**Compiler Project**

**Authors:** Arlind Lacej, Ridvan Plluzhina  
**Course:** Formal Languages and Compilers  
**Academic Year:** Second Semester 2024/25

* **Table of Contents**

1. Project Overview
2. Language Specification
3. Input Format & Examples
4. Installation & Usage
5. Test Cases
6. Technical Implementation
7. Error Handling
8. Learning Outcomes
9. **Project Overview**

This project provides a practical and theoretical insight into the core components of compiler design and implementation.

It implements a **lexical analyzer** and **parser** for a simple programming language using:

* **FLEX** for lexical analysis (tokenization)
* **Bison/YACC** for syntax analysis (parsing)
* **Custom symbol table** for semantic analysis

**Key Features**

* Variable declarations (int, float)
* Arithmetic expressions (+, -, \*, /)
* Assignment operations with type checking
* Comprehensive error reporting
* Symbol table management

1. **Language Specification**

**Grammar Rules**

program ::= stmt\_list

stmt\_list ::= stmt

| stmt\_list stmt

stmt ::= var\_decl ';'

| assignment ';'

| expression ';'

var\_decl ::= type ID

type ::= "int"

| "float"

assignment ::= ID '=' expression

expression ::= expression '+' term

| expression '-' term

| term

term ::= term '\*' factor

| term '/' factor

| factor

factor ::= '(' expression ')'

| NUM

| ID

ID ::= [a-zA-Z\_][a-zA-Z0-9\_]\*

NUM ::= integer | floating-point literal

**Supported Data Types**

* int – Integer variables
* float – Floating-point variables

**Operators**

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| + | Addition | x + y |
| - | Subtraction | x - y |
| \* | Multiplication | x \* y |
| / | Division | x / y |
| = | Assignment | x = 5 |

1. **Input Format & Examples**

**Basic Syntax Rules**

* Each statement must end with a semicolon (;)
* Variables must be declared **before use**
* Comments start with // and continue to end of line

**Valid Input Examples :**

// Variable declarations

int x;

float y;

// Assignments

x = 10;

y = 3.14;

// Expressions

x = 5 + 3;

y = x \* 2.5;

**Sample Test Files :**

**test1.txt**

float x;

float y;

x = 2.0 + 2.0;

y = 3.25 + 3.25;

**test2.txt**

int a;

a = 5.5; // Type warning: float assigned to int

**test3.txt**

z = 8; // Error: undeclared variable

1. **Installation & Usage**

**Prerequisites**

* **MSYS2** environment
* **GCC** compiler
* **Flex** (lexical analyzer generator)
* **Bison** (parser generator)

**Compilation Steps :**

# 1. Generate lexer

flex lexer.l

# 2. Generate parser

bison -d parser.y

# 3. Compile everything

gcc -Wall -g -o my\_compiler parser.tab.c lex.yy.c symbol\_table.c -lfl

# 4. Run with test file

./my\_compiler < tests/test1.txt

**Quick Build with Makefile :**

#Build all targets:  
make

#Run all tests:  
make test

#Clean up generated files:  
make clean

1. **Test Cases**

| **Test File** | **Purpose** | **Expected Output** |
| --- | --- | --- |
| test1.txt | Basic arithmetic | Assignment OK: x = 4.000000 |
| test2.txt | Type checking | Type error: assigning float to int |
| test3.txt | Undeclared variables | Error: Undeclared variable z |
| test4.txt | Multiple variables | Multiple successful assignments |
| test5.txt | Complex expressions | Expression evaluation results |

**Sample Output :**

$ ./my\_compiler < tests/test1.txt

Assignment OK: x = 4.000000

Assignment OK: y = 6.500000

$ ./my\_compiler < tests/test2.txt

Type error: assigning float to int variable a

Assignment OK: a = 5.500000

1. **Technical Implementation**

**File Structure**

compiler\_project/

├── lexer.l # Flex lexical analyzer

├── parser.y # Bison parser grammar

├── symbol\_table.c # Symbol table implementation

├── symbol\_table.h # Symbol table header

├── tests/ # Test input files

│ ├── test1.txt

│ ├── test2.txt

│ └── ...

└── build/ # Compiled binaries

└── my\_compiler

**Symbol Table Operations :**

* insert(name, type) – Add new variable
* lookup(name) – Get variable type
* set\_value(name, value) – Update variable value
* get\_value(name) – Retrieve variable value

**Token Types :**

* NUM – Numeric literals (e.g., 123, 3.14)
* ID – Identifiers (variable names)
* INT\_TYPE – "int" keyword
* FLOAT\_TYPE – "float" keyword

1. **Error Handling**

The compiler provides comprehensive error detection:

**Lexical Errors**

* Invalid characters or tokens

**Syntax Errors**

* Missing semicolons
* Invalid expression structure
* Malformed statements

**Semantic Errors**

* **Undeclared variables:**  
  Error: Undeclared variable x
* **Type mismatches:**  
  Type error: assigning float to int variable x

**Example Error Output :**

Error: Undeclared variable z

Type error: assigning float to int variable a

1. **Learning Outcomes**

This project demonstrates understanding of:

Lexical Analysis – The ability to transform raw source code into a sequence of meaningful tokens using tools like FLEX. This involves recognizing keywords, identifiers, literals, and symbols, which form the building blocks for further analysis.

Syntax Analysis – Understanding how to construct and apply context-free grammars using Bison/YACC to parse token streams into structured parse trees. This stage validates the programs syntactic correctness according to language rules.

Semantic Analysis – Developing and managing a symbol table to enforce semantic correctness, such as type checking, variable declarations, and scope resolution. This ensures that the program not only follows the grammar but also makes logical sense.

Compiler Construction – Gaining experience with the full compilation process, from lexical analysis to parsing and semantic checking, simulating a real-world compiler pipeline.

Tool Usage – Learning how to effectively use industry-standard tools like Flex for lexical analysis and Bison/YACC for parser generation, including integrating them within a build system and testing environment.