Lesson 3: Context-Free Grammars and Pushdown Automata

Introduction to context-free grammars (CFG)

Context-Free Grammars (CFG) are an essential concept in computer science and linguistics for describing the syntax or structure of formal languages. They serve as a foundation for many areas of study, including programming languages, compilers, natural language processing, and theoretical linguistics. In this text, we will explore the fundamentals of CFG and their significance in various domains.

At its core, a context-free grammar is a set of production rules that define how symbols can be combined to form valid sentences in a language. These rules consist of a left-hand side (LHS) and a right-hand side (RHS), separated by an arrow (→). The LHS represents a non-terminal symbol, while the RHS consists of a sequence of terminals and non-terminals. Terminals are symbols that cannot be further expanded, while non-terminals can be replaced by a sequence of terminals and non-terminals.

A production rule essentially states that a non-terminal symbol can be replaced by a particular sequence of symbols. For example, consider a simple CFG for arithmetic expressions:

1. E → E + E
2. E → E - E
3. E → (E)
4. E → number

In this grammar, the non-terminal symbol E represents an arithmetic expression, and the terminals are the plus sign (+), minus sign (-), parentheses ((), and numbers. The production rules define how expressions can be formed. Rule 1 states that an expression (E) can be replaced by an expression (E) followed by a plus sign (+) and another expression (E). Similarly, the other rules define alternative ways of constructing valid expressions.

CFGs are often represented using a notation called Backus-Naur Form (BNF), which provides a concise and standardized way to describe the production rules. BNF uses angle brackets (<>) to denote non-terminal symbols and quotes to represent terminals. For example, the arithmetic expression CFG can be written in BNF as:

<expression> ::= <expression> "+" <expression>
| <expression> "-" <expression>
| "(" <expression> ")"
| <number>

The BNF notation allows for the compact representation of complex grammars, making it easier to define the syntax of programming languages, define the structure of natural languages, or even model other formal languages.

One crucial property of context-free grammars is that they exhibit a hierarchical structure. This property implies that sentences generated from a CFG can be analyzed recursively, breaking them down into smaller constituents until reaching the terminals. This hierarchy is useful for parsing algorithms, which can determine the structure of a sentence based on the given CFG.

In summary, context-free grammars provide a powerful framework for describing the syntax of formal languages. They are widely used in various fields, from programming language design to natural language processing. By defining production rules that govern how symbols can be combined, CFGs enable us to generate and analyze sentences in a structured and hierarchical manner. Understanding CFGs is fundamental for anyone interested in language processing, formal language theory, or computational linguistics.

Derivations and parse trees

Deriviations

A derivation in a CFG provides a formal and systematic way to generate sentences by applying production rules. It demonstrates the step-by-step transformation of the start symbol (often denoted as S) into a sentence composed of terminals.

The process of deriving a sentence begins with the start symbol. At each step, a non-terminal symbol is chosen, and a production rule is applied to replace it with its corresponding RHS expansion. The choice of the non-terminal symbol can depend on various factors, such as the desired structure of the generated sentence or specific parsing algorithms being employed.

The application of production rules continues until all non-terminal symbols have been replaced, resulting in a fully derived sentence consisting solely of terminals. The sequence of rule applications defines a unique derivation for a given sentence, allowing us to trace the path from the initial start symbol to the final sentence.

Derivations are useful for understanding the structure and syntax of a language defined by a CFG. They provide insights into how different non-terminal symbols are combined to form valid sentences. By examining the steps of a derivation, one can gain a deeper understanding of how the language's grammar operates and how sentences are constructed according to the specified rules.

Derivations also play a significant role in the analysis and processing of sentences. They form the basis for parsing algorithms, which aim to determine the syntactic structure of a sentence based on a given CFG. By employing techniques such as top-down parsing or bottom-up parsing, parsing algorithms utilize derivations to construct parse trees or identify potential parsing errors.

Understanding derivations in CFGs is essential for various applications in computer science and linguistics. It enables the design and implementation of programming languages and compilers, as well as the development of natural language processing systems. Additionally, derivations serve as a theoretical foundation for formal language theory, allowing for the classification and comparison of different language classes based on their grammatical rules and structural properties.

Let's consider a simple CFG as an example:

1. S → aSb
2. S → ε

Here, rule 1 states that the start symbol S can be replaced by the sequence "aSb," where "a" and "b" are terminals, and rule 2 indicates that S can be replaced by the empty string (ε).

To illustrate a derivation, let's derive the sentence "aabbb" using the given CFG:

S (Apply rule 1)
aSb (Apply rule 1)
aaSbb (Apply rule 1)
aabSbbb (Apply rule 2)

aabbb

The derivation above shows the step-by-step process of replacing the non-terminal S according to the production rules until the final sentence "aabbb" is obtained.

Parse trees

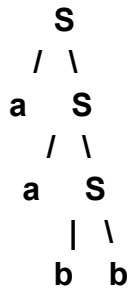
Parse trees, also known as derivation trees or syntax trees, provide a visual representation of the structure of a sentence derived from a CFG. They illustrate the hierarchical relationship between the symbols in the sentence and show how the production rules have been applied.

In a parse tree, each node represents a symbol (either a non-terminal or a terminal), and the edges represent the application of a production rule. The root of the tree represents the start symbol, and the leaves correspond to the terminals of the sentence. The internal nodes represent non-terminals, and their children represent the symbols obtained by applying a specific production rule.

The construction of a parse tree begins with the start symbol as the root node. Each subsequent level in the tree represents the expansion of non-terminal symbols into their corresponding RHS expansions according to the production rules. The children of each non-terminal node represent the symbols obtained by applying a specific production rule.

To illustrate the hierarchical relationships and the application of production rules, parse trees often adhere to certain conventions. For example, the leftmost child of a non-terminal node corresponds to the leftmost symbol in the RHS of the associated production rule. This convention helps maintain the order of symbols in the derived sentence.

Using the same example sentence "aabbb," we can construct a parse tree based on the given CFG:



In this parse tree, the root node represents the start symbol S. The leftmost child of the root node corresponds to the first "a" in the derived sentence, and the rightmost child corresponds to the second "a." The right child of the second non-terminal S represents the "b" in the derived sentence. Finally, the leaves of the tree correspond to the remaining "b" terminals.

Parse trees provide a visual representation of the hierarchical structure of a sentence derived from a CFG. They clearly illustrate how the production rules are applied at each step, showing the expansion of non-terminals into terminals. By examining the parse tree, one can easily understand the syntactic structure of the sentence and trace the path from the start symbol to the terminals.

Parse trees are particularly useful in parsing algorithms, where they serve as a guide for analyzing the syntactic validity of a sentence based on the given CFG. They can also aid in subsequent processing steps, such as semantic analysis or code generation.

In summary, derivations and parse trees are essential concepts in the study of CFGs. Derivations demonstrate the step-by-step process of replacing non-terminals to derive a sentence, while parse trees provide a graphical representation of the hierarchical structure of the sentence. Together, these concepts aid in understanding and analyzing the syntax of formal languages and are crucial in areas such as parsing, language processing, and compiler design.

Pushdown automata (PDA) and their relation to CFGs

Pushdown Automata (PDA) and Context-Free Grammars (CFG) are fundamental concepts in formal language theory and play a vital role in the study of formal languages and their computational properties. The relationship between PDAs and CFGs goes beyond their ability to recognize the same class of languages, extending to their applications in language processing, parsing algorithms, compiler design, and formal language theory.

A Pushdown Automaton (PDA) is a computational model that extends the capabilities of finite automata by introducing a stack as an additional memory component. The PDA consists of a finite control, an input tape, and a stack. It operates by reading symbols from the input tape, transitioning between states in the finite control based on the current input symbol and the top symbol of the stack, and performing push, pop, or no

operation on the stack. The stack allows the PDA to store and retrieve symbols, facilitating the processing of languages with hierarchical structures.

Context-Free Grammars (CFGs), on the other hand, are formal systems used to describe the syntax or structure of formal languages. A CFG consists of a set of production rules that define how symbols can be combined to form valid sentences in a language. The rules consist of a left-hand side (LHS) and a right-hand side (RHS), separated by an arrow (→). The LHS represents a non-terminal symbol, while the RHS consists of a sequence of terminals and non-terminals. CFGs provide a powerful framework for defining and analyzing the structure of formal languages, including programming languages and natural languages.

The connection between PDAs and CFGs lies in their ability to recognize the same class of languages, known as context-free languages. A context-free language is precisely the language generated by a CFG. This connection is established by the "Chomsky-Schützenberger representation theorem," which states that for any CFG, there exists an equivalent PDA that can recognize the language generated by that CFG, and vice versa.

Given a CFG, an equivalent PDA can be constructed that simulates the derivation process of the CFG. The stack in the PDA corresponds to the derivation in the CFG, storing the non-terminal symbols encountered during the parsing process. The transitions in the PDA are defined based on the production rules of the CFG, with the stack operations reflecting the replacement of non-terminal symbols during the derivation.

Conversely, given a PDA, it is possible to construct an equivalent CFG. The non-terminals of the CFG correspond to the states of the PDA, and the production rules are defined based on the transitions of the PDA. The start symbol of the CFG corresponds to the initial state of the PDA, and the productions are constructed to mimic the stack operations of the PDA.

This equivalence between PDAs and CFGs highlights the inherent connection between the generation and recognition of context-free languages. It demonstrates that the context-free languages can be recognized by a simple form of automaton with a stack, which captures the notion of the hierarchical structure exhibited by CFGs.

The relationship between PDAs and CFGs has significant implications in various areas of computer science. In language processing, PDAs and CFGs are utilized in parsing algorithms, which analyze the syntactic structure of sentences based on the given CFG.

The equivalence between PDAs and CFGs enables the use of parsing algorithms that exploit either formalism to determine the structure of a sentence. Parsing algorithms, such as top-down parsing and bottom-up parsing, utilize the stack-like behavior of PDAs or the derivation process of CFGs to construct parse trees and determine the validity of sentences.

In compiler design, the equivalence between PDAs and CFGs is employed in the syntax analysis phase, where the input program's syntax is checked against the specified grammar. This phase utilizes techniques such as LL (left-to-right, leftmost derivation) or LR (left-to-right, rightmost derivation) parsing, which make use of the CFG representation of the language and leverage the stack-like behavior of PDAs.

Moreover, the relationship between PDAs and CFGs has theoretical implications in formal language theory. It allows researchers to reason about the properties and limitations of context-free languages using the formalism of PDAs. Properties such as closure properties, decidability, and complexity can be studied through the lens of PDAs and CFGs, providing insights into the computational nature of context-free languages.

In summary, Pushdown Automata (PDAs) and Context-Free Grammars (CFGs) are intimately connected formal models. They share a close relationship by recognizing the same class of languages, known as context-free languages. The equivalence between PDAs and CFGs enables the interchangeability of representations and facilitates their application in language processing, parsing algorithms, compiler design, and theoretical studies of formal languages. This relationship forms a foundation for understanding the structure and computation of context-free languages in various domains of computer science and formal language theory.