**Name:** Ayesha Malik & Yusra Arshad Nawaz

**Registration Number:** FA20-BCS-029 & FA20-BCS-054

**Date:** 27th December 2023

**Sec:** B

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

**Compiler Construction Lab Terminal**

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

**Overview**

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

**Lexical Analysis**

The code begins with lexical analysis, where it splits the source code into tokens. Tokens include data types (`DATATYPE`), identifiers (`IDENTIFIER`), operators (`OPERATOR`), integers (`INTEGER`), and end-of-statements (`END\_STATEMENT`). The lexical analyzer also detects and handles errors, such as invalid numeric values.

**Symbol Table**

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

**Grammar Rules**

The code defines grammar rules for variable declarations, assignments, if-else statements, arithmetic expressions, and function declarations. These rules are later used to build a syntax tree.

**First and Follow Sets**

The script calculates and prints the First and Follow sets for the grammar rules. These sets are crucial for parsing and building the syntax tree.

**Syntax Tree Construction**

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

**Semantic Analysis**

Semantic analysis is performed on the syntax tree to catch errors such as variable re-declarations and incompatible data types in arithmetic operations. The semantic analysis tree is printed to display the results.

**Output**

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

**Conclusion**

This code serves as a foundation for a basic compiler front-end, demonstrating the process of lexical analysis, syntax tree construction, and semantic analysis. It provides a starting point for more advanced compiler development and serves as an educational resource for understanding compiler concepts.

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

**`build\_syntax\_tree(tokens)`**

This function is responsible for constructing a syntax tree from a list of tokens. The syntax tree represents the hierarchical structure of the program based on its syntax. Here's a breakdown of the key components:

**- Parameters:**

- `tokens`: A list of tokens generated from the source code.

**- Logic:**

- The function initializes a root node labeled as "PROGRAM" to represent the entire program.

- It iterates through each token in the input list.

- Depending on the type of token (e.g., DATATYPE, IDENTIFIER, OPERATOR, INTEGER, END\_STATEMENT), it creates a corresponding node and adds it as a child to the current node in the tree.

- The `current\_node` pointer is updated accordingly, allowing the function to traverse and build the tree in a depth-first manner.

- Whenever an END\_STATEMENT token is encountered, the `current\_node` is reset to the root, indicating the end of a statement.

**- Return Value:**

- The function returns the root of the constructed syntax tree.

**`perform\_semantic\_analysis(node)`**

This function is responsible for performing basic semantic analysis on the syntax tree. In the provided code, it specifically checks for compatibility of data types in arithmetic operations. Here's a breakdown:

**- Parameters:**

- `node`: A node in the syntax tree.

**- Logic:**

- The function recursively traverses the syntax tree in a depth-first manner.

- When it encounters a node labeled as "DATATYPE," it stores the current data type.

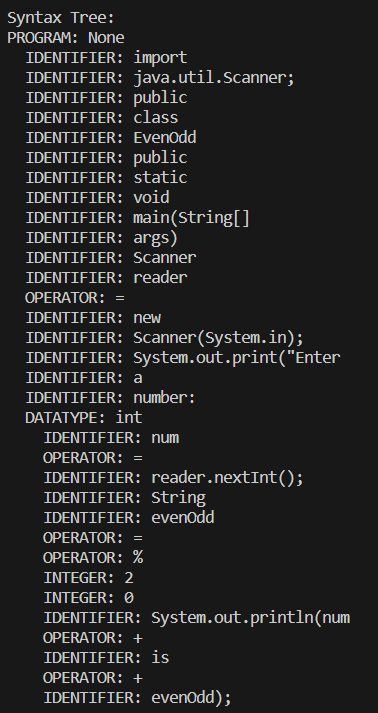
- When it encounters a node labeled as "OPERATOR" representing an arithmetic operation, it checks the compatibility of the data types of its operands.

- If operands are identifiers, it looks up their data types from the symbol table and prints a semantic error if the types are incompatible.

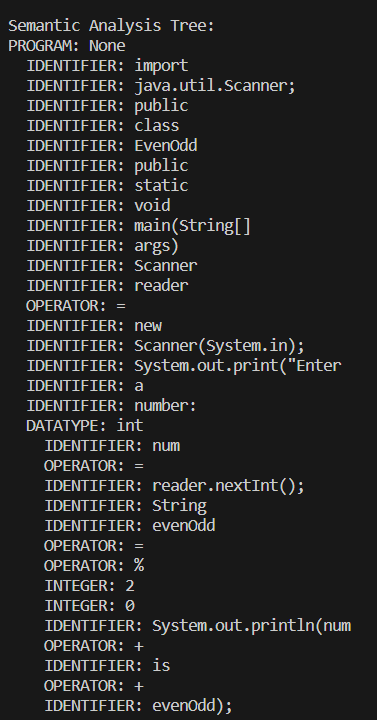
- If one operand is an identifier and the other is an INTEGER, it checks compatibility accordingly.

- The semantic errors are printed to the console.

**Output (Syntax tree):**

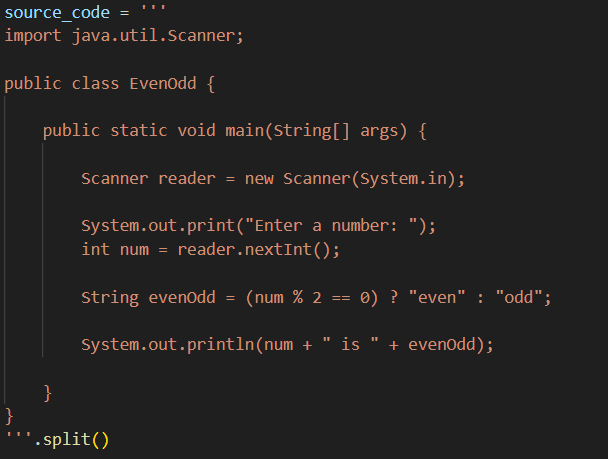
****

**Output (Semantic Analysis):**

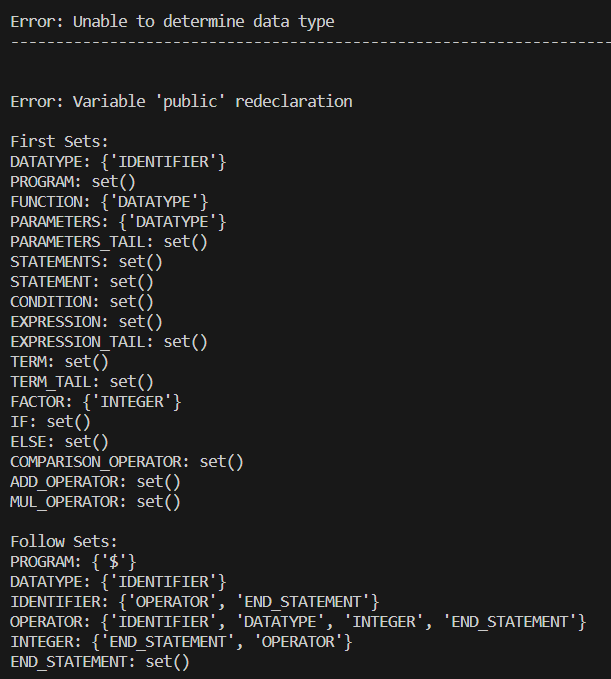
****

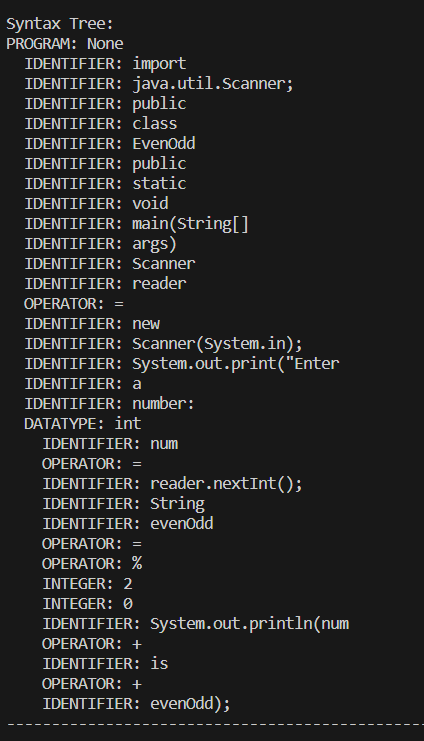
1. **Task 3: (Data Flow)**

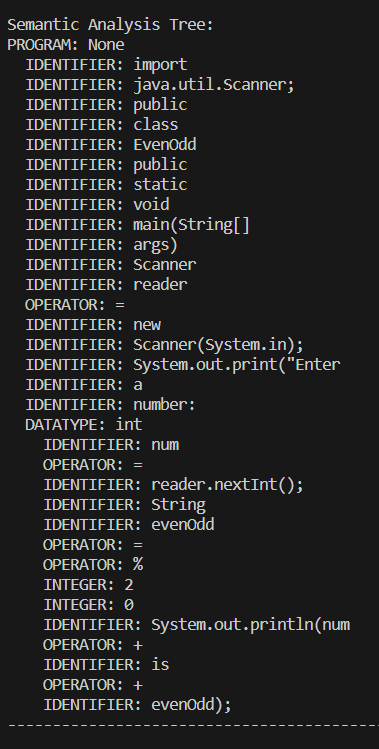
**Input:**

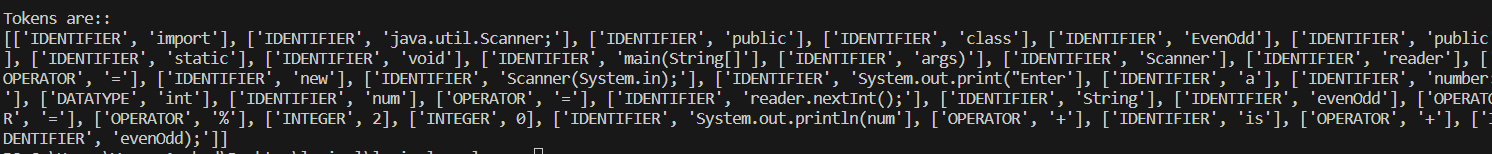


**Output:**









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

**Syntax Tree:**

A syntax tree is a hierarchical representation of the syntactic structure of a program. It breaks down the program into its constituent parts and their relationships. In the provided code:

**Syntax Tree Construction:**

1. SyntaxTreeNode Class: This class represents a node in the syntax tree. Each node has a label, a value, and a list of children.

2. build\_syntax\_tree Function: This function takes a list of tokens as input and constructs a syntax tree based on the syntax rules. It uses a stack-like approach to navigate through the tree, where each node represents a part of the program (e.g., DATATYPE, IDENTIFIER, OPERATOR). The function returns the root of the syntax tree.

3. print\_syntax\_tree Function: This function prints the syntax tree using depth-first traversal. It displays each node's label and value, along with its children.

**Example Syntax Tree:**

For the given source code snippet:

```java

int result = 100 + otherVar;

```

**The syntax tree might look like:**

```

PROGRAM: None

DATATYPE: int

IDENTIFIER: result

OPERATOR: =

INTEGER: 100

OPERATOR: +

IDENTIFIER: otherVar

```

---

**Semantic Analysis:**

Semantic analysis is the process of ensuring that the meaning of the program is correct with respect to the language specifications. It involves checking for semantic errors and building a symbol table to keep track of variable information.

**Semantic Analysis Construction:**

1. Symbol Table: A dictionary named `symbol\_table` is used to store variable information such as data type and value.

2. perform\_semantic\_analysis Function: This function performs semantic analysis on the syntax tree. It checks for compatibility of data types in expressions and reports semantic errors if found.

3. print\_semantic\_tree Function: This function prints the semantic tree. It displays each node's label and value, providing a structured view of the semantic analysis.

**Example Semantic Analysis:**

For the given source code snippet, the semantic analysis might check for data type compatibility in the expression `100 + otherVar`. If `otherVar` is declared as an integer, it's considered valid. Otherwise, a semantic error is reported.

**Link between Syntax Tree and Semantic Analysis:**

1. Syntax Tree Generation: The syntax tree is first constructed based on the syntactic structure of the source code.

2. Traversal for Semantic Analysis: The generated syntax tree is then traversed during semantic analysis. As the tree is traversed, information about identifiers and operators is used to perform semantic checks.

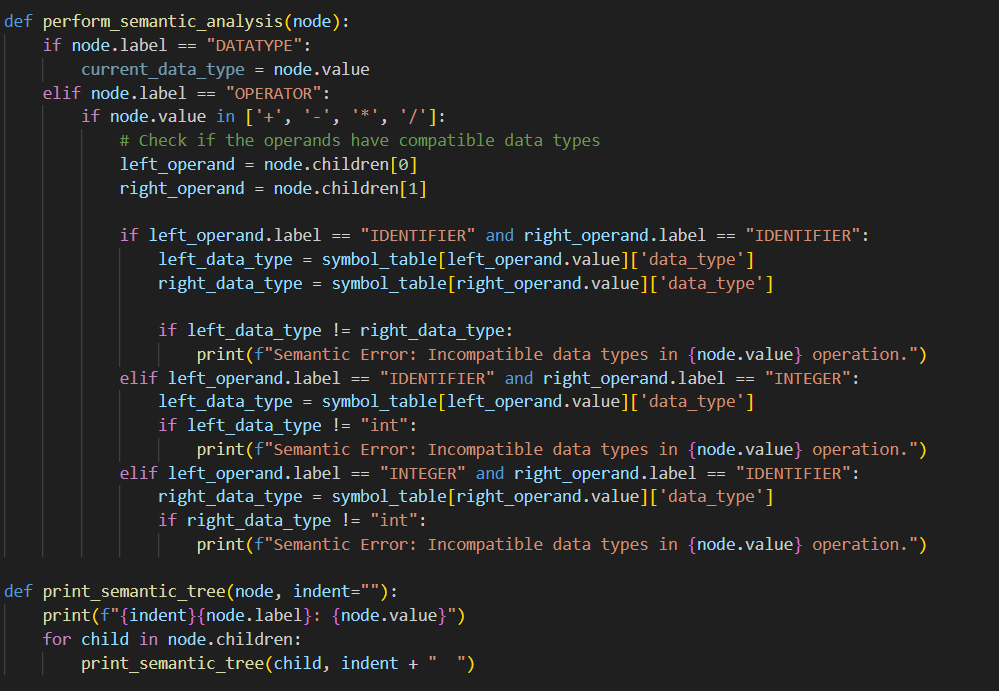
3. Symbol Table Usage: The symbol table is accessed during semantic analysis to retrieve information about identifiers and their data types.

4. Error Reporting: If semantic errors are detected during traversal, appropriate error messages are generated, indicating the nature of the error.

**Syntax Tree Function:**



**Semantic Analysis Function:**

****

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.

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