Objectives

1. Implementation of the lexical analyzer using Flex.
2. Identification and tokenization of various lexical elements, including keywords, identifiers, literals, and operators.
3. Definition of grammar rules to capture the syntax of the designed programming language.
4. Semantic analysis to ensure correctness in variable usage, type checking, and scope handling.
5. Development of a Turing-complete programming language with five primitive data types (int, float, bool, char, string).
6. Construction of a parser using Bison and integration with the lexer for unified functionality.

Introduction

FLEX

FLEX (Fast Lexical Analyzer Generator) was used for lexical analysis, the first phase of the compiler. It divides the source code into meaningful tokens like keywords, identifiers, literals, and operators. These tokens serve as input for the parser. The lexer was also designed to remove comments and unnecessary whitespaces to streamline the parsing process.

BISON

Bison was employed for syntax analysis and semantic validation. It generates a bottom-up parser from the grammar definition, designed to support a strongly typed programming language. Grammar rules define valid combinations of tokens, and semantic actions were used to generate an Abstract Syntax Tree (AST) and perform type checking.

Together, Flex and Bison facilitated the development of a custom compiler capable of analyzing, parsing, and validating source code for the defined language.

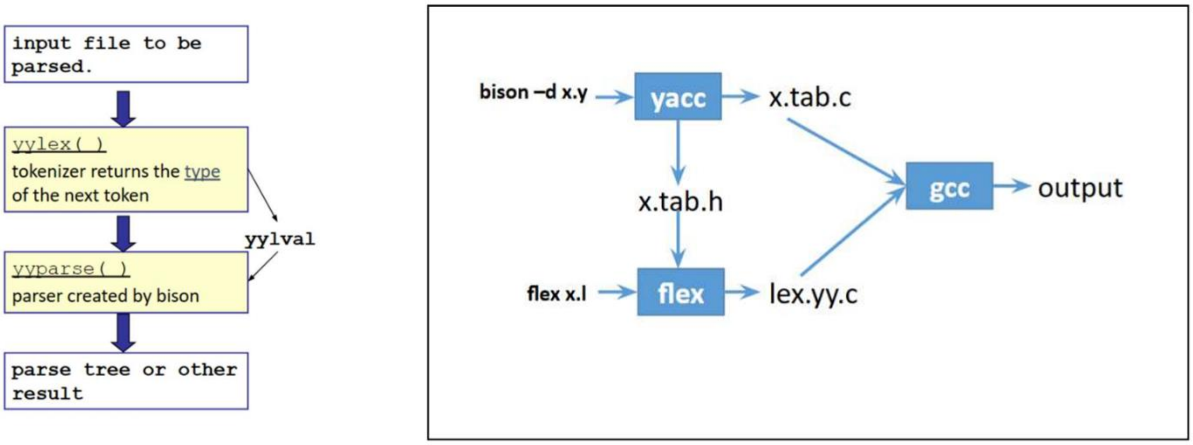

Fig: The parser process of a language and the union of flex and parse.

Key Features of the Programming Language

Data Types

- **INTEGER: Standard signed integers.**
  Example:

  INTEGER X = 10;

- **FLOAT: Floating-point numbers.**
  Example:

  FLOAT PI = 3.14;

- **BOOLEAN: Logical true/false values.**
  Example:

  BOOLEAN IS_TRUE = TRUE;

- CHARACTER: Single ASCII characters.
  Example:

  ```
  CHARACTER LETTER = 'A';
  ```

- STRING: Sequences of characters.
  Example:

  ```
  STRING MESSAGE = "HELLO";
  ```

Operators

- Arithmetic Operators: +, -, *, /, %.
  Example:

  ```
  X = A + B * C;
  ```

- Relational Operators: <, >, <=, >=, ==, !=.
  Example:

  ```
  IF (X > Y)
  ```

- Logical Operators: &&, ||, !.
  Example:

  ```
  IF (X > Y && Y < Z)
  ```

Control Structures

- IF-ELSE:

  ```
  IF (X > 10) {
      PRINT("GREATER THAN 10");
  } ELSE {
  ```

```
        PRINT("LESS THAN OR EQUAL TO 10");
    }
```

- WHILE:

```
     WHILE (X < 10) {
        X++;
    }
```

- FOR:

```
    FOR (INTEGER I = 0; I < 5; I++) {
        PRINT(I);
    }
```

- SWITCH-CASE:

```
    SWITCH (X) {
        CASE 1: PRINT("ONE"); BREAK;
        DEFAULT: PRINT("OTHER");
    }
```

Functions

Functions support argument passing, return values, and scoping. Example:

```
    INTEGER ADD(INTEGER A, INTEGER B) {
        RETURN A + B;
    }
```

Error Handling

Semantic errors such as type mismatches, undeclared variables, or incompatible operations are handled with meaningful error messages.

Table: Tokens Used in the Project

Serial No.	Token	Input String	Meaning
1	TOKEN_INT	INT	Integer keyword
2	TOKEN_FLOAT	FLOAT	Float keyword
3	TOKEN_BOOL	BOOL	Boolean keyword
4	TOKEN_CHAR	CHAR	Character keyword
5	TOKEN_STRING	STRING	String keyword
6	TOKEN_IF	IF	Conditional keyword
7	TOKEN_ELSE	ELSE	Else keyword
8	TOKEN_WHILE	WHILE	While loop keyword
9	TOKEN_FOR	FOR	For loop keyword
10	TOKEN_SWITCH	SWTICH	Switch-case construct
11	TOKEN_CASE	CASE	Case in switch statements
12	TOKEN_DEFAULT	DEFAULT	Default case in switch
13	TOKEN_PLUS	+	Addition operator

Serial No.	**Token**	**Input String**	**Meaning**
14	TOKEN_MINUS	-	Subtraction operator
15	TOKEN_MULT	*	Multiplication operator
16	TOKEN_DIV	/	Division operator
17	TOKEN_AND	&&	Logical AND operator
18	TOKEN_OR	`	
19	TOKEN_EQ	==	Equality comparison
20	TOKEN_NEQ	!=	Not equal comparison
21	TOKEN_LT	<	Less than operator
22	TOKEN_GT	>	Greater than operator
23	TOKEN_RETURN	RETURN	Return statement
24	TOKEN_BREAK	BREAK	Exit loop or switch
25	TOKEN_CONTINUE	CONTINUE	Skip to next iteration
26	TOKEN_PRINT	EXERT	Output statement

Implementation

1. **Lexer (Flex)**
    - Defined rules to tokenize keywords, operators, and identifiers.
    - Integrated handling for comments and multi-line strings.

2.Parser (Bison)

- Defined grammar for control structures, variable declarations, and expressions.
- Semantic rules validated type compatibility and scoping.
- Generated an Abstract Syntax Tree for further processing.

3.Semantic Analysis

- Performed type checking and ensured variables were declared before use.
- Checked compatibility of operands in expression

Challenges

1. Handling nested control structures and scoping rules.
2. Implementing advanced error recovery mechanisms for syntax errors.
3. Managing memory allocation and freeing during AST creation and destruction.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Principles of Compiler Design*.
2. Official Flex and Bison documentation.
3. Tutorials on compiler construction at GeeksforGeeks.