**Team 07**

|  |  |
| --- | --- |
| **Gogineni Ashrith Sai** | **BL.EN.U4AIE20015** |
| **Konda Reddy Balaji Reddy** | **BL.EN.U4AIE20029** |
| **Thanakanti Ganesh Madhav** | **BL.EN.U4AIE20067** |

**Implementing Parser and Lexer for a simple C-like Language using the C-tools Flex and Bison and self-written Custom Structures and Code**

**Introduction:** This project aims to create parser and lexer for a simple programming language similar to C. In this project, we have used two popular tools called Flex and Bison. Flex helps in creating the part of the compiler that analyzes the source code's structure, while Bison helps in creating the part that understands the grammar rules and builds a parse tree.

First, it will analyze the code to identify and categorize its individual components, such as keywords, operators, and variables. This process is called lexical analysis. Then, it will use the grammar rules defined in the language to parse the code.

**Methodology:**

* **Lexical Analysis:**

The input source code is processed by the lexer generated with Flex.

The lexer scans the input character by character and groups them into meaningful tokens or lexical units.

Tokens can represent identifiers, keywords, operators, literals, symbols, or other language-specific constructs.

Lexical analysis ensures that the input is split into individual tokens for further processing.

* **Syntax Analysis or Parsing:**

The token stream produced by the lexer is then processed by the parser generated with Bison.

The parser applies grammar rules defined in a context-free grammar to recognize the structure of the input code.

It constructs a syntax tree, representing the hierarchical structure of the code based on the grammar rules.

The syntax tree reflects the relationship between different language constructs, such as expressions, statements, and declarations.

* **Semantic Analysis:**

Once the syntax tree is constructed, semantic analysis is performed to check the correctness and meaning of the code.

Semantic analysis involves verifying language-specific rules and performing various checks.

Type checking is carried out to ensure that expressions and operations are compatible.

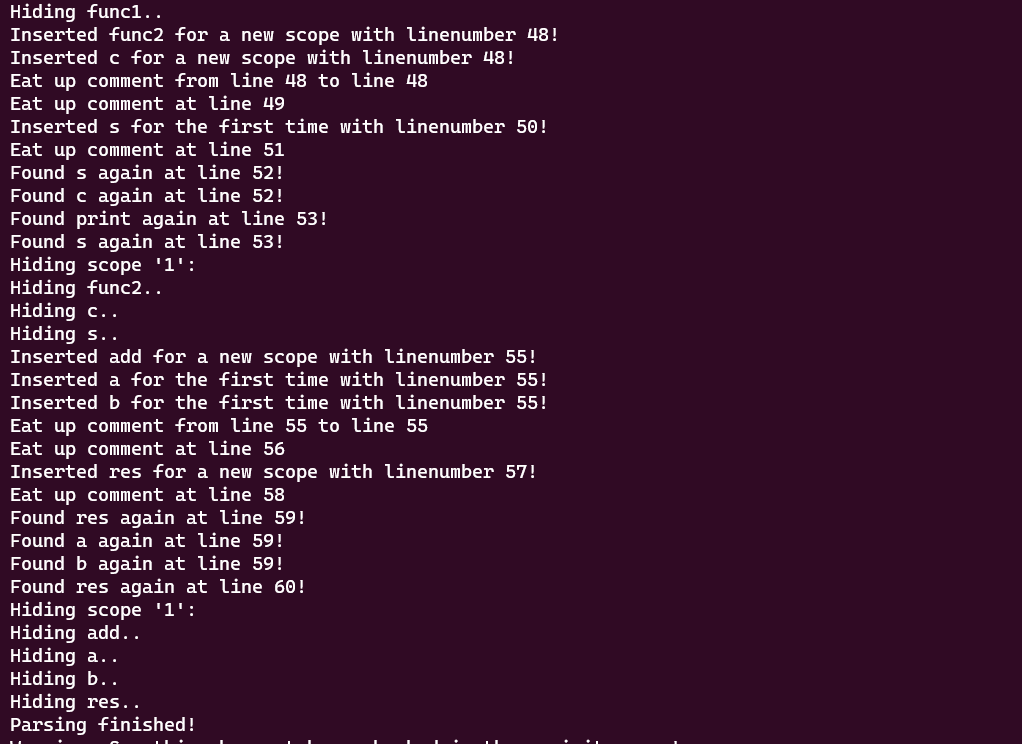
Variable scoping rules are enforced, ensuring that variables are declared before use and within the appropriate scope.

Other semantic validations, such as checking function call arguments or enforcing language-specific constraints, are performed.

Semantic analysis aims to identify and report any errors or inconsistencies in the code.

**A screenshot of a computer

Description automatically generated Results:**

****

**Lexical Analysis Semantic Analysis**

**Conclusion:**

In summary, during the development of our compiler, we focused on key aspects such as type declaration, type checking, integration of components, and the implementation of a revisit queue. The revisit queue concept allowed us to handle function calls before their declaration, addressing challenges related to unknown return types and parameter compatibility. By inserting undeclared variables into the queue and generating appropriate warnings, we ensured the accurate identification of undeclared variables. Through continuous improvement and refinement, our compiler has become more robust and capable of efficiently analyzing and processing source code in a reliable manner.

**References:**

**Books**

•"Flex and Bison" by John Levine. This book provides an in-depth introduction to using Flex and Bison for compiler development, and it includes examples and practical advice.

•"Writing a Compiler in Go" by Thorsten Ball: This is a tutorial series that teaches how to build a compiler using the Go programming language. It provides a step-by-step guide to implementing a lexer, parser, and code generator, and it also covers more advanced topics like optimization.

•"The LLVM Compiler Infrastructure Project": LLVM is a popular open-source compiler infrastructure that provides tools for building compilers, code generators, and other related tools.

**Project References**

•"Tiny C Compiler": This is a small but self-contained C compiler that is designed to be easy to understand and modify. The source code is available on GitHub: https://github.com/TinyCC/tinycc.

•"C4 Compiler": This is a C compiler that is designed to be small, fast, and easy to use. It uses a custom intermediate representation (IR) and includes features like inline assembly and register allocation. The source code is available on GitHub: https://github.com/rswier/c4.

•"Nim Compiler": This is a compiler for the Nim programming language, which is designed to be high-level and expressive while also producing efficient code. The source code is available on GitHub: https://github.com/nim-lang/Nim