CPT316: Programming Language Implementation and Paradigms
Semester 1 2024/2025
Assignment 1 (Final Version)

Weightage: 15%
Due Date: 29 November 2024 (Friday, before 5pm)
Submission: Submit via the elearn@USM by the due date.
Assignment Type: Group Work (4 members per group)

Assignment Overview

In this assignment, you will design and implement a simple compiler for a basic programming language called MiniLang. The compiler will handle lexical analysis, syntax analysis, and semantic analysis. The primary goal is to demonstrate your understanding of compiler construction while utilizing at least two different programming paradigms (e.g., procedural, object-oriented, functional) in constructing your solution.

Language Specification (MiniLang)
The language specification for MiniLang includes the following constructs:

1. Variables and Assignments:
   - Variable names consist of letters and digits, starting with a letter.
   - Assignment is done using the = operator.

2. Arithmetic Operations:
   - Support for basic arithmetic operations: +, -, *, /. ***(Choose at least two arithmetic operations)***
   - Parentheses ( ) can be used for grouping expressions.

3. Control Flow:
   - Conditional statements: if, else **OR/ AND** Loop statement: while.
   - Relational operators: ==, !=, <, >, <=, >=.

4. Output:
   - A print statement that outputs the value of an expression.

Example Program (MiniLang):

```
x = 5;
y = x * 10;
if (y > 40) {
    print(y);
} else {
    print(x);
}
```

Assignment Requirements

You are required to develop three phases of the Compiler:

1. **Lexical Analysis (Tokenizer/ Scanner):**
   - Implement a scanner that reads the MiniLang program and breaks it down into tokens (e.g., keywords, identifiers, literals, operators).
   - Tasks:
     - Tokenize the source code into its components: variables, operators, control flow keywords, and expressions.
     - Display lexical errors, such as unrecognized symbols or malformed tokens (e.g., variable names starting with digits, illegal characters).
   - Expected Output: A list of tokens identified in the source code, or an error message if lexical errors are detected.

2. **Syntax Analysis (Parser):**
   - Implement a parser using context-free grammar (CFG) to analyze the token sequence and generate an abstract syntax tree (AST).
   - Tasks:
     - Create a grammar for MiniLang that handles assignments, arithmetic operations, control flow, and output statements.
     - Implement the parser that constructs the AST based on the grammar.
     - Detect and display syntax errors (e.g., mismatched parentheses, missing semicolons, invalid statement structures).
   - Expected Output: The AST representation of the input program, or an error message if syntax errors are detected

3. ~~**Semantic Analysis:**~~
   - ~~Implement a type checker to ensure that variables are used correctly in the MiniLang program. The type checker will enforce type consistency and ensure all expressions are valid.~~
   - ~~Tasks:~~
     - ~~Ensure that all variables are declared before use.~~
     - ~~Check that arithmetic operations are performed on compatible types.~~
     - ~~Handle errors such as mismatched types, undeclared variables, or invalid operations.~~
   - ~~Expected Output: A success message if the program passes semantic checks, or an error message if type or semantic errors are detected.~~

Additional Requirements:

Use at least two different programming paradigms in your solution.
For example, you may use a procedural approach for the scanner and an object-oriented or a functional approach for the parser and type checker.

4. **Test Cases:**
   You are required to submit four MiniLang source code files to demonstrate the correctness of your compiler. These test cases should cover both valid and invalid programs to ensure that your compiler is robust and functions as expected.

   a. ~~**Two**~~ **One Valid Source Code Files:**
      - **Valid Test 1:** A MiniLang program demonstrating basic constructs such as variable assignments and arithmetic operations. This file should contain no lexical, or syntax errors.
      - ~~**Valid Test 2**: A more complex MiniLang program using control flow (e.g., if-else or while loops) and arithmetic expressions. This file should also be free of any errors.~~

      The compiler should successfully tokenize, parse, and pass all semantic checks, producing an appropriate Abstract Syntax Tree (AST) and final output**.**

b. One ~~Two~~ **Invalid Source Code Files:**
   - **Invalid Test 1 (Lexical/Syntax Errors):** A MiniLang program that includes lexical or syntax errors (e.g., unrecognized symbols, missing semicolons, improper use of parentheses).
   - ~~**Invalid Test 2 (Semantic Errors):** A MiniLang program that includes semantic errors (e.g., type mismatches, undeclared variables, invalid operations).~~

   The compiler should detect and report these errors with detailed and meaningful error messages, indicating the type and location of the errors.

Report Requirements

Your report should be well-organized and contain the following sections. Each section should detail your design choices, implementation process, and analysis.

The report should contain:

1. **Language Specification:**
   Provide a comprehensive description of the MiniLang language your compiler processes, including syntax, grammar rules, and semantics.

2. **Lexical Analysis:**
   Describe how you implemented the scanner/tokenizer, detailing the token types, tokenization process, and error handling mechanisms.

3. **Syntax Analysis:**
   Explain the design and implementation of the parser, outlining the grammar rules, how the AST is constructed, and how syntax errors are detected and handled.

4. **Semantic Analysis:**
   Detail the implementation of the type checker, including the rules enforced, scope management, and how semantic errors are detected and reported.

5. **Test Cases:**
   Provide the **source code** and **screenshots** of the output for each phase (Lexical Analysis, Syntax Analysis, ~~Semantic Analysis)~~ for all valid and invalid test cases.
   Ensure all figures are properly labelled.

6. **Programming Paradigms:**
   Reflect on how you utilized multiple programming paradigms (procedural, object-oriented, functional, etc.) in the construction of your compiler. Describe how each paradigm contributed to the solution.

Submission Guidelines

- Submit the entire project as a single **zip file**. The zip file should contain:
  1. The **source code** of the compiler.
  2. the two MiniLang **test cases.**
  3. The **report** in PDF format.
- The zip file should be named as `GroupNumber_CompilerAssignment.zip`.
- Late submissions will incur a penalty unless prior approval is obtained.

Evaluation Criteria

- Correctness and Functionality (50%): Tokenization, parsing, and type checking should work correctly.
- Application of Programming Paradigms (10%): Demonstration of at least two programming paradigms.
- Test Cases (20%): Quality of test cases and their respective outputs.
- Report Quality and Completeness (20%): Clarity and thoroughness of explanations, including the output screenshots.

Refer to the Rubric on the next page

Evaluation Criteria

Criteria	1 – Inadequate	2 - Needs Improvement	3 - Satisfactory	4 - Good	5 - Excellent	Weight
Correctness and Functionality (50%)						
Lexical Analysis (Scanner)	Lexical analysis does not function or significant errors.	Lexical analysis functions partially but with multiple errors.	Lexical analysis mostly functions with minor issues.	Lexical analysis works well, detects most tokens correctly, minor errors.	Lexical analysis is flawless and handles all tokens and errors perfectly.	20%
Syntax Analysis (Parser)	Parser does not function or produces incorrect syntax trees.	Parser works partially, with significant issues in parsing constructs.	Parser functions with minor errors, handles most constructs.	Parser works well, produces correct AST with a few minor issues.	Parser is flawless, produces correct AST, and handles all constructs correctly.	20%
~~Semantic Analysis (Type Checker)~~	~~Type checking is absent or incorrect for most constructs.~~	~~Type checking is incomplete or incorrect in several areas.~~	~~Type checking is mostly correct, minor issues.~~	~~Type checking is well-implemented, minor issues in type handling.~~	~~Type checking is flawless, handles all constructs correctly, including errors.~~	~~15%~~
Application of Programming Paradigms	No clear use of different paradigms or poorly implemented.	Minimal application of two paradigms with weak execution.	Two paradigms used correctly in most areas, minor integration issues.	Two paradigms applied effectively, good integration in different components.	Paradigms are used seamlessly and appropriately, with clear purpose and strong execution.	10%
Report Quality and Completeness (50%)						
Language Specification	Language specification is unclear or missing.	Language specification is incomplete or confusing.	Language specification covers most aspects with minor omissions.	Language specification is clear and detailed, minor issues.	Language specification is clear, comprehensive, and well-structured.	10%
Lexical and Syntax Analysis	Little to no explanation of design choices for these phases.	Basic explanations provided, but missing significant details.	Clear explanation with some omissions or lack of depth.	Detailed explanation of design choices with minor omissions.	Comprehensive, clear, and well-justified explanation of all phases.	10%
Test Cases	Few or no valid test cases provided, significant execution issues, missing screenshots.	Some valid test cases, but incomplete or incorrect output and missing screenshots.	Valid test cases provided and function with minor issues, some output and screenshots.	Test cases demonstrate all language features, mostly accurate results, with output and screenshots provided for each phase.	Comprehensive valid test cases, showcasing all features perfectly with detailed output and screenshots for each phase.	10%
	No or very few invalid test cases provided, error handling missing, screenshots absent.	Some invalid test cases provided, but error handling and output screenshots are incomplete.	Invalid test cases provided, and error handling works with minor issues, screenshots provided for some phases.	Invalid test cases demonstrate error handling effectively with minor issues, screenshots included for all phases.	Invalid test cases are comprehensive, with flawless error handling and detailed output screenshots for each phase.	10%
Programming Paradigms	No discussion of programming paradigms or poorly explained.	Minimal discussion, lacking depth or clarity.	Discussion covers paradigms, but lacks depth or clarity.	Clear discussion with solid understanding of the paradigms.	Thorough, insightful, and well-explained discussion of paradigm usage.	10%