IMPLEMENTATION OF LL(1) PARSER

Aditya Rajesh Sakri Vaka Satwik Reddy Vamsi Krishna Vunnam  
BL.EN.U4AIE20001 BL.EN.U4AIE20070 BL.EN.U4AIE20075

**ABSTRACT :**

An LL1 parser is a type of top-down parser used in computer science to analyse and validate code syntax. This parser uses a predictive parsing technique, where the parser predicts the next token in the input stream and matches it with the expected token. Implementing an LL1 parser involves adding Grammar Augmentation, Removing Left Recursion, Removing Left Factoring, Calculating First and Follow sets and constructing a parse table and using it to parse the input. The LL(1) parser implementation utilizes stack and buffer data structures to validate input strings. In this parsing technique we start parsing from the top (start symbol of parse tree) to down (the leaf node of parse tree) in a top-down manner.

**INTRODUCTION :**

A LL(1) parser is a type of top-down parser commonly used in compiler construction and syntax analysis. It is based on the LL(1) parsing algorithm, where "LL" stands for "Left-to-right, Leftmost derivation," and the number "1" indicates that the parser uses a lookahead of one token.

The LL(1) parsing algorithm operates by building a parse tree from an input string by applying a set of production rules defined by a grammar. It starts from the root symbol of the grammar and repeatedly expands non-terminals to match the input tokens. The lookahead token is used to determine which production rule to apply at each step. The LL(1) parsing algorithm has certain restrictions and requirements. To be LL(1) parsable, a grammar must be unambiguous and satisfy the LL(1) grammar properties. These properties include having distinct lookahead sets for each alternative of a non-terminal and ensuring that there are no first/follow set conflicts.

Recursive descent parsing involves defining recursive procedures for each non-terminal symbol, while table-driven parsing involves constructing parsing tables and using them to guide the parsing process efficiently and provide a foundational understanding of parsing techniques and form the basis for more advanced parsing algorithms. They are widely used in compiler construction, programming language analysis, and other areas where efficient and accurate parsing is required.

**METHODOLOGY :**

**A picture containing text, screenshot, font, diagram

Description automatically generated**

Fig 1 : Flowchart / methodology of LL1 parsing implementation.

**IMPLEMENATION :**

Based on the above methodology the first step is Augmenting the given grammar involves introducing a new start symbol and a new production rule. This step ensures that there is a unique starting point for the parsing process. Removing left recursion is crucial to prevent infinite recursion during parsing. It entails identifying and handling non-terminals that directly or indirectly produce themselves. By replacing these left-recursive rules with equivalent right-recursive ones, we ensure progress in the parsing process. Applying left factoring helps eliminate common prefixes in the grammar. This step involves creating new non-terminals and rewriting rules to factor out the shared prefixes. By doing so, we avoid ambiguity and enable efficient parsing decisions.

Computing the First and Follow sets for each non-terminal is essential for LL(1) parsing. The First set comprises the terminals that can initiate derivations from a non-terminal, while the Follow set contains the terminals that can appear after a non-terminal in a derivation. These sets guide the construction of the parsing table. Constructing the LL(1) parsing table involves creating a 2D table with non-terminals as rows and terminals as columns. Each table entry represents a production rule to apply when the non-terminal matches the current input token. The parsing table enables deterministic parsing decisions based on the current non-terminal and lookahead token.

Parsing the string using a stack entails an iterative process. The stack is initially populated with the augmented start symbol, and the input string is tokenized. The parsing algorithm compares the top of the stack with the current input token, applies the appropriate production rule from the parsing table, and updates the stack and input accordingly. This process continues until the stack is empty or an error is encountered, determining the validity of the input string.

**RESULTS :**

****

Fig 2: Results for sample grammar Testing

A screenshot of a computer

Description automatically generated with medium confidence

Fig3: GUI Using Stream lit.

**AKNOWLEDGEMENT :**

I would like to express my sincere gratitude to Dr. Meena Belwal, for supporting and helpful in the successful completion of my term project. Their assistance and encouragement were invaluable throughout this journey.

**CONCLUSION:**  
In conclusion, implementing an LL(1) parser using Python allows for efficient and accurate parsing of programming language syntax. The LL(1) parsing technique, based on predictive parsing, offers advantages such as simplicity, speed, and error detection. By following key steps such as grammar augmentation, left recursion removal, left factoring, and constructing the parsing table, we can successfully parse input strings. The implementation showcased here demonstrates the process of validating and parsing strings using a stack and buffer approach. The LL(1) parser implementation in Python provides a solid foundation for building robust compilers and language processors.

**REFERENCES :**

1. Alfred V. Aho, Ravi Sethi, and Jerey D. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1986.
2. Michael Sipser. Introduction to the Theory of Computation. PWS Publishing, 1997.
3. A. V. Aho and J. D. Ullman, "Principles of Compiler Design," Addison-Wesley, 1977.
4. S. Chattopadhyay, "Compiler Design," CRC Press, 2009.