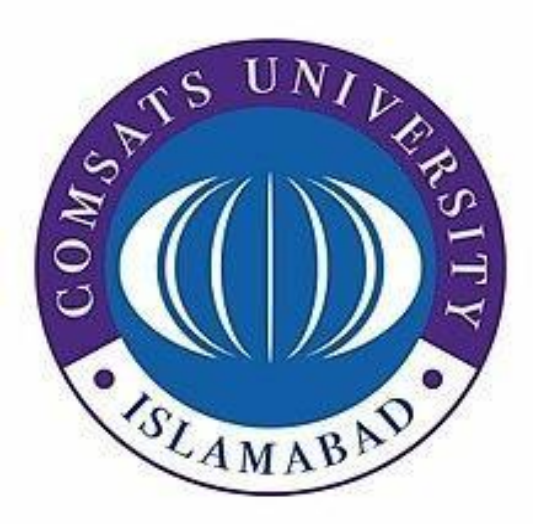
COMPILER CONSTRUCTION

LAB Terminal



By

M Bilal Younas M Sharjeel Iqbal

FA19-BCS-062

FA20-BCS-022

Section - B

Submitted to,

Dr. Bilal Bukhari

Department of Computer Science

COMSATS University Islamabad,

Attock Campus

PAPER SOLUTION

Q1 . Brief of the project

- **Lexical Analyzer (Scanner):**
  This component breaks the source code into a sequence of tokens by analyzing the characters.

- **Syntax Analyzer (Parser):**
  It checks the syntax of the source code based on the grammar rules of the programming language and generates a syntax tree.

- **Semantic Analyzer:**
  This component ensures that the semantics of the source code are correct. It checks for meaningful constructs and relationships between symbols.

- **Intermediate Code Generator:**
  Converts the source code into an intermediate representation, making it easier to optimize and target different platforms.

- **Code Optimizer:**
  Enhances the intermediate code to improve the program's performance. It includes techniques like constant folding, loop optimization, and dead code elimination.

- **Code Generator:**
  Translates the optimized intermediate code into machine code or another high-level language, depending on the target platform.

- **Symbol Table:**
  Manages information about symbols (variables, functions, etc.) in the source code, facilitating scope resolution and type checking.

- **Error Handler:**
  Detects and reports errors in the source code, providing meaningful error messages to assist developers in debugging.

- **Run-time Library:**
  A set of pre-written code that handles certain operations at runtime, such as input/output, memory management, and other language-specific features.

- **Loader and Linker:**
  If the compiler generates object code, a loader loads the code into memory for execution, and a linker combines it with other necessary modules and libraries.
- **Debugging and Profiling Tools:**
  Additional tools that help developers identify and fix errors, as well as analyze the performance of the compiled code.

Q2 . Functionalities (code Flow) :-

- In compiler construction, control flow refers to the order in which the program's statements are executed. It involves managing the flow of control through the generated code, ensuring that the program executes in the correct sequence

- **CODE :-**

```
class Compiler:
    def __init__(self):
        self.label_count = 0

    def compile_if_else(self, condition, true_branch, false_branch):
        # Generate unique labels for branches
true_label = f"LABEL_TRUE_{self.label_count}"
false_label = f"LABEL_FALSE_{self.label_count}"
end_label = f"LABEL_END_{self.label_count}"

        # Increment label count for uniqueness
        self.label_count += 1

        # Compile the condition
        self.compile_condition(condition, true_label, false_label)

        # Emit code for the true branch
self.emit_label(true_label)        self.compile(true_branch)
        self.emit_jump(end_label)

        # Emit code for the false branch
self.emit_label(false_label)
        self.compile(false_branch)

        # Emit the end label
self.emit_label(end_label)

    def compile_condition(self, condition, true_label, false_label):
# Simplified: Assume condition is a variable or constant
self.emit("LOAD", condition)
        self.emit_jump_if_zero(false_label)

    def compile(self, code):
        # Simplified compilation of code
        print("Compiling:", code)
```

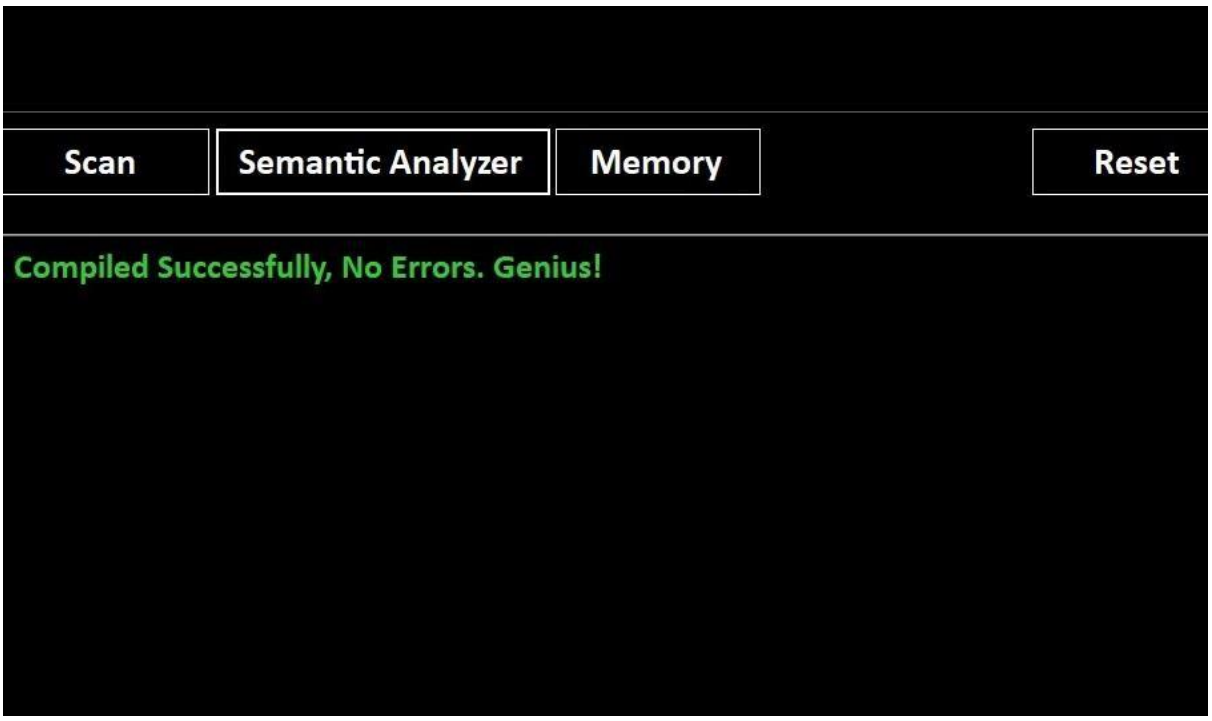
```
    def emit(self, operation, operand=None):
        print(f"{operation} {operand}")

    def emit_jump(self, label):
print(f"JUMP {label}")

    def emit_jump_if_zero(self, label):
print(f"JUMP_IF_ZERO {label}")

    def emit_label(self, label):
        print(f"{label}:")
```

Example usage compiler = Compiler() if_else_code = "x 0
JUMP_IF_ZERO LABEL_TRUE_0 LABEL_FALSE_0"
compiler.compile_if_else("x", "true_branch_code", "false_branch_code")



Q3 . DATA Flow :-

In compiler construction, data flow refers to the movement and transformation of data through a program during its execution. Analyzing data flow is crucial for optimizing code and understanding how variables are used and modified throughout the program.

Code :-

```
class Compiler:     def __init__(self):         self.variable_table = {}

    def compile_assignment(self, variable, expression):
        # Compile the expression
        self.compile(expression)

        # Emit code to store the result in the variable
self.emit("STORE", variable)

    def compile_arithmetic_operation(self, operand1, operand2, operator,
result_variable):         # Compile operands         self.compile(operand1)
self.compile(operand2)

        # Emit code for the arithmetic operation
if operator == '+':             self.emit("ADD")
elif operator == '-':             self.emit("SUB")
elif operator == '*':             self.emit("MUL")
elif operator == '/':             self.emit("DIV")

        # Store the result in the specified variable
self.emit("STORE", result_variable)

    def compile(self, code):
        # Simplified compilation of code
        print("Compiling:", code)

    def emit(self, operation, operand=None):
        print(f"{operation} {operand}")

# Example usage
compiler = Compiler()

# Example 1: Simple assignment assignment_code
= "x 10"
compiler.compile_assignment("x", "10")

# Example 2: Arithmetic operation arithmetic_code
= "y x 5 +"
compiler.compile_arithmetic_operation("x", "5", "+", "y")
```

Q4 . How functions works. Step by step :-

Sure, let's break down how functions work in the context of a compiler. A compiler translates source code written in a high-level programming language into machine code or another lower-level representation. Here are the steps involved in the functioning of a compiler:

1. Lexical Analysis (Scanning):
- The compiler scans the source code to identify and tokenize different components such as keywords, identifiers, literals, and symbols.
- Tokens represent the basic building blocks of the programming language.

2. Syntax Analysis (Parsing):
- The compiler parses the tokens to create a syntactic tree or abstract syntax tree (AST).
- The tree structure represents the hierarchical relationships between different language constructs.
- Functions and their parameters are identified during this phase.

3. Semantic Analysis:
- The compiler performs semantic analysis to ensure that the program adheres to the language's rules and semantics.
- It checks for correctness in variable usage, type compatibility, and other semantic rules.

4. Intermediate Code Generation:
- The compiler generates an intermediate representation of the source code.
- This representation is often an intermediate code, which is a platformindependent and high-level representation of the program.
- Functions are typically identified, and intermediate code for function calls and definitions is generated.

5. Code Optimization:
- The compiler performs various optimization techniques to improve the efficiency of the generated code.
- This may include common subexpression elimination, dead code elimination, and loop optimization.

6. Code Generation:
- The compiler translates the intermediate code into the target machine code or another lower-level representation.
- This involves mapping the high-level constructs to machine instructions. - Functions are translated into the corresponding machine code, and memory locations for variables are determined.

7. Register Allocation:
- The compiler assigns registers to variables to optimize the use of the processor's registers.
- Register allocation is crucial for improving the performance of the generated code.

8. Code Linking and Assembly:
- If the program consists of multiple source files or external libraries, the compiler may link them together.
- The linked code is then assembled into an executable file.

9. Loader and Execution:
- The loader loads the executable file into memory.
- The program is executed by the computer's processor, following the instructions generated by the compiler.

10. Error Handling:
- Throughout the compilation process, the compiler performs error checking and reports any syntax or semantic errors found in the source code.
- Error messages are generated to help the programmer identify and fix issues.

In summary, the functioning of a compiler involves multiple phases, each handling specific aspects of the compilation process. Functions in the source code are identified, and their corresponding machine code is generated during the intermediate code generation and code generation phases.

Q5 what challenges you faces during the project

we faced the problem of linking the different components of the compiler together .