Deep Science Publishing
https://doi.org/10.70593/978-93-49910-92-8_6

Chapter 6: A Study on push-down automata

Soumodip Kumar Paul, Babul Sarkar

Abstract: This chapter explores the concept, structure and applications of Push-Down Automata (PDA), a theoretical model used to recognize context-free languages (CFLs). The chapter provides a detailed introduction to PDA, their historical background and various terminologies related to PDAs including their acceptance. Furthermore, the chapter highlights the role of PDAs across various important domains. By bridging theory and practical applications, PDAs remain a critical component in fields like programming languages and formal verification, showcasing their relevance in both academic research and industry practices.

Keywords: Automata Theory, Context-Free Grammar, Formal Languages, Push-Down Automata.

1 Introduction

Push-Down Automata (PDA) is a fundamental concept in automata theory, an essential area of theoretical computer science that explores how machines process different classes of languages. PDAs extend the capabilities of finite automata by incorporating a stack which is a type of memory structure that allows the storage and retrieval of data in a Last-In-First-Out (LIFO) manner.

Soumodip Kumar Paul
Department of Mathematics, Chandigarh University, Punjab, India.

Babul Sarkar
Department of Mathematics, Chandigarh University, Punjab, India.

This stack memory gives PDAs the ability to recognize context-free languages (CFLs) which finite automata cannot handle. PDAs serve as a computational counterpart to context-free grammars (CFGs), making them an important tool in parsing and language recognition.

A key feature of PDAs is their ability to manage recursive patterns through their stack operations. For example, a PDA can handle nested structures such as balanced parentheses or properly nested expressions. This makes PDAs more powerful than finite automata because they can process input that follows a hierarchical or recursive pattern. Their role in parsing ensures that source code conforms to the predefined grammatical rules, making them crucial components in compilers. Another important feature of PDAs is that they can remember infinite amount of information unlike Deterministic Finite Automata (DFA). (Hopcroft, J. E., Motwani, R., & Ullman, J. D., 2001)

PDAs operate through state transitions based on the current input, the top symbol on the stack, and the rules defined by a transition function. These automata accept languages in two ways: either by reaching a final state or by emptying the stack after processing the input. Such versatility makes them effective for applications across various domains like natural language processing (NLP), XML validation, and expression evaluation.

2 Literature review

Push-Down Automata (PDA) plays a crucial role in automata theory, a branch of computer science that models computational processes. The concept of automata traces back to the mid-20th century, when mathematicians began exploring abstract machines to understand logical tasks and language processing. In 1956, Noam Chomsky introduced context-free grammars (CFGs), a significant breakthrough that formed the basis for PDAs. These automata provide a mechanism to recognize context-free languages (CFLs), a class of languages with recursive patterns, which finite automata cannot handle. (De la Higuera, C., 2010)

Pioneers such as Alan Turing (1936) laid the foundation of computation theory through the concept of Turing Machines, which inspired further research on automata. Stephen Kleene contributed significantly in 1951 by formalizing regular expressions, which describe patterns in regular languages. Although finite automata could process simple patterns, they were insufficient for handling recursive structures like nested parentheses. This gap led to the development of PDAs, as researchers aimed to design computational models capable of recognizing more complex languages.

In subsequent years, John Hopcroft and Jeffrey Ullman expanded on Chomsky's work. Their contributions to automata theory in the 1960s and 1970s formalized the PDA, describing it as a finite automaton combined with a stack—a memory structure that follows the Last-In-First-Out (LIFO) principle. The stack enables PDAs to store

intermediate data temporarily, making them powerful enough to recognize context-free languages.

PDAs operate by processing input strings while manipulating their stack based on transition rules. A PDA can accept a language by either reaching a final state or emptying its stack after reading the entire input. This dual mode of acceptance allows PDAs to efficiently parse programming languages and other structured data formats. For instance, PDAs play an essential role in syntax analysis during compilation, ensuring the correctness of code structure. (J. E. Hopcroft, R. Motwani, and J. D. Ullman, 2006)

Today, PDAs remain integral to compiler design and parsing algorithms, helping process and validate complex code. With their roots in the works of Chomsky, Kleene, and Turing, PDAs exemplify the intersection of mathematics, logic, and computer science, demonstrating how theoretical models can influence practical applications in technology.

3 Methods and materials

Let us now discuss the formal meaning of PDAs and also some important terminologies related to them.

3.1. Formal definition of a PDA A Push-Down Automata (PDA) is a computational model that extends finite automata by using a stack to store symbols. It allows recognition of context-free languages (CFLs) that require memory for matching nested patterns, such as balanced parentheses. PDAs can accept a language in two ways: by reaching a final state or by emptying the stack after reading the input. The stack plays a key role, following the Last-In-First-Out (LIFO) principle, meaning the most recently added symbol must be removed first. (Wintner, S., 2010)

A PDA is formally represented as a 7-tuple: $(Q, \Sigma, S, \delta, q_0, I, F)$

where:
Q: A finite set of states
Σ: Input alphabet
S: Stack symbol
δ: Transition function, which is given by $\delta: Q \times (\Sigma \cup \varepsilon) \times S \rightarrow F \times S$
q_0: Initial state
I: Initial stack symbol
F: Set of final states

Basically, a PDA has 3 components: i) input tape, ii) a control unit and iii) a stack with infinite size. It can be represented as-

3.2. Input tape It is divided into many cells or symbols. The input head is read only and may only move from left to right one symbol at a time

3.3. Finite control unit It has some pointer which points the current symbol to be read.

3.4. Stack A stack is a structure in which we can push and remove the items from one end only. It has an infinite size. In PDA, it is used to store the elements temporarily.

3.5. Instantaneous Description (ID) ID of a PDA is represented by a triplet (q, w, s) where

q: the current state

w: unconsumed input

s: stack contents

ID is an informal notation of how a PDA compute an input string and make a decision that the string is accepted or rejected.

3.6. Turnstile notation It is used for connecting pairs of IDs that represent one or many moves of a PDA . It is denoted by "⊢". For example, (p, b, t) ⊢ (q, w, a) which means while taking a transition from p to q, the input symbol 'b' is consumed and top of the stack is represented by 'a'. (Bengio, Y., Ducharme, R., & Vincent, P., 2003)

3.7 Acceptability of a Push-Down Automata

There are two primary methods by which a PDA can accept input:

- **Acceptance by Final State:**
  In this method, the PDA accepts the input if, after reading all symbols, it reaches a state belonging to the set of final states (F). From the starting state, we can make moves that end up in a final state with any stack values. The stack values are irrelevant after we end up in a final state. This approach is often used when the goal is to verify whether the machine reaches an expected conclusion. For a PDA $(Q, \Sigma, S, \delta, q_0, I, F)$, the language accepted is
  $$L(PDA) = \{w: (q_0, w, I) \vdash^* (q, \varepsilon, X), q \in F\}$$
  where the '*' means in one or more moves.
- **Acceptance by Empty Stack:**
  Here, the PDA accepts the input if, after reading the entire string, the stack becomes empty. This mode is particularly useful for recognizing languages with

balanced or nested structures, such as correctly nested parentheses. For a PDA $(Q, \Sigma, S, \delta, q_0, I, F)$, the language accepted is

$$L(PDA) = \{w: (q_0, w, I) \vdash^* (q, \varepsilon, \varepsilon), q \in Q\}$$

where the '*' means in one or more moves.

Let us now take an example for better understanding.

Example 1: Construct a PDA that accepts the language $L = \{a^n b^n : n \geq 1\}$.
Solution: We need to construct a Push-Down Automaton (PDA) for the language L. This language contains strings with an equal number of a's followed by an equal number of b's. A PDA can keep track of the number of a's using a stack and match them with corresponding b's by popping from the stack.
Working procedure:

- Push an a onto the stack for each a in the input.
- Pop an a from the stack for each b.
- If the input is fully read and the stack is empty, the string is accepted.

Here, minimum length of element is 2, so number of states will be 2+1 = 3.
Also, Q: set of states = $\{q_0, q_1, q_f\}$
Σ: input alphabet = {a, b}
S: stack symbol = $\{z_0, a\}$ [$\because z_0$ is the initial stack symbol and a is pushed later]
δ: Transition function
q_0: Initial state
I: Initial stack symbol
F: Set of final states = $\{q_f\}$.

In this problem, we can write the language as L = {ab, aabb, aaabbb, …}.
Now, let us take an element of L, let's consider aaabbb.

$\therefore$ Execution for input 'aaabbb' is given by-

1. Start at q_0 with stack z_0.
2. Read a: Push a $\rightarrow$ Stack: az_0.
3. Read a: Push a $\rightarrow$ Stack: aaz_0.
4. Read a: Push a $\rightarrow$ Stack: $aaaz_0$.
5. Read b: Pop a $\rightarrow$ Stack: aaz_0.
6. Read b: Pop a $\rightarrow$ Stack: az_0.
7. Read b: Pop a $\rightarrow$ Stack: z_0.
8. End with an empty stack, move to q_f, and accept the string.

So, the input string is completely read and the stack is also empty at the end. Therefore, this PDA successfully recognizes the language L.

4 Applications

Push-Down Automata (PDA) has a wide range of applications, particularly in areas where recursive and hierarchical structures need to be analysed. Its ability to handle context-free languages (CFLs) makes it crucial in various fields of computer science, programming, and data processing. Some applications are discussed below. (Kumar, S., & Kour, G., 2025)

1. **Compiler Design and Syntax Analysis**: During the compilation of source code, the syntax analyser (parser) verifies whether the code conforms to the syntax rules defined by a programming language's grammar. Most programming languages, such as C, Python, and Java, have context-free grammars, and PDAs are used to parse these grammars effectively. The stack-based memory of a PDA allows it to manage nested structures. (Kumar, S., 2024)
2. **Parsing of Markup Languages (XML/HTML)**: Markup languages like HTML and XML involve hierarchical structures with nested tags (e.g., <html><body> Text </body></html>). A PDA can validate whether the tags are properly nested and closed. This ensures that documents conform to the expected structure, which is essential for rendering content correctly on web browsers or processing data for applications. (Singh, M. K., & Kumar, S., 2024)
3. **Natural Language Processing (NLP)**: PDAs play a role in natural language processing by modelling and analysing the syntactic structure of sentences. Many languages follow context-free grammar rules, and PDAs help in parsing sentences to check if they conform to grammatical standards. (Tiwari, K. K., Singh, A., & Kumar, S., 2025)
4. **Expression Evaluation and Arithmetic Calculations**: PDAs are also used in mathematical expression evaluation, such as converting infix expressions to postfix or prefix notations and evaluating them. For example, a PDA can validate an arithmetic expression like (3 + 5) * (2 - 1) by ensuring that the parentheses are balanced and operations are applied in the correct order.

These applications demonstrate how PDAs bridge theory and practice, playing a critical role in syntax analysis, data validation, and software reliability, proving their importance in modern computing.

Conclusions

Push-Down Automata (PDA) is a fundamental model in theoretical computer science that extends the capabilities of finite automata by using a stack-based memory. PDAs are essential for recognizing context-free languages (CFLs), which include many of the grammars used in programming languages, natural language processing, and markup languages. Their stack-based structure makes them well-suited for processing nested patterns and recursive constructs, such as balanced parentheses, nested loops, and mathematical expressions, which finite automata alone cannot handle. This ability to track previous inputs through stack operations gives PDAs greater computational power and makes them highly relevant in various real-world applications.

Despite their practical importance, PDAs remain a theoretical concept used mainly for understanding the limitations and possibilities of automata models. They represent a step between the simpler finite automata and the more powerful Turing machines, bridging theory and practice. While PDAs are not directly implemented in software systems, their logic underpins many practical tools, such as parsers and compilers.

In conclusion, Push-Down Automata demonstrates how theoretical models can address real-world challenges in computing. By bridging gaps between simple automata and complex computations, PDAs play an indispensable role in the fields of programming, natural language processing, and software verification, underscoring their relevance in both academia and industry.

References

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2001). Introduction to automata theory, languages, and computation. *Acm Sigact News*, *32*(1), 60-65.

De la Higuera, C. (2010). *Grammatical inference: learning automata and grammars*. Cambridge University Press.

J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed., Pearson, 2006.

Wintner, S. (2010). Formal language theory. *The Handbook of Computational Linguistics and Natural Language Processing*, 9-42.

Bengio, Y., Ducharme, R., & Vincent, P. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, *3*, 1137-1155.

Kumar, S., & Kour, G. (2025, March). Advanced Machine Learning Approaches for Fastag Fraud Detection. In *2025 International Conference on Automation and Computation (AUTOCOM)* (pp. 149-154). IEEE.

Kumar, S. (2024, May). Advancements in meta-learning paradigms: a comprehensive exploration of techniques for few-shot learning in computer vision. In *2024 International conference on intelligent systems for cybersecurity (ISCS)* (pp. 1-8). IEEE.

Singh, M. K., & Kumar, S. (2024, April). Stress Detection During Social Interactions with Natural Language Processing and Machine Learning. In *2024 International Conference on Expert Clouds and Applications (ICOECA)* (pp. 297-301). IEEE.

Tiwari, K. K., Singh, A., & Kumar, S. (2025, February). A Comprehensive Analysis of CNN-Based Deep Learning Models: Evaluating the Impact of Transfer Learning on Model Accuracy. In *2025 2nd International Conference on Computational Intelligence, Communication Technology and Networking (CICTN)* (pp. 62-67). IEEE.

Kumar, S., Rampal, S., Gaur, M., & Gaur, M. (2024, March). Advanced ensemble learning approach for asthma prediction: Optimization and evaluation. In *2024 International Conference on Automation and Computation (AUTOCOM)* (pp. 283-288). IEEE.