

Predictive Parser

**Bachelor of Technology
Computer Science and Engineering**

Submitted By

ARKAPRATIM GHOSH (13000121058)

SEPTEMBER 2023



**Techno Main Salt Lake
EM-4/1, Sector-V,
Kolkata- 700091
West Bengal
India**

TABLE OF CONTENTS

1. Introduction.....	Error! Bookmark not defined.
2. Body	Error! Bookmark not defined.
2.1. Parsing and its Importance	4
2.2. Types of Parsing Techniques:	6
2.3. Components of Predictive Parser	7
2.4. Algorithm to construct Predictive Parsing Table	9
2.5. Error Handling and Resolution	13
2.6. Advantages Of Predictive Parsing.....	15
2.7. Disadvantages of Predictive Parsing.....	15
3. Conclusion	Error! Bookmark not defined.
4. References	Error! Bookmark not defined.

1. Introduction

Parsing is an essential process in the field of compiler design, facilitating the transformation of human-readable source code into structured data that can be analyzed and executed by computers. A parser is a fundamental component of a compiler that breaks down the input source code into a hierarchical structure, typically represented as a syntax tree or an abstract syntax tree. Among the various parsing techniques, predictive parsing stands out as an elegant and efficient top-down approach for understanding the syntactic structure of programming languages.

Parsing techniques can be broadly categorized into top-down and bottom-up parsing. Top-down parsing starts from the root of the syntax tree (the start symbol of the grammar) and proceeds to construct the tree by successively expanding non-terminal symbols into their corresponding terminal and non-terminal constituents. Predictive parsing is a specific variant of top-down parsing that employs a parsing table to predict the appropriate production rules based on the current non-terminal symbol and a limited lookahead of the input stream.

In the realm of predictive parsing, the LL(k) family of parsers holds prominence, where "LL" signifies Left-to-right scanning of the input and Leftmost derivation of grammar symbols, and "k" indicates the number of lookahead tokens used to make parsing decisions. The simplest and most widely used variant is the LL(1) parser, which employs a single-token lookahead to predict the production rules. This means that the parser looks at the next input token to determine the parsing action, making it efficient and straightforward to implement.

Predictive parsing involves the construction of a parsing table that guides the parser's decisions. This table is based on the First and Follow sets associated with the grammar symbols. The First set of a non-terminal consists of terminals that can start the strings derived from that non-terminal. The Follow set of a non-terminal comprises terminals that can appear immediately after the non-terminal in valid derivations. By using these sets, the parsing table indicates which production to select for a given non-terminal and lookahead token, allowing the parser to predict the next steps accurately.

The process of constructing a predictive parsing table involves meticulously calculating the First and Follow sets for each grammar symbol and filling in the table entries with the corresponding production rules. While this approach offers simplicity and ease of implementation, it is essential to acknowledge its limitations. Predictive parsing is limited to LL(1) grammars, meaning that the grammar must be unambiguous and possess a unique predictive parsing solution. Moreover, as the grammar becomes more complex, the parsing table can grow significantly, potentially leading to inefficiency in memory usage.

Predictive parsers represent a significant advancement in the field of compiler design and parsing techniques. Their top-down approach, reliance on lookahead tokens, and efficient parsing table construction make them valuable tools for generating syntax trees from source

code. By understanding the core concepts of predictive parsing, including LL(1) grammars, First and Follow sets, and parsing table construction, one gains insight into the inner workings of compilers and their role in translating high-level programming languages into machine-executable code. In the subsequent sections of this report, we will delve deeper into the mechanics of predictive parsing, exploring its various components, construction methodologies, and real-world applications.

2. Body

2.1. Parsing and its Importance

Parsing serves as a pivotal process within the realm of compiler design, facilitating the translation of human-readable source code into a structured representation that computers can comprehend and execute. At its core, parsing involves analyzing the syntax of a programming language to establish the relationships between various components of the code. This step is indispensable as it forms the foundation for subsequent stages of compilation, including semantic analysis, optimization, and code generation.

In the context of compilers, source code is often written by programmers using high-level languages, which are designed to be more intuitive and human-friendly than machine code. These high-level languages offer a higher level of abstraction, enabling programmers to focus on solving complex problems without delving into low-level details. However, this abstraction comes at the cost of requiring a translation process before the code can be executed on a computer's hardware.

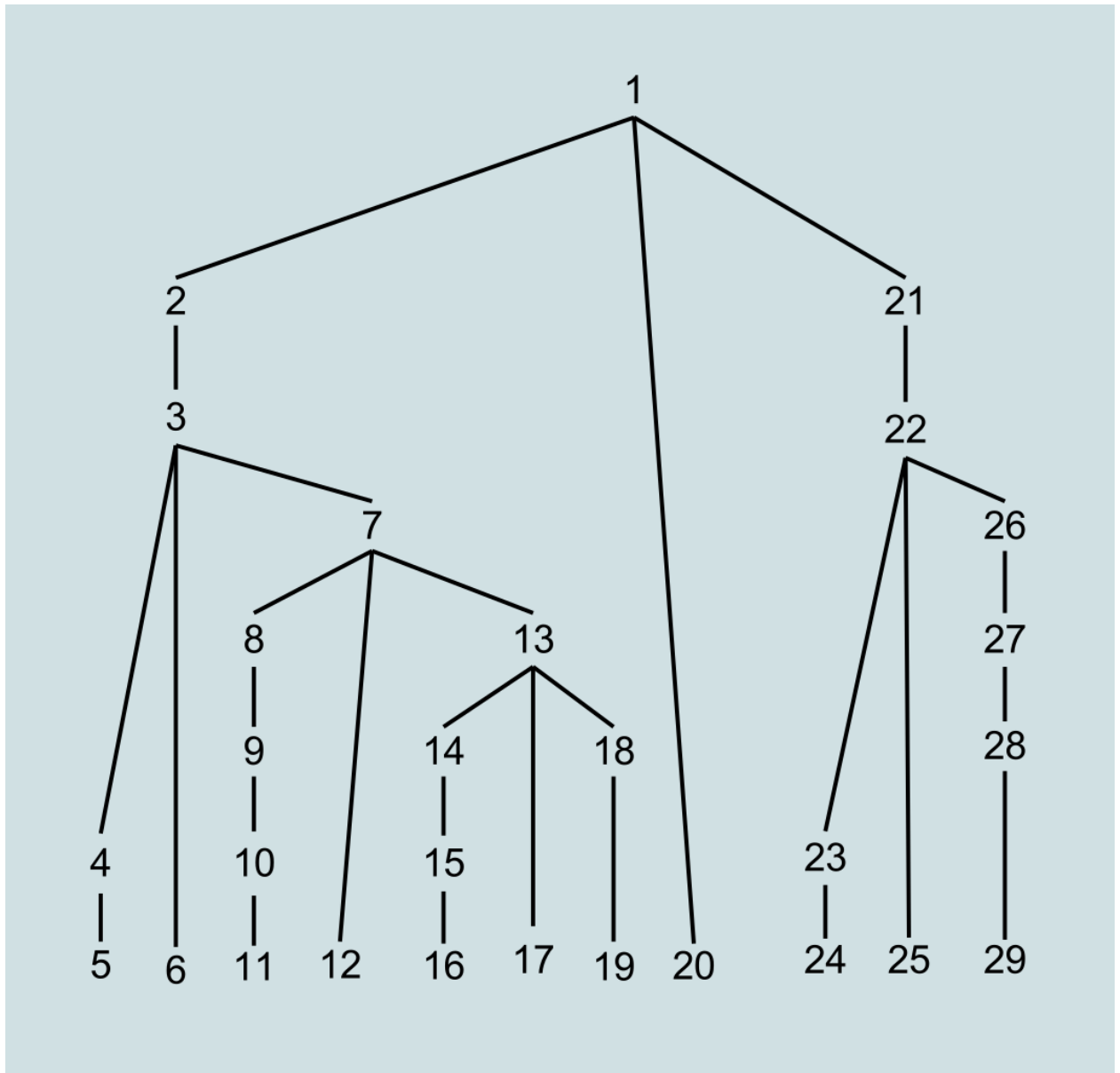
The primary purpose of parsing is twofold: error detection and syntax analysis. During the parsing phase, the compiler examines the code for adherence to the language's specified syntax rules. Any violations of these rules are flagged as syntax errors, making it easier for programmers to identify and rectify mistakes early in the development cycle. By catching these errors at an early stage, parsing contributes to the production of more robust and reliable software.

Furthermore, parsing plays a crucial role in generating a hierarchical representation of the source code's structure. This representation is typically a syntax tree or an abstract syntax tree (AST). These trees break down the code into smaller components, such as statements, expressions, and declarations, while preserving the relationships between them. The syntax tree captures the essence of the code's structure, allowing subsequent compiler phases to analyze and optimize the program's behavior.

As programming languages become more sophisticated and diverse, the need for accurate parsing becomes even more pronounced. Different languages have their own distinct syntax rules and grammar specifications. A well-implemented parser must be capable of

understanding the nuances of each language, ensuring that the code is transformed accurately and efficiently.

In the broader context of software development, parsing extends beyond compilers. Text editors, integrated development environments (IDEs), and various tools rely on parsing to provide features such as syntax highlighting, code completion, and error checking. This parsing-driven functionality enhances the programming experience by enabling developers to write code more effectively and with fewer mistakes.



2.2.Types of Parsing Techniques:

Parsing techniques are essential components of the compilation process, responsible for breaking down human-readable source code into a structured representation that can be understood and processed by computers. These techniques can be broadly categorized into two main groups: top-down parsing and bottom-up parsing. Each category approaches parsing from a distinct perspective, offering a range of advantages and trade-offs.

Top Down parsing

Top-down parsing begins with the start symbol of the grammar and attempts to construct the syntax tree by successively expanding non-terminal symbols until the entire input is parsed. This approach simulates the derivation of the input string from the start symbol and follows a top-to-bottom traversal of the syntax tree.

Recursive Descent Parsing:

One of the most prominent top-down parsing techniques is recursive descent parsing. In recursive descent parsing, the parser employs a set of recursive procedures or functions to match grammar rules. Each non-terminal in the grammar corresponds to a procedure, and the parser selects the appropriate procedure based on the current non-terminal symbol.

Predictive Parsing:

Predictive parsing is a specific type of top-down parsing that utilizes a predictive parsing table to determine the production rule to apply at each step. This table is constructed based on the First and Follow sets of grammar symbols and provides a predictive approach to parsing. Predictive parsing is especially efficient when implemented as LL(k) parsers, where "LL" stands for Left-to-right scanning and Leftmost derivation, and "k" represents the number of lookahead tokens used for making parsing decisions.

Bottom-Up Parsing:

Bottom-up parsing takes the opposite approach, starting with the input symbols and attempting to construct the syntax tree by reducing the input to the start symbol. This approach simulates a reverse derivation of the input string.

Shift-Reduce Parsing:

Shift-reduce parsing is a common bottom-up technique where the parser shifts input symbols onto a stack until a production rule can be reduced. Once the parser identifies a portion of the input that matches the right-hand side of a production rule, it replaces that portion with the corresponding non-terminal and proceeds to the next step.

LR Parsing:

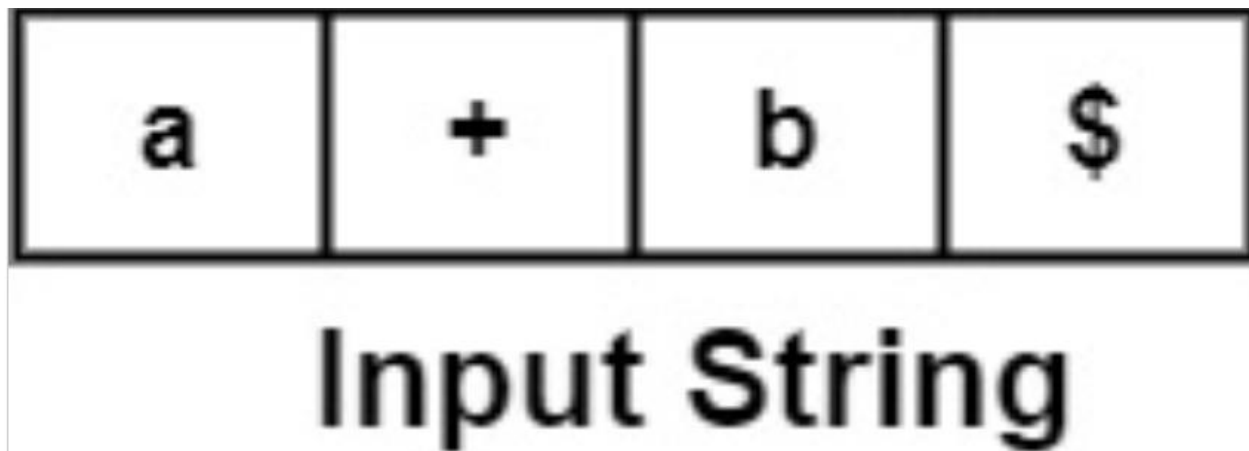
LR parsing is a family of bottom-up techniques that are highly powerful and widely used. LR parsers use a table-driven approach to handle different parsing actions, including shifting and reducing. LR parsers are more capable than LL parsers in handling a broader range of grammars, including left-recursive ones.

LALR Parsing:

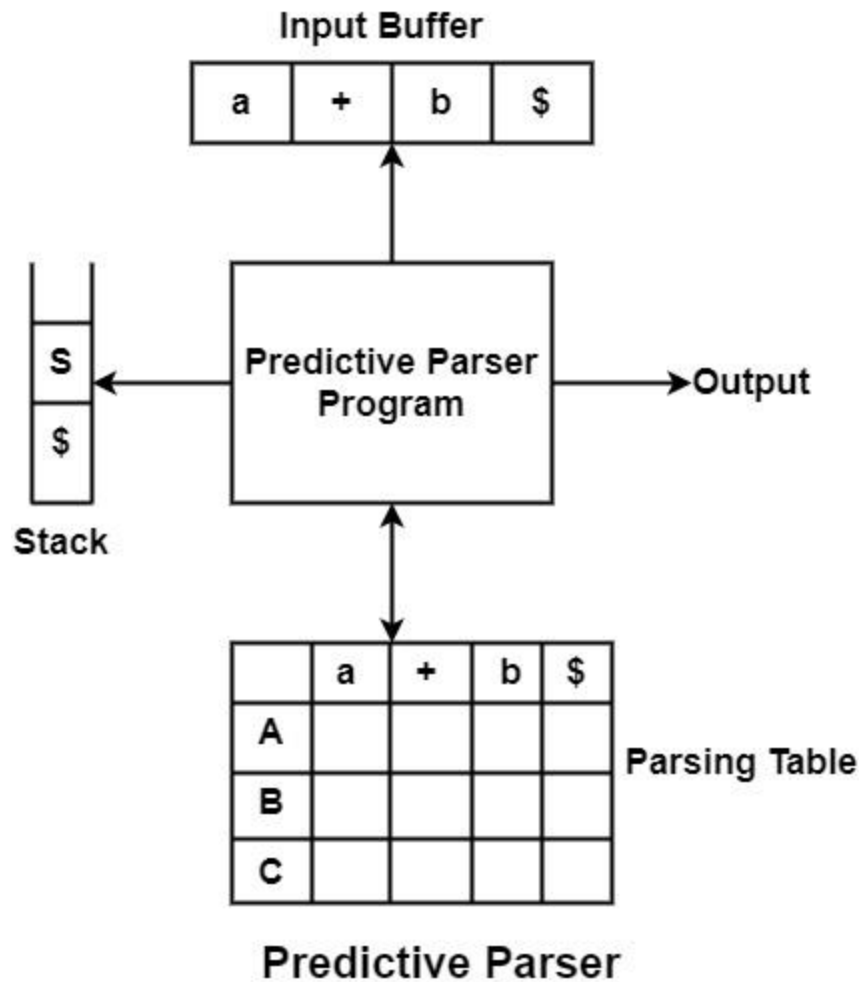
Look-Ahead LR (LALR) parsing is a variant of LR parsing that aims to reduce the complexity of LR parsing tables while maintaining the ability to handle a significant subset of context-free grammars.

2.3.Components of Predictive Parser

Input Buffer - The input buffer includes the string to be parsed followed by an end marker \$ to denote the end of the string.



Stack - It contains a combination of grammar symbols with \$ on the bottom of the stack. At the start of Parsing, the stack contains the start symbol of Grammar followed by \$.

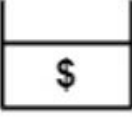
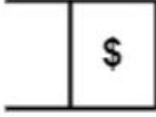

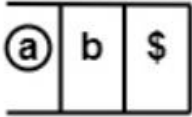
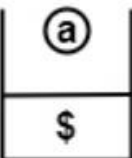
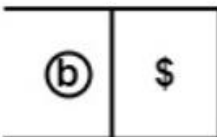

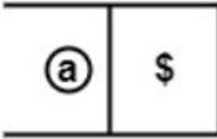


Parsing Table - It is a two-dimensional array or Matrix $M[A, a]$ where A is nonterminal and ' a ' is a terminal symbol.

All the terminals are written column-wise, and all the Non-terminals are written rowwise.

Parsing Program - The parsing program performs some action by comparing the symbol on top of the stack and the current input symbol to be read on the input buffer.

Actions - Parsing program takes various actions depending upon the symbol on the top of the stack and the current input symbol. Various Actions taken are given below -

Description	Top of Stack	Current Input Symbol	Action
1. If stack is empty, i.e., it only contains \$ and current input symbol is also \$.			Parsing will be successful and will be halted.
2. If symbol at top of stack and the current input symbol to be read are both terminals and are same.			Pop a from stack & advance to next input symbol.
3. If both top of stack & current input symbol are terminals and top of stack \neq current input symbol e.g. $a \neq b$.			Error
4. If top of stack is non-terminal & input symbol is terminal.			Refer to entry $M[X, a]$ in Parsing Table. If $M[X, a] = X \rightarrow ABC$ then Pop X from Stack Push C, B, A onto stack.

2.4. Algorithm to construct Predictive Parsing Table

Input – Context-Free Grammar G

Output – Predictive Parsing Table M

Method – For the production $A \rightarrow \alpha$ of Grammar G.

- For each terminal, a in FIRST (α) add $A \rightarrow \alpha$ to $M[A, a]$.
- If ϵ is in FIRST (α), and b is in FOLLOW (A), then add $A \rightarrow \alpha$ to $M[A, b]$.
- If ϵ is in FIRST (α), and \$ is in FOLLOW (A), then add $A \rightarrow \alpha$ to $M[A, \$]$.
- All remaining entries in Table M are errors.

		Terminal Symbols			
		a	b	+	\$
Non-Terminals Symbols	A				
	B		←		
	C				
	D				

M [B, b]

M[C, +]

Following are the steps to perform Predictive Parsing

- Elimination of Left Recursion
- Left Factoring
- Computation of FIRST & FOLLOW
- Construction of Predictive Parsing Table
- Parse the Input String

Consider the following grammar –

- $E \rightarrow E + T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow (E) \mid id$

After removing left recursion, left factoring

- $E \rightarrow TT'$
- $T' \rightarrow +TT' \mid \epsilon$
- $T \rightarrow FT''$
- $T'' \rightarrow *FT'' \mid \epsilon$
- $F \rightarrow (E) \mid id$

STEP

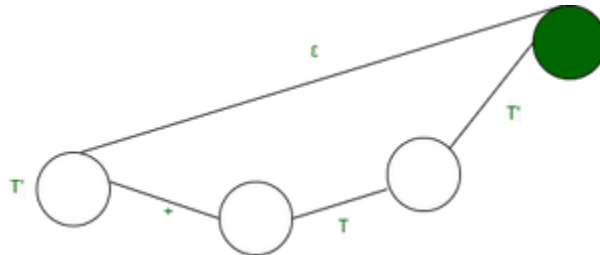
1:

Make a transition diagram(DFA/NFA) for every rule of grammar.

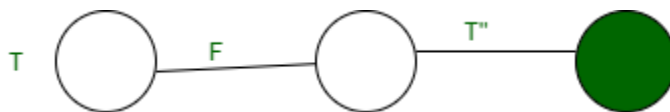
- $E \rightarrow TT'$



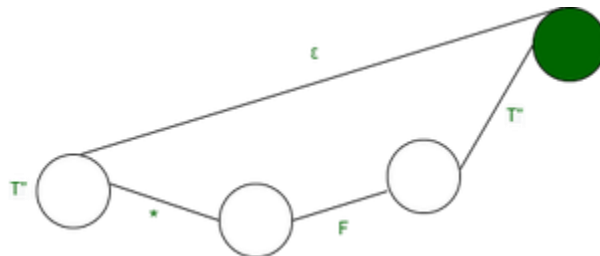
- $T' \rightarrow +TT' | \epsilon$



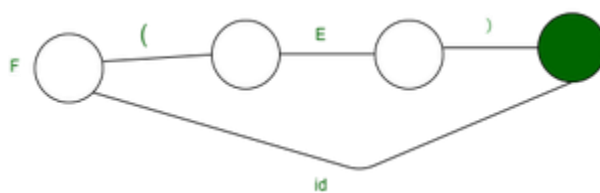
- $T \rightarrow FT''$



- $T'' \rightarrow *FT'' | \epsilon$



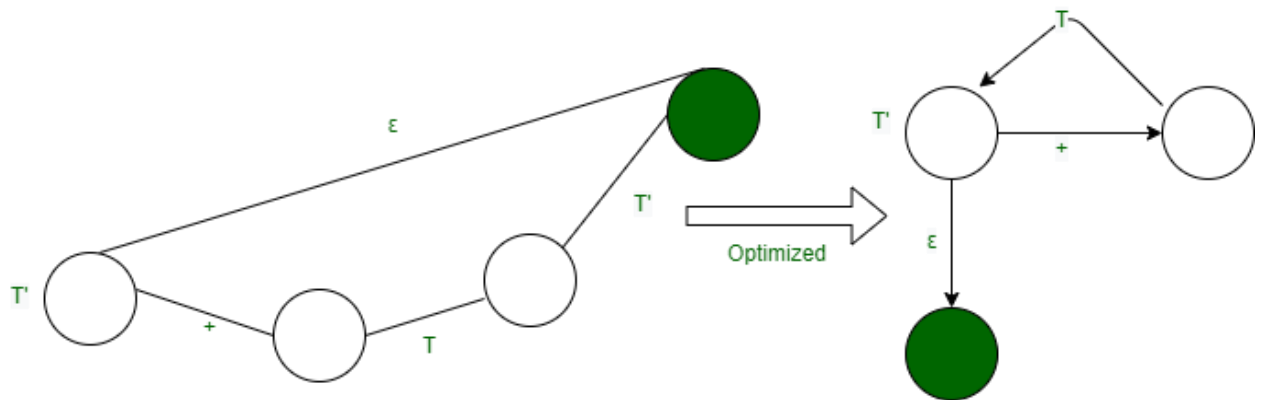
- $F \rightarrow (E) | id$



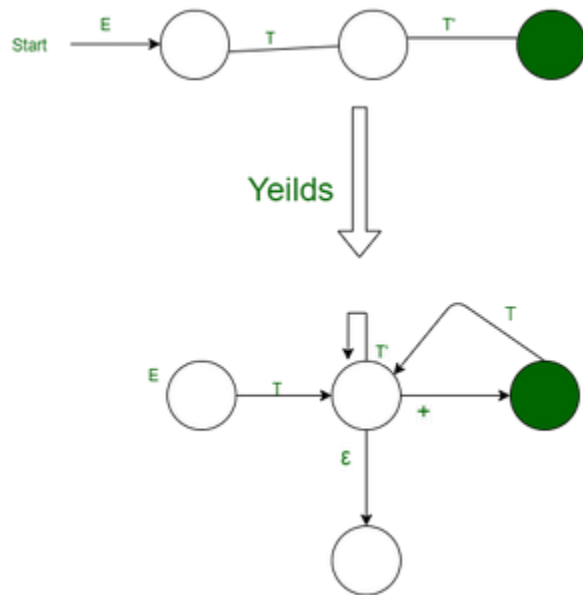
STEP 2:

Optimize the DFA by decreases the number of states, yielding the final transition diagram.

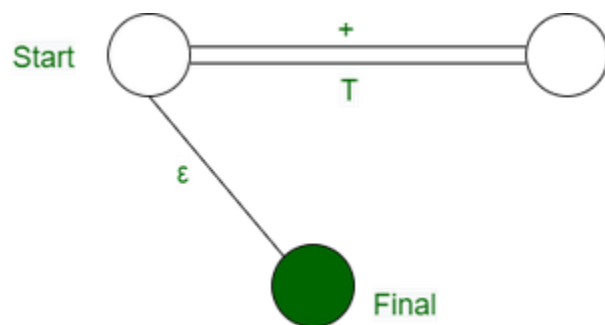
- $T' \rightarrow +TT' | \epsilon$

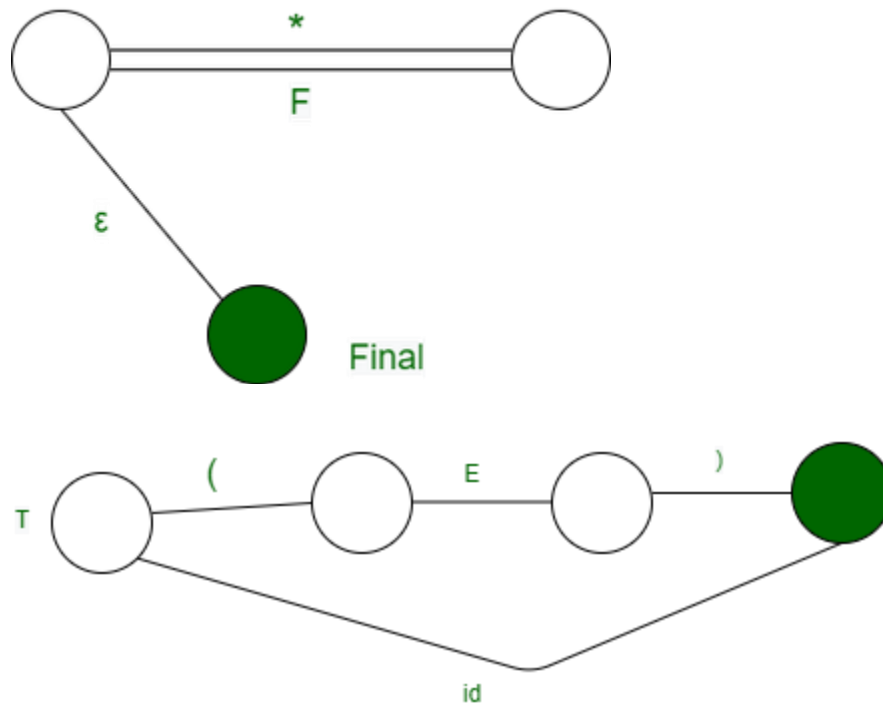


It can be optimized ahead by combining it with DFA for $E \rightarrow TT'$



Accordingly, we optimize the other structures to produce the following DFA





STEP 3:

Simulation on the input string.

Steps involved in the simulation procedure are:

1. Start from the starting state.
2. If a terminal arrives consume it, move to the next state.
3. If a non-terminal arrives go to the state of the DFA of the non-terminal and return on reached up to the final state.
4. Return to actual DFA and Keep doing parsing.
5. If one completes reading the input string completely, you reach a final state, and the string is successfully parsed.

2.5. Error Handling and Resolution

While predictive parsing is a powerful and efficient technique for analyzing the syntactic structure of programming languages, it is not immune to encountering errors or conflicts. Errors can arise due to syntax violations in the input code, and conflicts can occur when the parsing table is ambiguous or lacks clear instructions for the parser to follow. Effective error handling and conflict resolution mechanisms are crucial to ensuring accurate and reliable parsing results.

Error Handling in Predictive Parsing:

Syntax Errors: Predictive parsers are designed to detect syntax errors early in the parsing process. When a parsing decision cannot be made due to a mismatch between the current non-terminal and the lookahead token, the parser identifies a syntax error and reports it to the user. This aids programmers in locating and correcting mistakes in their code promptly.

Error Recovery: After encountering an error, the parser can employ error recovery strategies to continue parsing and identify subsequent errors. Common strategies include skipping tokens until a known synchronization point is reached or inserting missing tokens to help the parser regain its context.

Error Reporting: Effective error messages that provide information about the nature and location of errors greatly assist programmers in debugging their code. Error messages should be informative and user-friendly, helping developers understand the issue and resolve it efficiently.

Conflict Resolution in Predictive Parsing:

Ambiguities in the Grammar: Sometimes, a grammar can be inherently ambiguous, leading to multiple possible parsing paths. This ambiguity can manifest as conflicts in the predictive parsing table, where the parser is uncertain about which production rule to choose for a specific input symbol.

First/Follow Set Conflicts: Conflicts can occur when multiple production rules share a common prefix in their First sets. In such cases, the parser cannot predict which rule to apply based on the current input token and lookahead.

LL(1) Ambiguity: Predictive parsers operate under the constraint of LL(1) grammars, where ambiguity in the grammar can cause difficulties. Ambiguity can arise due to left recursion or overlapping First and Follow sets.

Conflict Resolution Strategies:

Precedence and Associativity: Assigning precedence and associativity rules to operators in the grammar helps resolve conflicts related to operator ambiguity. This ensures that the parser correctly interprets expressions involving multiple operators.

Factorization and Restructuring: Restructuring the grammar by factoring out common prefixes or using intermediate non-terminals can help eliminate conflicts. By modifying the grammar to be more LL(1)-friendly, the predictive parsing table becomes easier to construct.

Explicit Disambiguation: Introducing additional syntax or rules to explicitly specify the desired interpretation in cases of ambiguity can help the parser make unambiguous decisions.

Extended Lookahead: In some cases, extending the lookahead beyond a single token (e.g., LL(2) or LL(3)) can provide the parser with more context, enabling it to resolve certain conflicts.

2.6. Advantages Of Predictive Parsing

- Predictive parsers operate with a single-token lookahead, which allows them to predict the next parsing action accurately. This lookahead-driven approach minimizes backtracking and leads to efficient parsing.
- Predictive parsers operate with a single-token lookahead, which allows them to predict the next parsing action accurately. This lookahead-driven approach minimizes backtracking and leads to efficient parsing.
- The structured nature of predictive parsing enables the generation of informative error messages. These messages help programmers understand the nature and location of syntax errors, making debugging easier.
- The structured nature of predictive parsing enables the generation of informative error messages. These messages help programmers understand the nature and location of syntax errors, making debugging easier.

2.7. Disadvantages of Predictive Parsing

- Predictive parsers are limited to LL(1) grammars, which can restrict the class of grammars they can handle. LL(1) grammars should be unambiguous, and conflicts in the parsing table must be resolved.
- Ambiguous grammars can lead to conflicts in the parsing table. Resolving these conflicts requires restructuring the grammar or using explicit disambiguation rules, which might complicate the parser's implementation.
- The single-token lookahead of predictive parsers can be limiting in some cases. Complex parsing decisions might require more context than a single token can provide, leading to inaccuracies in parsing.
- The single-token lookahead of predictive parsers can be limiting in some cases. Complex parsing decisions might require more context than a single token can provide, leading to inaccuracies in parsing.
- Left-recursive grammars can pose challenges for predictive parsers. Left recursion can lead to infinite loops in parsing and necessitates the use of techniques like left-factoring to handle them.
- In some cases, modifying the grammar to adhere to the LL(1) constraint or to eliminate conflicts can require significant changes. This modification might lead to a departure from the original intuitive representation of the language.

3. Conclusion

Predictive parsing stands as a fundamental technique in the landscape of compiler design, offering a structured and efficient approach to understanding the syntactic structure of programming languages. Throughout this report, we have explored the intricacies of predictive parsing, delving into its core principles, advantages, challenges, and applications.

Predictive parsing stands as a fundamental technique in the landscape of compiler design, offering a structured and efficient approach to understanding the syntactic structure of programming languages. Throughout this report, we have explored the intricacies of predictive parsing, delving into its core principles, advantages, challenges, and applications.

The importance of predictive parsing extends beyond its role in compilers. Integrated development environments (IDEs) and code editors employ parsing techniques to provide features like syntax highlighting, autocompletion, and error checking, thus improving the programming experience. Furthermore, understanding predictive parsing contributes to a deeper comprehension of language processing, formal grammars, and the principles underlying programming language design.

The importance of predictive parsing extends beyond its role in compilers. Integrated development environments (IDEs) and code editors employ parsing techniques to provide features like syntax highlighting, autocompletion, and error checking, thus improving the programming experience. Furthermore, understanding predictive parsing contributes to a deeper comprehension of language processing, formal grammars, and the principles underlying programming language design.

In the broader context of compiler design, predictive parsing represents a pivotal step in translating high-level programming languages into machine-executable code. It underscores the intricate interplay between theory and practice in the realm of software development, exemplifying the need to balance theoretical concepts with practical implementations.

As the field of programming languages continues to evolve, predictive parsing remains a valuable tool in the arsenal of techniques for analyzing and processing code. Its strengths in efficiency, simplicity, and error detection contribute to the robustness of compilers and language processing tools. By mastering the concepts and nuances of predictive parsing, software developers and compiler engineers are equipped with essential skills to navigate the intricate world of language translation and software development.

4. References

- [What is a Predictive Parser \(tutorialspoint.com\)](https://www.tutorialspoint.com/compiler-design/predictive-parsing.htm)
- [Predictive Parser in Compiler Design - GeeksforGeeks](https://www.geeksforgeeks.org/predictive-parsing-in-compiler-design/)