**Name:** Muhammad Alyaan & Ayesha Anwaar

**Registration Number:** FA20-BCS-048 & FA20-BCS-018

**Date:** 27th December 2023

**Sec:** B

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Compiler Construction Lab Terminal**

1. **Task 1: (Brief of Code)**

**Overview**

This code represents a simple compiler front-end for a C++-like programming language. It includes a lexical analyzer to tokenize source code, a parser to build a syntax tree, and semantic analysis to catch errors in the code. The language supports variable declarations, assignments, arithmetic operations, conditional statements (if-else), and function declarations.

**Lexical Analysis**

The code begins with lexical analysis, where it tokenizes the provided C++-like source code into specific categories, such as data types (`DATATYPE`), identifiers (`IDENTIFIER`), operators (`OPERATOR`), integers (`INTEGER`), and end-of-statements (`END\_STATEMENT`). The lexical analyzer also checks for errors, such as redeclarations of variables.

**Symbol Table**

A symbol table (`symbol\_table`) is maintained to store variable information, including data type and value. This table is utilized during semantic analysis to ensure proper variable usage.

**Grammar Rules**

The code defines extensive grammar rules for variable declarations, assignments, if-else statements, arithmetic expressions, and function declarations. These rules guide the parsing process and the construction of a syntax tree.

**First and Follow Sets**

The script calculates and prints the First and Follow sets for the defined grammar rules. These sets play a crucial role in parsing and syntax tree construction.

**Syntax Tree Construction**

The syntax tree is constructed based on the parsed tokens and grammar rules. The tree visually represents the hierarchical structure of the source code.

**Semantic Analysis**

Semantic analysis is performed on the syntax tree to catch errors, such as incompatible data types in arithmetic operations. Detected errors are printed with appropriate messages.

**Output**

The script outputs the tokens, First sets, Follow sets, syntax tree, and semantic analysis tree for the provided C++-like source code.

**Conclusion**

This code serves as a basic foundation for a compiler front-end, demonstrating key concepts such as lexical analysis, parsing, and semantic analysis. It can be used as a starting point for more advanced compiler development and serves as an educational tool for understanding compiler design principles.

1. **Task 2: (Control Flow)**

**`** **Sure, let's explain two functions from the provided code:**

**`calculate\_first\_sets(grammar\_rules)`**

**Description:**

This function calculates the First sets for each non-terminal in the given context-free grammar rules. The First set of a non-terminal is the set of terminals that can start the strings derivable from that non-terminal. The algorithm iterates through the grammar rules and computes the First sets based on the terminals and non-terminals encountered in the rules.

**Parameters:**

- `grammar\_rules`: A list of tuples representing the grammar rules.

**Return:**

- A dictionary (`first\_sets`) where keys are non-terminals, and values are sets of terminals representing their First sets.

**`calculate\_follow\_sets(grammar\_rules, first\_sets)`**

**Description:**

This function calculates the Follow sets for each non-terminal in the given context-free grammar rules, using the already computed First sets. The Follow set of a non-terminal is the set of terminals that can appear immediately to the right of it in some derivation. The algorithm iterates through the grammar rules and computes the Follow sets based on the First sets and other Follow sets.

**Parameters:**

- `grammar\_rules`: A list of tuples representing the grammar rules.

- `first\_sets`: A dictionary containing the precomputed First sets of non-terminals**.**

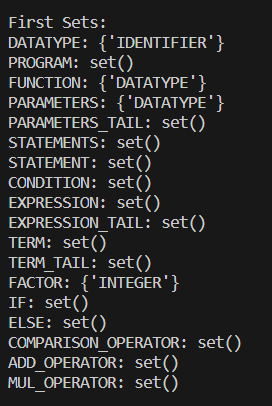
**Return:**

**-** A dictionary (`follow\_sets`) where keys are non-terminals, and values are sets of terminals representing their Follow sets.

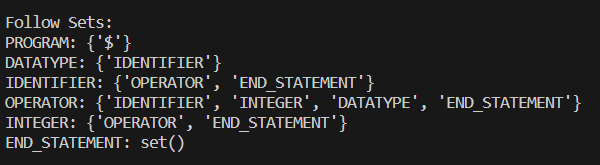
**Explanation:**

Both of these functions are crucial for building predictive parsers and ensuring the correctness of the parsing process. They are part of the broader task of constructing a parser that can analyze and interpret the source code based on the defined grammar rules. The First sets are used to determine the starting terminals for each non-terminal, while the Follow sets are used to guide the parsing process and identify valid termination points for parsing non-terminals.

The calculated First and Follow sets are then used in the parsing process to make decisions about which production rules to apply during the construction of the syntax tree. These sets are fundamental in ensuring that the parser can handle different syntactic constructs and make appropriate choices during parsing.

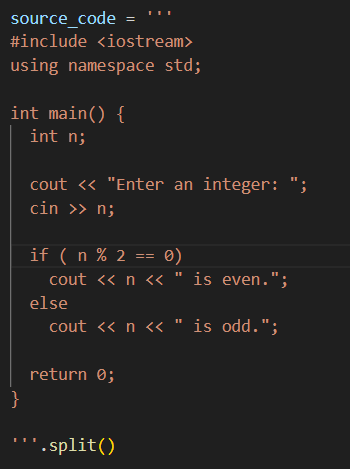
**Output (First Sets): **

**Output (Semantic Analysis):**

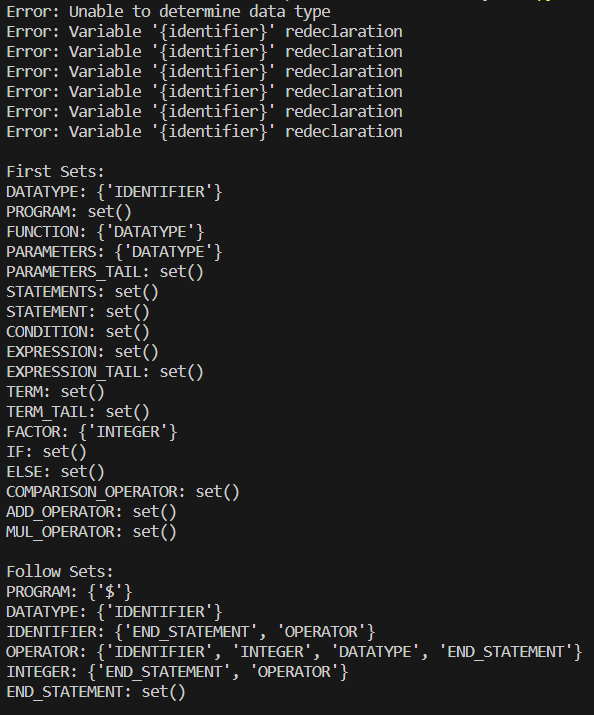
****

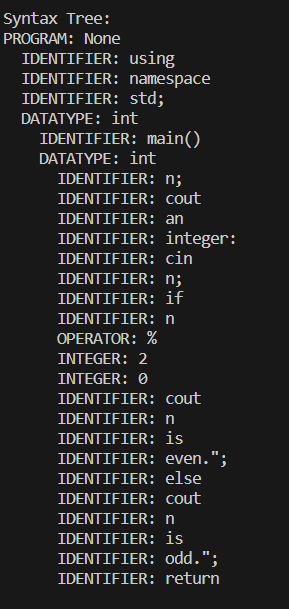
1. **Task 3: (Data Flow)**

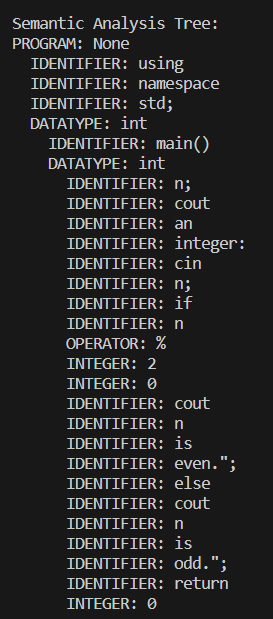
**Input:**

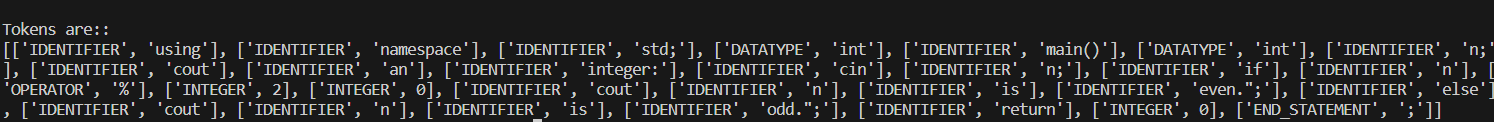


**Output:**

****

****

****

****

1. **Task 4: (Functions Working)**

**First Set Calculation:**

The first set of a non-terminal in a context-free grammar consists of the terminals that can begin the strings derivable from that non-terminal. The following functions and explanations describe how the first sets are calculated:

**`calculate\_first\_sets(grammar\_rules)`**

**1. Initialization:**

- Initialize an empty dictionary `first\_sets` to store the first sets for each non-terminal.

**2. Iterate Over Grammar Rules:**

- For each production rule in the grammar:

- Identify the non-terminal on the left-hand side (`non\_terminal`).

- Initialize an empty set `first\_set` for the non-terminal.

**3. Calculate First Set for the Right-Hand Side:**

- Iterate over symbols on the right-hand side of the production rule:

- If the symbol is a terminal, add it to the `first\_set` and break the loop.

- If the symbol is a non-terminal, add its first set to the `first\_set`.

- If the symbol's first set does not contain the empty string (EPSILON), break the loop.

**4. Assign First Set:**

- Assign the calculated `first\_set` to the `non\_terminal` in the `first\_sets` dictionary.

**5. Repeat:**

- Repeat these steps until the first sets stabilize.

**6. Return First Sets:**

**-** Return the `first\_sets` dictionary containing the calculated first sets.

**Follow Set Calculation:**

The follow set of a non-terminal in a context-free grammar represents the terminals that can appear immediately to the right of that non-terminal in some derivation. The following functions and explanations describe how the follow sets are calculated:

**`calculate\_follow\_sets(grammar\_rules, first\_sets)`**

**1. Initialization:**

- Initialize an empty dictionary `follow\_sets` to store the follow sets for each non-terminal.

- Add the end-of-input marker ('$') to the follow set of the start symbol.

**2. Iterate Over Grammar Rules:**

- For each production rule in the grammar:

- Iterate over symbols in the right-hand side of the production rule:

- If the symbol is a non-terminal:

- Calculate the first set of the sequence following the non-terminal (excluding epsilon).

- Add the calculated first set to the follow set of the non-terminal.

- If epsilon is in the first set, add the follow set of the left-hand side non-terminal.

**3. Repeat Until Stability:**

**-** Repeat the process of updating follow sets until no more changes occur.

**4. Return Follow Sets:**

**-** Return the `follow\_sets` dictionary containing the calculated follow sets.

**Link Between First and Follow Sets:**

**1.** Dependency on First Sets:

- The `calculate\_follow\_sets` function relies on the `calculate\_first\_sets` function to compute the first sets of non-terminals.

**2. Usage in Follow Set Calculation:**

- In the `calculate\_follow\_sets` function, when calculating the first set of a sequence, it calls the `calculate\_first\_of\_sequence` function, which uses the `first\_sets` calculated earlier.

**3. Iterative Process:**

**-** The first sets are initially calculated, and then they are used to calculate the follow sets.

- The process is iterative, with both sets being updated until stability is reached.

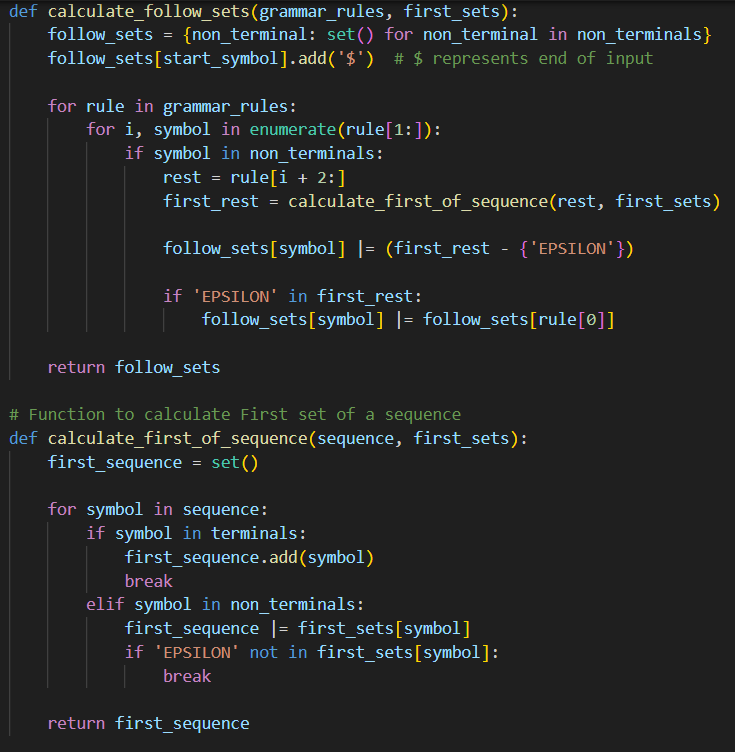
**4. Link in Algorithm Flow:**

- The calculated first sets provide information about the terminals that can start a production, influencing how follow sets are determined.

**5. Overall Dependency:**

- The link between the two sets is established through their dependency on each other in the context of the grammar rules and their symbols.

**First and Follow set functions:**

****

1. **Task 5: (Challenges faced during code)**

Writing the provided code involves dealing with various challenges, and here are some potential challenges that may arise:

1. **Parsing Logic:**

- Constructing a syntax tree involves correctly parsing the source code based on the given grammar rules. Any error in parsing logic can lead to an incorrect syntax tree.

2. **Error Handling:**

- The code contains some error-checking mechanisms, such as checking for variable redeclaration or invalid numeric values. Ensuring comprehensive and accurate error handling for various scenarios can be challenging.

3. **Semantic Analysis:**

- Performing semantic analysis, such as checking for incompatible data types in expressions, requires a clear understanding of the programming language's semantics. Handling various scenarios of semantic errors can be complex.

4. **Symbol Table Management:**

- Managing the symbol table involves tracking variable information, including data type and value. Ensuring the symbol table is updated correctly and consistently throughout the code is crucial.

5. **Grammar Rule Definitions:**

- The grammar rules defined in `grammar\_rules` must accurately represent the syntax of the programming language. Any inconsistencies or omissions in the grammar rules can lead to incorrect parsing and analysis.

6. **Tokenization:**

- Tokenizing the source code involves correctly identifying and categorizing each token. Regular expressions are used for tokenization, and ensuring that tokens are identified accurately is crucial for correct parsing.

7. **Algorithm Correctness:**

- The correctness of the algorithms used for calculating first and follow sets, as well as building the syntax tree, is vital. Any logical errors in these algorithms can lead to incorrect results.

8. **Code Readability and Maintainability:**

- The code's readability and maintainability are essential for long-term development and debugging. Ensuring clear and concise code with appropriate comments can help in understanding and maintaining the code.

9. **Handling Large Codebases:**

- The provided code is a simplified example. Handling larger codebases and more complex language features would require additional considerations for scalability and efficiency.

10. **Testing:**

- Comprehensive testing is crucial to ensure that the code works correctly for various input scenarios. Writing thorough test cases for different parts of the code is a significant challenge.

11. **Understanding the Grammar:**

- Developing and maintaining a parser requires a deep understanding of the grammar of the programming language. Ensuring that the grammar accurately represents the language's syntax is critical.

12. **Integration with Other Components:**

- In a real-world scenario, this code might be part of a larger compiler or interpreter. Integrating it with other components and ensuring proper communication between them can be challenging.

Addressing these challenges involves careful consideration of the language's syntax and semantics, thorough testing, and continuous improvement based on feedback and experience. Additionally, building language processing tools often involves a combination of theoretical understanding and practical implementation skills.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**