | id.

Use recursive descent or LL(1) parsing to implement the parser.

Accepts inputs like (a+b)*c and builds parse trees.

Validates structure and syntax of the input.

Rejects invalid inputs (e.g., unmatched parentheses).

Can include variable support (like x, y) and integer literals.

Outputs derivation steps or parse tree structure.

Helpful in understanding compiler parsing stages.

Ideal for compiler or syntax analysis projects.

12. CFG Visual Grammar Tree Builder

Objective: Visualize the derivation tree for any CFG and input string.

Input includes CFG rules and a string to parse.

Builds a parse tree dynamically as the grammar generates the string.

Highlights derivation steps with node expansions.

Helps visualize recursive productions and ambiguity.

Supports leftmost and rightmost derivations.

Useful for teaching parsing techniques and ambiguity detection.

Can export tree structure as image or JSON.

Built using JS libraries like D3.js or Python's Graphviz.

Allows users to test their own grammars interactively.

13. Grammar Ambiguity Detector

Objective: Analyze CFGs for ambiguity by comparing multiple derivation trees.

Users enter CFG and input string.

Tool generates all possible derivation trees for the string.

If more than one tree exists, grammar is ambiguous.

Shows both leftmost and rightmost derivations.

Highlights conflicting paths in parse trees.

Useful for debugging grammars in compilers and parsers.

Handles simple ambiguous grammars like E → E+E | E*E | id.

Can suggest disambiguation using precedence rules.

Output includes grammar classification and resolution tips.

14. Grammar to PDA Converter

Objective: Convert a CFG to its equivalent PDA representation.

Takes grammar in Chomsky Normal Form or Greibach Normal Form.

Applies standard rules to generate transitions for PDA.

Displays the resulting PDA transitions and stack operations.

Simulates PDA for string inputs from the grammar.

Helps in understanding CFG-PDA equivalence.

Useful for mapping theory to PDA behavior.

Includes animation of stack processing for grammar strings.

Offers both automatic and manual CFG-to-PDA translation.

Supports exporting PDA as JSON or graph.

15. CFG Language Generator

Objective: Generate a list of valid strings from a given CFG.

User provides production rules and terminal symbols.

Tool generates strings up to specified length.

Shows derivation steps for each string.

Can use recursive traversal with backtracking to avoid infinite loops.

Useful for testing grammars and language coverage.

Can identify unproductive or unreachable rules.

Supports bounded generation (e.g., max depth or length).

Outputs in batch format with optional derivation trees.

Great for verifying CFG design in course projects.

16. LL(1) Parser Generator

Objective: Build a tool that constructs LL(1) parse tables from CFG.

Calculates FIRST and FOLLOW sets for each non-terminal.

Generates parsing table and validates LL(1) property.

Rejects ambiguous or left-recursive grammars.

Includes step-by-step table construction interface.

Parses input strings using the generated table.

Detects conflicts and suggests grammar transformations.

Visual trace of parsing using the table.

Supports input in standard BNF notation.

Helpful in building foundational parsers for small languages.

17. CFG Simplifier and Normalizer

Objective: Simplify and convert any CFG into CNF (Chomsky Normal Form).

Removes unreachable symbols, ε-productions, and unit rules.

Applies steps systematically and shows intermediate grammars.

Final output is equivalent grammar in CNF.

Used in algorithms like CYK and CFG-to-PDA conversion.

Includes rules and logs for transparency.

Handles both small and complex grammars.

Great for learning grammar optimization.

Supports batch CFG simplification for testing.

Built with support for JSON or plain-text input.

18. CFG Quiz Generator Tool

Objective: Build a quiz app that generates CFG-based questions.

Includes MCQs and derivation challenges from a given CFG.

Users solve problems like identifying valid strings, derivation steps.

Checks grammar knowledge and parsing understanding.

Can randomize production rules or string lengths.

Includes hints and solution explanation.

Useful for automated practice in courses or labs.

Supports difficulty levels and progress tracking.

Admin can upload grammars for dynamic quiz creation.

Gamified design with points and feedback.

19. CFG for Simple Programming Language Grammar

Objective: Define a CFG for a subset of a programming language.

Supports expressions, conditionals, loops, variable declarations.

E.g., statements like if (x < y) then x = y;.

Grammar includes rule-based token structure and block nesting.

Can be parsed using LL(1) or recursive descent parser.

Visual parse trees validate syntactic correctness.

Prepares foundation for compiler or interpreter projects.

Modular grammar to add functions or arrays.

Includes lexer integration for token generation.

Can be extended to parse real code snippets.

20. CFG Derivation Game

Objective: Create an educational game where players derive strings from CFG.

Player selects rules to apply step-by-step.

Game checks if final string matches the target.

Time-based or level-based challenges.

Hints show possible derivation paths.

Visual derivation tree grows with each move.

Encourages students to think in grammar rules.

Includes examples from class (like palindromes or balanced strings).

Built using web interface with interactive buttons.

Supports teacher-uploaded custom grammars.

Turing Machines (TM) – Projects 21 to 30

21. Turing Machine for Binary Addition

Objective: Design a TM to perform binary addition of two numbers.

Input format: two binary strings separated by a delimiter (e.g., 101+011).

TM scans both numbers and simulates bit-by-bit addition with carry.

Performs write, move, and state transitions based on rules.

Final output replaces the input with the resulting binary sum.

Demonstrates the computational power of TM over FSM/PDA.

Includes tape snapshot at each computation step.

Used in modeling arithmetic computation logic.

Good for illustrating memory head and infinite tape concept.

Can be extended to support subtraction or multiplication.

22. Turing Machine to Check Palindromes

Objective: Create a TM that accepts palindromes over {a, b}.

Works by comparing leftmost and rightmost symbols.

Marks matched characters and moves inward recursively.

If unmatched, halts in a reject state.

Accepts strings like abba, aba, aa.

Rejects strings like abab, abcba.

Shows state transition logic and head movement.

Demonstrates symmetric pattern recognition in TM.

Tape visualization helps track symbol comparison.

Useful for explaining non-regular language handling.

23. TM for Unary Multiplication

Objective: Build a TM that multiplies two unary numbers (e.g., 111*11).

Uses marker symbols to simulate multiplication as repeated addition.

Rewrites tape to produce the final unary output (e.g., 111111111).

Handles input parsing, loop transitions, and cleanup.

Great for demonstrating algorithm design on TM.

Provides clear example of looping and recursion in TM.

Showcases stack-like and tape-based memory handling.

Includes intermediate tape contents as part of output.

Tests student understanding of nested transitions.

Scalable to more complex arithmetic logic.

24. Turing Machine Simulator GUI

Objective: Build a tool to simulate and visualize Turing Machines.

User enters states, transitions, input tape.

Simulates step-by-step execution with head movement.

Displays current state, tape snapshot, and transition logs.

Supports pausing, editing, and restarting.

Highlights current symbol and transition rule applied.

Can preload common TM problems (palindrome, addition).

Built using Python/Tkinter or JS/React.

Improves learning by visual interaction.

Great classroom demo tool.

25. TM for String Reversal

Objective: Design a TM that reverses a string over {a, b}.

Reads and pushes characters to the right end.

Swaps characters from ends iteratively.

Marks processed symbols to avoid reprocessing.

Tape ends with reversed version of input.

Shows how TMs simulate memory stacks and queues.

Includes real-time tape visualization.

Good for teaching intermediate TM concepts.

Can be extended to include more alphabets.

Encourages step-based debugging and transition design.

26. TM for Even Number of 1s

Objective: Accept binary strings with an even number of 1s.

Moves across tape, toggling state on each 1.

Accepts if ends in even state, rejects otherwise.

Handles large strings efficiently with finite transitions.

Visualizes parity checking using TM logic.

Illustrates deterministic transition behavior.

Useful for basic TM construction practice.

Also applicable to FSM vs TM comparisons.

Can be shown alongside equivalent FSM for contrast.

Simple, yet effective problem for starters.

27. Multi-tape TM Simulator

Objective: Build a simulator for multi-tape Turing Machines.

Supports two or more tapes for complex computations.

User defines transitions for each tape head.

Simulates sorting, copying, or arithmetic operations.

Shows benefit of multitape over single tape models.

Visualizes head movement per tape and their state.

Helps explore computational speedup with multitape.

Excellent for advanced TM theory practice.

Uses file I/O for tape definitions.

Can preload tasks like string copy or mirror.

28. TM for Language L = { $a^n b^n c^n$ | n ≥ 1 }

Objective: Accepts strings where number of a's, b's and c's are equal and ordered.

Not regular or context-free, hence needs TM.

Marks a, then finds corresponding b and c.

Repeats for each set, marking used characters.

Halts if all matched correctly.

Rejects if order or count is wrong.

Demonstrates higher power of TMs over PDAs.

Supports tape printouts for debugging.

Ideal for advanced students.

Challenge: support extended inputs.

29. TM for Balanced Parentheses

Objective: Accept valid expressions with balanced (), {}, [].

Simulates stack behavior using the tape.

Marks matched opening and closing brackets.

Rejects on mismatch or unclosed brackets.

Multi-symbol support: '(', '{', '[' and corresponding closings.

Detects errors early in transition state.

Shows use of multiple transitions per input.

Highlights practical parsing application.

Helpful in compiler theory mapping.

Encourages handling multi-state parsing.

30. TM that Converts Unary to Binary

Objective: Read unary number and write equivalent binary.

Unary input: e.g., 1111 (represents 4).

Counts ones and updates a binary representation.

Uses tape as memory counter and output encoder.

Tests logical design in TM step transitions.

Good use of working symbols and rewrite logic.

Includes handling of tape boundaries.

Provides example of arithmetic-based transformation.

Excellent mix of math and TM logic.

Fun for coding TM from scratch.

Turing Machines and Hybrid Projects – Projects 31 to 40

31. TM for Anagram Checker

Objective: Create a Turing Machine that checks if two strings are anagrams.

Input: two strings separated by a special character (e.g., ab#ba).

Marks letters in the first string and searches for matches in the second.

Each letter is removed once matched to avoid repetition.

Rejects if mismatches or leftover letters exist.

Simulates counting and pattern tracking on the tape.

Demonstrates memory tracking and permutation logic.

Includes step logs for match and remove operations.

Can support case-insensitive matching.

Good for exploring TM design for non-trivial conditions.

32. TM for Prime Number Checker

Objective: Accept unary representations of prime numbers (e.g., 11111 for 5).

Divides unary string by numbers starting from 2 (simulated subtraction).

Rejects if divisible, accepts if not.

Involves modular arithmetic logic on the TM tape.

Demonstrates complex logic using a single tape.

Good project for mathematical modeling using TM.

Includes output of YES/NO with trace.

Can log each divisor tested.

Challenging and suitable for higher UG levels.

Tests understanding of loops and control.

33. Universal Turing Machine Simulator

Objective: Simulate a Universal TM that reads another TM as input.

Reads a TM's description and a string to simulate its execution.

Simulates tape, transitions, and output generation.

Demonstrates the principle that one TM can simulate another.

Key project for understanding computability theory.

Useful in advanced automata and compiler theory.

Good candidate for GUI or CLI-based simulator.

Supports transition log and step mode.

Foundation for real-world interpreters.

Ambitious but achievable in 4 weeks.

34. Hybrid CFG-TM Parser

Objective: Use CFG for parsing and TM for post-validation.

CFG parses syntax structure, TM validates semantic patterns.

Useful for inputs like nested HTML/XML or code patterns.

Two-stage design: CFG parser followed by TM.

Demonstrates layered language processing.

Good mix of theory and programming.

Implements pipeline-style processing.

Supports error detection with specific messages.

Visualization for both parser and tape logic.

Encourages modular automata thinking.

35. TM-based Calculator for Postfix Expressions

Objective: Evaluate postfix expressions like 23+5*.

TM uses symbols and stack emulation to perform operations.

Handles digits and operators with tape-based memory.

Pushes operands, performs operations on pop.

Final result is displayed at end of tape.

Great example of computation using TM model.

Simulates operand stack and instruction flow.

Tape logs show state at each step.

Includes error handling (invalid syntax).

Helps link Automata with real computation.

36. TM Password Validator

Objective: Accept passwords that meet given rules (e.g., 1 digit, 1 capital, ≥8 chars).

Checks input from left to right and marks symbols.

Transitions through states representing rules met.

Rejects if any rule fails at the end of input.

Demonstrates conditional logic in TM.

Great for real-world validation modeling.

Useful for showing composite state design.

Encourages encoding rules in state diagrams.

Can include detailed error states.

Supports password strength report.

37. TM for Binary to Decimal Conversion

Objective: Convert binary strings (e.g., 1011) to decimal form (11).

Reads binary bits and computes their value using powers of two.

Stores partial results on tape and updates them as it reads bits.

Outputs the decimal result in base-10 digits.

Tape shows intermediary math and final result.

Challenging due to base conversion handling.

Requires clever encoding of arithmetic on TM.

Useful for demonstrating TM's numerical processing power.

Tests encoding schemes and data reuse.

Fun and rewarding for math-inclined students.

38. TM-based Palindrome Transformer

Objective: Given any string, output the nearest palindrome using TM logic.

If not a palindrome, modify the tape to convert into one.

Find mismatched pairs and replace to balance.

Demonstrates error correction logic.

Uses scanning, rewriting, and checking.

Can count number of edits required.

Includes final tape state showing output.

Visualizes transition strategy and matching.

Useful for pattern recognition studies.

Extends basic palindrome check to editing.

39. Self-Modifying TM

Objective: Build a TM that changes its own transition rules based on input.

Implements a meta-TM that rewrites its transition set on tape.

Simulates logic like conditionals and loops with modified behavior.

Highly advanced – shows potential of programmable computation.

Useful to explore ideas of self-adaptive systems.

Tape structure includes a transition table area.

Logs each modification to TM configuration.

Advanced UG or early graduate-level challenge.

Connects to modern reconfigurable computing.

Can include sandboxed transitions for testing.

40. TM for Recognizing Language L = { ww | w ∈ {a,b} }*

Objective: Accept strings that are two copies of the same word.

E.g., abab, aa, baba, but reject aabb, abbb.

Compares the first half to the second half.

Marks and checks symbols recursively from both ends.

Works only for even-length strings.

Requires careful pointer and state logic.

Excellent for symmetry recognition in TM.

Shows complexity beyond context-free languages.

Includes trace logs for every pair matched.

Tests student's ability to design complex logic.

Advanced/Creative Automata Projects

41. PDA for Palindromes with Middle Marker

Objective: Accept strings like abba, abcba, using a stack and a middle marker #.

Pushes characters until # is encountered.

Then pops the stack while reading remaining input.

Matches input symmetry around the middle.

Rejects if unmatched symbols or uneven structure.

Classic PDA application demonstrating stack power.

Includes PDA transition table and simulation trace.

Helpful for learning two-phase stack operations.

Enhances understanding of stack-based memory.

Visual diagrams show parsing symmetry.

42. PDA for Balanced HTML Tags

Objective: Accept HTML-like strings with properly nested tags.

Tags: <tag></tag>, e.g., <html><body></body></html>.

Pushes opening tags to the stack and pops on closing.

Checks for match and order during popping.

Rejects on mismatch or unclosed tag.

Replicates parsing process in browsers.

Can be expanded to support attributes.

Good exercise in real-world parsing logic.

Strong PDA application in markup processing.

Ideal for combining theory with web structures.

43. CFG Generator from PDA Transitions

Objective: Write code to automatically convert a PDA to equivalent CFG.

Takes PDA states and transitions as input.

Generates corresponding CFG rules (e.g., A → aB).

Validates generated CFG against test strings.

Includes comparison of PDA vs CFG behavior.

Highlights equivalence between CFG and PDA.

Helps students visualize grammar construction.

Excellent theory-practical bridge project.

Supports learning automata transformations.

Great base for compiler parsing tools.

44. CFG-based Programming Language Parser

Objective: Define CFG for a small programming language (arithmetic + conditionals).

Grammar supports if, else, loops, expressions.

Builds parser that checks syntax using CFG rules.

Implements derivation steps and parse tree.

Highlights ambiguity and operator precedence.

Supports user-defined test scripts.

Tool for compiler and parser design introduction.

Enhances CFG intuition via real coding examples.

Supports extension with functions or nesting.

Useful for classroom and lab activities.

45. Turing Machine that Solves Sudoku (Simplified)

Objective: Simulate a TM that checks valid Sudoku rows/columns.

Simplifies 9x9 grid to linear format or reduced grid.

Validates rules: no repeating numbers in row/col.

Encodes grid on tape and checks using states.

Logs errors and marks invalid positions.

Explores limits of TM in practical logic checks.

Highlights symbol encoding and state complexity.

Challenges creative design with constraints.

Good exploratory final-year project.

Can be extended to full grid solution.

46. Hybrid PDA+TM Chat Validator

Objective: Design a system that uses PDA for syntax and TM for semantics in chat input.

Example: PDA checks grammar of sentences, TM checks for spam patterns.

Simulates two-level validation for real-world communication.

Tracks message structure and flags invalid content.

Combines two automata models in one pipeline.

Perfect blend of theory and practical NLP filtering.

Uses rule-based detection and memory tracking.

Supports visualization of each stage separately.

High creativity + applied value.

Ideal for UG mini-thesis or group projects.

47. CFG-Based Code Obfuscator

Objective: Use CFG rules to transform code syntax while preserving logic.

Takes source code and applies grammar-based rewrites.

Outputs semantically identical but syntactically altered code.

Demonstrates usage of CFG in reverse parsing.

Includes derivation logs for each transformation.

Can be used to study malware code rewriting.

Explores ambiguity in grammar representation.

Real-world link to compilers and obfuscation.

Scalable with plug-in grammar sets.

Great interdisciplinary research project.

48. PDA for Real-time Parenthesis Balancer in Code Editors

Objective: Use PDA to assist live coding environments in detecting syntax issues.

Tracks stack state as programmer types code.

Provides live feedback on unbalanced braces.

Implements using JS/C++ for editor plugin.

Visual stack diagram shows current parsing state.

Real-world IDE utility based on PDA.

Bridges theory with developer tools.

Good for automation and UI development.

Includes PDA transition debugger.

Excellent for theory + system design fusion.

49. Turing Machine that Simulates Encryption

Objective: Design a TM that encodes input string using a shift cipher (e.g., Caesar).

Reads character, shifts symbol based on fixed offset.

Writes encrypted character back to tape.

Includes tape logs and shift logic diagram.

Supports decrypt mode using reverse shifts.

Highlights symbol manipulation with deterministic TM.

Good for combining Automata and Security.

Advanced string encoding using state control.

Educational tool for intro to cryptographic logic.

Fun and practical TM application.

50. PDA/CFG Visualizer Tool for Students

Objective: Build a tool where students enter CFG or PDA and see real-time derivation.

Supports drawing state transitions and grammar derivation tree.

Allows manual or automatic step-by-step execution.

Highlights current stack and tape content.

Exports simulation results and parse trees.

Encourages experimentation with inputs and rules.

Very helpful for visual learners and teachers.

Built using Python (Tkinter), JavaScript, or JavaFX.

Top-tier tool for automata education.

Capstone-worthy undergraduate project.