Deep Science Publishing
https://doi.org/10.70593/978-93-49910-92-8_4

Chapter 4: Investigating the computational Power of Pushdown Automata (PDA) and their relation to Context-Free Grammars (CFG)

Miss Daisy, Jitendra Kumar

Abstract: The computational capacity of Pushdown Automata (PDA) and their connection to Context-Free Grammars (CFG)[8] are examined in this term paper. By adding a stack, PDAs expand the capabilities of finite automata and make it possible to recognize context-free languages (CFLs), which are less powerful than Turing machines but more complex than regular languages. The structure of CFLs can then be formally described by Context-Free Grammars. This paper investigates the equivalence between PDAs and CFGs, showing that a PDA can recognize any context-free language and a CFG can generate any language that a PDA accepts. Constructions that transform PDAs into CFGs and vice versa are used to investigate the relationship between these two models. Furthermore, the real-world uses of PDAs and CFGs in domains like compiler design, programming languages, and natural language processing are discussed. This exploration highlights the fundamental role of PDAs and CFGs in understanding language recognition and their significance in computational theory

Keywords: PDA, CFG, Context free languages, etc.

Miss Daisy
Department of Mathematics, Chandigarh University, Punjab, India.

Jitendra Kumar
Department of Mathematics, Marwari College, Darbhanga, Bihar, India.

1 Introduction

Understanding the foundations of computation depends much on formal language theory, with Pushdown Automata (PDA) and Context-Free Grammars (CFG) acting as basic ideas in this area. Context-free languages (CFLs), a class of languages that can be parsed quickly and have broad applications in computer science, are described and recognized by both PDAs and CFGs. A Pushdown Automaton (PDA) is a kind of finite automaton enhanced with a stack, which offers an extra memory structure for identifying languages needing more than finite state memory. This extension lets PDAs identify context-free languages, which are more complex than those recognized by simpler finite automata but simpler than the languages handled by Turing machines. The design of compilers, programming language interpreters, and other computational systems where context-free languages are relevant depends on PDAs. Conversely, a Context-Free Grammar (CFG) is a formal grammar meant to produce context-free languages. Production rules in a CFG describe how start symbol derives strings of symbols in the language. In programming language syntax, where they specify the structure of programming languages and enable the creation of valid programs, CFGs are rather common. (J. E. Hopcroft, R. Motwani, and M. Ullman, 2006)

The equivalence between PDAs and CFGs is among the most significant findings in formal language theory. This equivalence suggests that every language acknowledged by a PDA could be produced by a CFG, and vice versa. Knowing this link not only clarifies how these ideas are used in actual computing activities but also increases our knowledge of the computational power of PDAs and CFGs. This paper aims to explore the computational power of Pushdown Automata, analyze their relationship with Context-Free Grammars, and discuss their significance in theoretical and practical applications. By examining the equivalence between these two models, we will demonstrate how PDAs and CFGs serve as two sides of the same coin when it comes to the recognition and generation of context-free languages. Through this exploration, the paper provides a clearer understanding of the strengths and limitations of PDAs, the role of CFGs in language design, and the importance of these concepts in computational theory. (Sipser, M., 2012)

2 Pushdown Automata (PDA): Definition and Properties

A Pushdown Automaton (PDA) is a computational model that adds to the power of a finite automaton by introducing a second memory component called stack. The stack gives PDA the cap ability to hold an unbounded amount of information, which is important for recognizing context-free languages (CFLs) that are not recognizable by less powerful finite state machines (FSMs). The PDA can therefore handle more complex languages than finite automata, but still less powerful than Turing machines. (Kurtz, D., May, M., 2018)

A PDA is formally defined as a 7-tuple:

$P = (Q, \Sigma, \Gamma, \delta, q0, Z0, F)$

Where:

- **Q** is a finite set of states, representing the different configurations the PDA can be in.
- **Σ** is the input alphabet, a finite set of symbols that the PDA can read from the input string.
- **Γ** is the stack alphabet, a finite set of symbols that can be pushed onto or popped from the stack.
- **δ** is the transition function, which defines the state transitions. Specifically, $\delta: Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow 2Q \times \Gamma*$, where the PDA moves from one state to another, based on the current input symbol and the symbol on top of the stack, while possibly modifying the stack.
- **q0** is the initial state, where the computation begins.
- **Z0** is the initial stack symbol, placed on the stack at the beginning of the computation.
- **F** is the set of accepting states, where the PDA halts and accepts the input if it reaches any state in this set.

2.1. Basic Operation of a PDA

The function of a PDA is to read input symbols and change the stack according to the current state and the top symbol of the stack. In each step, the PDA may either:

- Read an input symbol and move to another state and alter the stack (push, pop, or do nothing to the stack),
- Alternately, it can shift without reading an input symbol, i.e., through ϵ-transitions, under which the PDA can act only depending upon the stack contents.

This stack-based memory allows PDAs to process recursive structures in languages, something highly relevant in the context of context-free languages.2. Deterministic vs. non-deterministic PDAs. (Minsky, M. 1967)

There are two primary types of PDAs:

2.1.1. Deterministic Pushdown Automata (DPDA):

- A Deterministic PDA is one in which, for any combination of the current state, input symbol, and top stack symbol, there may be at most one possible action (transition).

- DPDAs are more constrained in operation but have a simpler behaviour than NPDAs.
- A major property of DPDAs is that they can only accept deterministic context-free languages (DCFLs), which are themselves a subset of the context-free languages.

2.1.2 Non-Deterministic Pushdown Automata (NPDA):

- A Non-Deterministic PDA provides multiple transitions from the same combination of current state, input symbol, and top stack symbol.
- NPDAs are more expressive and can recognize all context-free languages, whereas DPDAs can recognize only deterministic context-free languages.
- Non-determinism enables an NPDA to "branch" along several computation paths, accepting a string if at least one of the paths terminates in an accepting configuration.

2.2. Types of Transitions

There are two main types of transitions in a PDA:

2.2.1 Input Transitions: The PDA reads an input symbol and makes a state transition while working on the stack. The PDA can:

- Push one or more symbols onto the stack,
- Pop the top symbol from the stack,
- Or leave the stack unchanged (if the input symbol doesn't affect the stack).

2.2.2 ϵ-Transitions: These transitions do not use an input symbol but enable the PDA to move from one state to another and alter the stack. ϵ-transitions allow the PDA to handle some patterns or conditions without having to read additional input, giving the flexibility required for context-free languages.4. Acceptance Conditions

A PDA can accept input strings in one of two ways:

1. **Final-State Acceptance**: The PDA accepts the input if, after reading the entire string, it reaches an accepting state in F.

2. **Empty Stack Acceptance**: The PDA accepts the input if, after processing all input symbols, the stack is empty (i.e., all symbols pushed onto the stack have been popped off).

In practice, PDAs typically use final-state acceptance or empty-stack acceptance, or a combination of both, depending on the definition.

2.3 PDA vs. Finite Automaton

While FAs may accept RLs, they have no memory for managing context-free constructs like recursive patterns or nested parentheses. PDAs, being equipped with a stack, close the gap here. For example, a finite automaton can never accept the language $L=\{ a^nb^n\}$, as it lacks the ability to compare and count a's and b's. But a PDA can identify this language by pushing a's onto the stack and popping them off while scanning the corresponding b's, making sure that the counts are equal. (Ginsburg, S., Parikh, R. 1966)

2.4 PDA Example

Consider a simple PDA for recognizing the language $L=\{a^nb^n \mid n\geq 0\}$. The PDA would operate as follows: (H. R. Lewis and C. H. Papadimitriou, 1981)

1. Start in the initial state q0 with the stack empty.
2. For each a read from the input, push an A symbol onto the stack.
3. For each b read, pop an A from the stack.
4. Accept the input if the entire string is processed, the stack is empty, and the PDA is in an accepting state.

This behaviour ensures that the number of a's and b's are the same, which is characteristic of context-free languages.

2.5. Applications of PDAs

Pushdown Automata find various applications in different areas of computer science, mainly where context-free languages are applied: (Kumar, S., & Kour, G., 2025, Sharma, S., Rana, S., & Dubey, S. S., 2024)

- **Compiler Design**: PDAs are employed in syntax analysis (parsing) in compilers when the grammar of a programming language is usually context-free.
- **Programming Language Design**: PDAs assist in the construction of parsers which check if a program provided is correct according to the syntactical rules specified by a CFG. (Kumar, S.
- **Natural Language Processing (NLP):** Context-free grammars are applied in linguistics to describe the syntax of natural languages, and PDAs supply a theoretical framework for parsing these languages.

3 Context-Free Grammars (CFG): Definition and Properties

Context-Free Grammars (CFGs) are formal descriptions employed for defining the form of context-free languages (CFLs), a family of languages that finds extensive use in

programming language design, compiler construction, and natural language processing. A CFG comprises a set of production rules permitting the derivation of strings over an alphabet, wherein the rules control how symbols of a language are replaced by symbols.

A Context-Free Grammar is formally defined as a 4-tuple:

G= (V, T, P, S)

Where:

- **V** is a finite set of **variables** or **non-terminal symbols**, which are symbols used to represent patterns or structures in the language. These are placeholders that will eventually be replaced by terminal symbols or other variables.
- **T** is a finite set of **terminal symbols**, which are the basic symbols of the language. Terminal symbols are the "building blocks" of the strings generated by the grammar and are not further replaced during the derivation process.
- **P** is a finite set of **production rules**, each of which is of the form $A \rightarrow \alpha$, where $A \in V$ is a non-terminal symbol, and α is a string consisting of both terminal and non-terminal symbols. These rules define how non-terminals can be rewritten into other symbols or sequences of symbols.
- **S** is the **start symbol**, a special non-terminal symbol from which derivations begin. The start symbol is the root of the derivation tree for generating strings in the language.

The key feature of CFGs is that the left-hand side of each production rule consists of a single non-terminal symbol, which is the defining characteristic of "context-free" grammars. This makes CFGs simpler and more tractable than **context-sensitive grammars**, where the left-hand side of a production rule can contain multiple symbols.

3.1. Basic Operation of a CFG

In a CFG, the derivation process starts with the start symbol SSS, and by applying production rules, the derivation continues until only terminal symbols are left, which constitute a valid string in the language. The rules determine how non-terminal symbols can be expanded into strings of other symbols, step by step, ending up with a full string of terminal symbols. A derivation can be represented as follows:

$S \Rightarrow \alpha 1 \Rightarrow \alpha 2 \Rightarrow \cdots \Rightarrow w$

Where w is a string of terminal symbols, and $\alpha 1$, $\alpha 2$, …. are intermediate derivations that involve replacing non-terminals with sequences of symbols.

3.2. Types of Productions

In a Context-Free Grammar, the production rules typically follow the general form:

$A \rightarrow \alpha A$

Where:

- A is a single non-terminal symbol on the left-hand side.
- α is a string of terminals and/or non-terminals on the right-hand side.

Production rules can vary in form but must always satisfy this structure, which ensures that each derivation step applies to a single non-terminal symbol. For example:

- **S→aSb**: This rule replaces S with aSb, which can lead to recursive derivations.
- **S→ab**: This rule directly generates the string "ab", replacing S with terminal symbols.

3.3. Derivations and Parse Trees

A derivation is a series of uses of production rules beginning with the start symbol and resulting in a string of terminal symbols. The derivation structure can be represented by a parse tree or derivation tree, which is a tree representation such that:

• The root is the start symbol.

• The interior nodes are non-terminals.

• The terminal symbols are the leaves.

Parse trees facilitate visualization of the application of the production rules in the process of derivation as well as of the hierarchical organization of the resultant string. As an illustration, a derivation of the string "aab" through a grammar whose rules are S→aSb and S→ab would consist of several applications of the rules and have an accompanying parse tree**.**

3.4. Properties of Context-Free Grammars

Context-Free Grammars possess some important characteristics differentiating them from other formal grammars, and these are the key to their function in language theory:

- **Generative Power**: A CFG can generate any context-free language (CFL). CFLs comprise many standard programming languages (such as C, Java, and Python), mathematical expressions, and other syntactical constructs.
- **Efficient Parsing**: Although more advanced grammars such as context-sensitive grammars need non-linear or multi-pass parsing, context-free grammars are

comparatively simpler to parse. There are effective algorithms such as LL parsing and LR parsing that can parse context-free languages in linear time or close to linear time.

- **Ambiguity:** A CFG is ambiguous if it is possible to derive a string more than one way using the production rules, which leads to more than one distinct parse tree for the same string. Ambiguity is not desirable in most applications, particularly in programming languages, since it can cause uncertainty regarding the meaning of a program or expression. For instance, the expression a+b×c can be interpreted differently based on the order of operations.
- **Context-Free Language**: The language produced by a CFG is a context-free language (CFL), which is the proper subset of recursively enumerable languages (RE) but the superset of regular languages (RL). CFLs contain nested structures, like balanced parentheses or matching HTML tags, which are not possible for regular languages to process.

3.5. Examples of Context-Free Grammars

Here are a few examples of CFGs and the languages they generate:

3.5.1. Language of Palindromes: The language $L=\{w|w=w^R\}$ (i.e., palindromes) can be generated by the following CFG:

$$S \rightarrow aSa|bSb|\epsilon$$

This grammar generates palindromes over the alphabet {a, b}. It uses the recursive production rules to ensure that the string is symmetric.

3.5.2. Balanced Parentheses: The language $L=\{(n)n|n\geq 0\}$ (i.e., properly nested parentheses) can be generated by the following CFG:

$$S \rightarrow (S)S|\epsilon$$

This grammar ensures that parentheses are properly balanced by recursively nesting pairs of parentheses.

1. **Arithmetic Expressions**: An arithmetic expression with addition and multiplication can be described by the following CFG:

$$E \rightarrow E+T|T$$

$$T \rightarrow T\times F|F$$

$$F \rightarrow (E)|a$$

Here, E represents an expression, T represents a term, and F represents a factor. This CFG generates expressions like a+a×a and ensures proper precedence of operators.

3.6. Normal Forms for Context-Free Grammars

There are two important **normal forms** for CFGs that simplify their analysis and use in parsing algorithms:

- **Chomsky Normal Form (CNF)**: A CFG is in Chomsky Normal Form if every production rule is of the form A→BC (where B and C are non-terminal symbols) A→a (where a is a terminal symbol). CNF is useful for algorithms such as the CYK (Cocke-Younger-Kasami) parser.
- **Greibach Normal Form (GNF)**: A CFG is in Greibach Normal Form if every production rule is of the form A→aα, where a is a terminal symbol and α is a string of non-terminal symbols. GNF is useful for constructing **top-down parsers**.

3.7. Applications of Context-Free Grammars

- Context-Free Grammars play a key role in computer science and linguistics in most areas:
- **Compiler Construction**: CFGs are extensively applied to specify programming language syntax. Compilers apply CFGs in syntax analysis (parsing) to translate source code into an interpretable or compitable form that can be translated into machine code.
- **Natural Language Processing (NLP):** CFGs are employed to represent the syntactic structure of natural languages. Syntax trees derived from CFGs assist in applications such as sentence parsing, part-of-speech tagging, and machine translation.
- **Mathematical Expressions**: CFGs are employed to specify the form of arithmetic expressions, regular expressions, and other mathematical formal languages.

4 Relationship Between Pushdown Automata (PDA) and Context-Free Grammars (CFG)

Pushdown Automata (PDA) and Context-Free Grammars (CFG) are two basic formal models for defining context-free languages (CFLs). PDAs are employed to accept languages, while CFGs are employed to produce languages. These two models may differ in terms of purpose but are equivalent as far as they can define the same class of languages — the context-free languages. This similarity creates a strong relationship between the two ideas, demonstrating that any context-free language is recognizable by

a PDA and vice versa, that any language that is accepted by a PDA is generable by a CFG.

This section explores the equivalence and correspondence between PDAs and CFGs, specifically how they can be converted into each other, their common computational power, and the theoretical significance of their relationship.

4.1. Equivalence of PDAs and CFGs

The key concept of the connection between PDAs and CFGs is that both recognize and produce the same type of language — the context-free languages. The formal equivalence can be divided into two large components:

•All context-free grammars are reducible to an equivalent PDA which accepts the same language.

•All PDAs can be transformed into an equivalent context-free grammar that produces the same language.

Thus, PDAs and CFGs are equivalent in terms of the languages they define.

4.2. From a Context-Free Grammar to a Pushdown Automaton (CFG → PDA)

A context-free grammar (CFG) can be translated into a Pushdown Automaton (PDA) that accepts the same language. The conversion of a CFG to a PDA is not very complex and is done by following the steps below:

Steps for Conversion:

4.2.1 Define the PDA Structure: The PDA PPP will have the following components:

- **States**: A PDA based on a CFG will have a single state, because the state in a PDA is typically used to manage the stack, not the input.
- **Stack Alphabet**: The stack alphabet of the PDA is composed of the non-terminal symbols of the CFG, plus a special bottom-of-stack symbol.

4.2.2 Stack Operations: The PDA will simulate the production rules of the CFG by manipulating its stack:

- For each non-terminal in the CFG, the PDA pushes the corresponding right-hand side of the production onto the stack.

- If the current non-terminal is replaced by terminals, the PDA pops symbols from the stack and matches them with the input.

4.2.3 Simulating Derivations: The PDA operates by reading the input string from left to right. At each step, the PDA applies rules based on the top stack symbol:

- If the top of the stack is a non-terminal, the PDA expands it according to the CFG's production rules.
- If the top of the stack is a terminal, the PDA matches it with the corresponding symbol in the input string.

4.2.4 Acceptance Condition: The PDA accepts the input if it processes the entire string and the stack is empty at the end. This ensures that all non-terminals have been replaced and the string is generated by the grammar.

Example:

Consider the CFG

S→aSb|ab

We can construct a PDA as follows:

- Start q0, where the PDA starts with the stack symbol S.
- On reading a, push S onto the stack.
- On reading b, pop S and ensure that the input is matched correctly.
- Acceptance: The PDA accepts when it reaches the end of the input and the stack is empty.

This PDA would recognize strings like a^nb^n, matching the behavior of the CFG.

4.3. From a Pushdown Automaton to a Context-Free Grammar (PDA → CFG)

Conversely, a PDA can be converted into a CFG that generates the same language it accepts. This process is slightly more involved, as it requires simulating the transitions and stack operations of the PDA within the structure of a CFG.

Steps for Conversion:

1. **Define the CFG Structure**: The grammar will have non-terminal symbols corresponding to pairs of states in the PDA. A non-terminal Ap,q in the CFG represents the possibility of the PDA going from state p to state q with the stack being empty.
2. **Create Production Rules**:

- **Initial and final states**: For every pair of states p and q, create a non-terminal Ap,q. The start symbol of the CFG will correspond to the initial state of the PDA and the accepting state(s).
- **Stack transitions**: For each transition in the PDA where the machine reads a symbol, pop, or push a symbol onto the stack, create corresponding production rules in the CFG that simulate these stack operations.

3. **Stimulate the Stack Operations of the PDA:** The most difficult part of the conversion process is emulating the stack operations of the PDA in a CFG. The grammar needs to be constructed so that it mimics the stack operations of the PDA, so that the non-terminal symbols of the CFG accurately reflect the correct "states" of the stack of the PDA at each step of the derivation.
4. **Acceptance Condition:** The CFG will produce strings that represent the paths the PDA follows from its start state to an accepting state, with the derivations agreeing with the PDA's transitions.

Example:

Consider a simple PDA that accepts the language $\{a^nb^n$ (the same language we used in the previous example for the CFG). This PDA has:

- **States** q0 and q1 with q0 as the initial state and q1 as the accepting state.
- **Transitions** for pushing and popping symbols onto the stack, as well as consuming input.

From this PDA, we can derive the following CFG:

$S \rightarrow aSb|abS$

This CFG generates the same language as the PDA accepts, demonstrating the equivalence between the two models.

4.4 Theoretical Significance of the Equivalence

The equivalence of PDAs and CFGs has a number of significant implications:

4.4.1 Formal Language Theory

The equivalence establishes that the class of context-free languages is exactly the class of languages that can be accepted by a PDA or defined by a CFG. This basic result makes

us aware of the limits of computation models and the expressive power of PDAs and CFGs.

4.4.2 Parsing and Compiler Construction

In practical terms, this equivalence implies that any context-free grammar employed to define the syntax of a programming language can be parsed by a PDA, and vice versa, any language accepted by a PDA can be defined by a CFG. This is the basis for most contemporary parsing algorithms employed in compiler construction, including LL parsers, LR parsers, and CYK parsers, which are derived from CFGs but frequently employ stack-based data structures (such as PDAs) during execution.

4.4.3 Programming Language Design

Most programming languages are context-free in their syntax. The equivalence of CFGs and PDAs guarantees that the syntax of these languages can be both specified and accepted by equivalent models. It also shows that context-free languages, although useful, are incapable of expressing all constructs of programming languages, particularly those involving richer memory devices (e.g., context-sensitive languages).

5 Computational Power of Pushdown Automata (PDA)

Pushdown Automata (PDA) are a model of computation that augment finite automata with a stack as memory. This stack gives PDAs the capacity to accept context-free languages (CFLs), a language class more expressive than those accepted by finite automata (which can only deal with regular languages).

The chief characteristic of PDAs is the capability to store and manipulate symbols in the stack, enabling them to process nested structures and recursion, which prevail in programming languages, mathematical formulas, and natural language processing.

Key Points:

- **PDA Power**: PDAs can accept context-free languages (e.g., balanced parentheses, arithmetic expressions, many programming language constructs).
- **Deterministic vs. Non-deterministic:** Deterministic PDAs (DPDAs) are able to recognize a lot of CFLs but not some languages (such as palindromes). Non-deterministic PDAs (NPDAs) are able to recognize all CFLs, including more complicated ones.
- **Chomsky Hierarchy:** PDAs occupy Type 2 in the Chomsky hierarchy, which is able to recognize context-free languages, which are strictly more powerful than regular languages but weaker than context-sensitive or recursively enumerable languages.

6 Limitations:

- Incapable of recognizing non-context-free languages.
- Fixed memory: Only one stack without random access or backtracking.
- Weak deterministic PDAs: Certain context-free languages cannot be recognized by deterministic PDAs.
- Incapable of processing context-sensitive languages or higher-level classes of languages.
- Not Turing-complete, and thus incapable of general computation besides recognition of context-free languages.

These constraints demonstrate that although PDAs are strong for some purposes such as syntax analysis in programming languages, they are inappropriate for applications involving more complex memory structures or general computation.

7 Real-Life Applications of Pushdown Automata (PDA) and Context-Free Grammars (CFG)

Pushdown Automata (PDA) and Context-Free Grammars (CFG) are fundamental in the theory of formal languages, with applications extending far beyond academia. Their computational power and ability to handle recursive and nested structures make them invaluable in various real-world domains. Below are key areas where PDAs and CFGs play a critical role in real-life applications:

7.1 Compiler Design and Syntax Analysis

In the process of converting high-level programming languages into machine-readable code, **compilers** perform multiple tasks, one of which is **syntax analysis** (also known as parsing). The syntax analysis phase checks if the code follows the grammar rules of the programming language. (Singh, M. K., & Kumar, S., 2024)

Role of PDAs and CFGs:

- **CFGs**: Programming languages such as C, Java, and Python are defined by CFGs, which describe the syntax rules (e.g., valid expressions, function definitions, and control structures).
- **PDAs**: PDAs are used to implement parsers, which verify the syntax of a program. The stack of a PDA is instrumental in recognizing recursive constructs like nested parentheses or blocks of code (if-else structures, loops, etc.).

Example: When a program's source code is compiled, PDAs are used to ensure the code adheres to the syntax of the programming language. For instance, if you have the

expression (3 + 5) * 2, the PDA checks that the parentheses are properly balanced before evaluating the expression.

7.2 Natural Language Processing (NLP)

In **Natural Language Processing (NLP)**, the goal is to enable machines to understand and process human languages. Human languages have recursive and nested structures that can be modeled effectively with CFGs and PDAs. (Tiwari, K. K., Singh, A., & Kumar, S., 2025)

Role of PDAs and CFGs:

- **CFGs**: These are used to define the grammar rules of natural languages. For instance, rules for constructing sentences like noun phrases, verb phrases, etc., can be captured using CFGs.
- **PDAs**: PDAs are used in parsers to process and understand syntactic structures in text. They can handle recursion and nested structures, which are common in language (e.g., "The man who met the woman is happy").

Example: In NLP, PDAs help in **sentence parsing**. For a sentence like "The cat sat on the mat," a CFG would define the rules for sentence structure (subject, verb, object), and a PDA would parse the sentence, ensuring it follows the syntax.

Applications:

- **Machine Translation**: Translating sentences from one language to another often requires understanding the grammatical structure. PDAs and CFGs help in parsing and translating complex sentences with nested clauses.
- **Speech Recognition**: In speech-to-text systems, PDAs help parse the structure of spoken sentences to convert them into written text.

7.3 Mathematical Expression Evaluation

Mathematical expressions often contain nested operations that need to be parsed correctly for evaluation. This is where PDAs and CFGs come into play.

Role of PDAs and CFGs:

- **CFGs**: Define the syntax of mathematical expressions, such as operator precedence and the grouping of operations.
- **PDAs**: Used to evaluate expressions in a manner that respects operator precedence and parentheses, making sure nested operations are computed in the correct order.

Example: Consider the expression (3 + (5 * 2)) - 4. The PDA uses its stack to first compute the multiplication inside the parentheses and then perform the addition and subtraction in the correct order.

Applications:

- **Evaluators for Programming Languages**: PDAs are used in interpreters and compilers to evaluate arithmetic expressions in programming languages.
- **Calculators**: In graphical and scientific calculators, PDAs and CFGs are used to evaluate complex mathematical expressions, ensuring proper handling of operations like addition, multiplication, and parentheses.

7.4 XML and HTML Document Parsing

XML (eXtensible Markup Language) and HTML (Hypertext Markup Language) are used extensively to structure data for web applications. These languages contain nested tags, which can be validated and parsed using PDAs and CFGs.

Role of PDAs and CFGs:

- **CFGs**: Define the structure of XML and HTML documents. Tags must follow specific rules for the document to be considered well-formed.
- **PDAs**: Used to parse the nested structure of XML and HTML. PDAs check that each opening tag has a corresponding closing tag, ensuring the document is well-formed.

Example: An XML document like:

xml

Copy

<book>

<title>Introduction to Automata Theory</title>

<author>John Doe</author>

</book>

PDAs check that each <book>, <title>, and <author> tag is properly opened and closed.

Applications:

- **Web Browsers**: Browsers use PDAs to parse HTML and render web pages. A well-formed HTML page ensures that the browser displays content correctly.

- **Data Validation**: PDAs help ensure that XML documents used in web services or databases are well-formed and adhere to the structure defined by a schema.

Conclusion

In this paper, we have discussed the inherent connection between Pushdown Automata (PDA) and Context-Free Grammars (CFG), two of the fundamental models of the theory of computation. In our study, we established that PDAs and CFGs are equally expressive in the sense that they can recognize each context-free language (CFL) using some PDA and accept each language recognized by a PDA using a CFG. This identification emphasizes the strong relationship between automata and formal language theory, yielding both theoretical framework and practical machinery for parsing languages and compiler design. Additionally, the computational capacity of PDAs, though formally less than Turing machines', is adequate to simulate a large class of syntactic structures used in programming languages and natural languages. The incorporation of a stack into PDAs adds a restricted form of memory that allows nested and recursive patterns to be recognized—an ability missing in finite automata. Although deterministic and nondeterministic PDAs are distinct in power, with the latter being strictly more powerful, this difference further highlights the subtleties in automata theory and the significance of computational models in language hierarchy understanding. Overall, the research on PDAs and their equivalence to CFGs not only increases our knowledge of formal languages but also reaffirms their relevance in real-world applications like syntax analysis and language design.

References

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

M. Sipser, Introduction to the Theory of Computation, 3rd ed. Cengage Learning, 2012.

D. Kurtz and M. May, Automata and Computability. Pearson, 2018.

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.

H. R. Lewis and C. H. Papadimitriou, Elements of the Theory of Computation. Prentice Hall, 1981.

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.

S. Ginsburg and R. Parikh, "Context-Free Languages," J. ACM, vol. 13, no. 2, pp. 189–191, 1966.

M. Minsky, Computation: Finite and Infinite Machines. Prentice-Hall, 1967.

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.

Sharma, S., Rana, S., & Dubey, S. S. (2024). ESAF: An Enhanced and Secure Authenticated Framework for Wireless Sensor Networks. *Wireless Personal Communications*, *136*(3), 1651-1673.

Kumar, M., Gupta, M. K., Mishra, R. K., Dubey, S. S., Kumar, A., & Hardeep. (2021). Security Analysis of a Threshold Quantum State Sharing Scheme of an Arbitrary Single-Qutrit Based on Lagrange Interpolation Method. In *Evolving Technologies for Computing, Communication and Smart World: Proceedings of ETCCS 2020* (pp. 373-389). Springer Singapore.