Final Project on

Compiler Design Lab

Date of submission: 10th April, 2025

Submitted by:
Md. Shakib Hossen

ID: 1905017, Reg. No.: 000012745
Session: 2019-2020
Department of Computer Science and Engineering

Begum Rokeya University, Rangpur

Submitted to:

Marjia Sultana

Assistant Professor

Department of Computer Science and Engineering

Begum Rokeya University, Rangpur

Project Overview

Title: Design and Implementation of a Mini Compiler for Arithmetic Expressions, Control Statements (If-Else, For, While), Using Flex and Yacc.

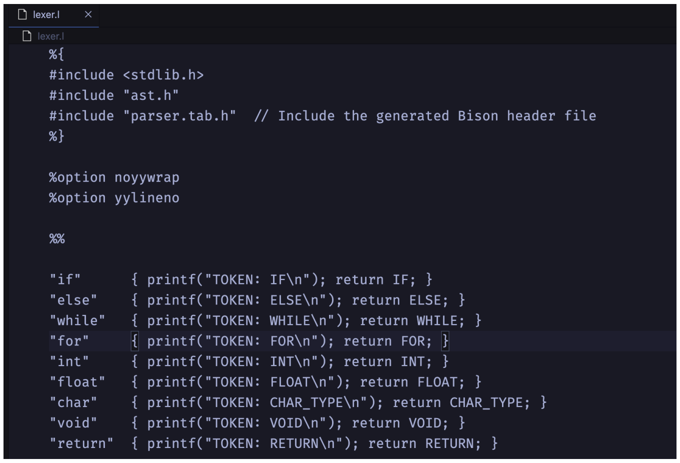
Languages Used: C, Flex, Yacc

Input: a set of statements that includes (If-Else, For, While)

Expected Output: Validity of the input expression, evaluate the expression if it is arithmetic statement, Show intermediate Code Generation.

Components and Their Functionality

1. lexer.l



```
%{
#include <stdlib.h>
#include "ast.h"
#include "parser.tab.h"  // Include the generated Bison header file
%}

%option noyywrap
%option yylineno

%%

"if"      { printf("TOKEN: IF\n"); return IF; }
"else"    { printf("TOKEN: ELSE\n"); return ELSE; }
"while"   { printf("TOKEN: WHILE\n"); return WHILE; }
"for"     { printf("TOKEN: FOR\n"); return FOR; }
"int"     { printf("TOKEN: INT\n"); return INT; }
"float"   { printf("TOKEN: FLOAT\n"); return FLOAT; }
"char"    { printf("TOKEN: CHAR_TYPE\n"); return CHAR_TYPE; }
"void"    { printf("TOKEN: VOID\n"); return VOID; }
"return"  { printf("TOKEN: RETURN\n"); return RETURN; }
```

•**Purpose**: Performs **lexical analysis** by tokenizing the input source code.

•**How It Works**:

–Uses regular expressions to identify tokens such as keywords (`for`, `printf`), identifiers, operators (`+`, `-`, `*`, `/`), and literals (numbers, strings).

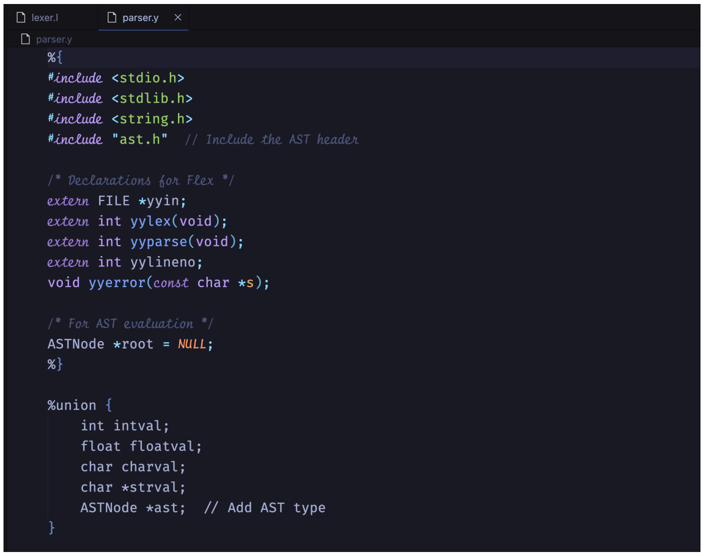
–Outputs a stream of tokens to the parser.

•**Example**: Input: for `(i = 0; i < 10; i++)`

•Output: Tokens: `FOR`, `IDENTIFIER`, `ASSIGN`, `NUMBER`, `SEMICOLON`, `IDENTIFIER`, `LESS_THAN`, `NUMBER`, etc.

---

2. parser.y



```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "ast.h"  // Include the AST header

/* Declarations for Flex */
extern FILE *yyin;
extern int yylex(void);
extern int yyparse(void);
extern int yylineno;
void yyerror(const char *s);

/* For AST evaluation */
ASTNode *root = NULL;
%}

%union {
    int intval;
    float floatval;
    char charval;
    char *strval;
    ASTNode *ast;  // Add AST type
}
```

- **Purpose**: Performs **syntax analysis** by parsing the token stream into an **Abstract Syntax Tree (AST)**.
- **How It Works**:

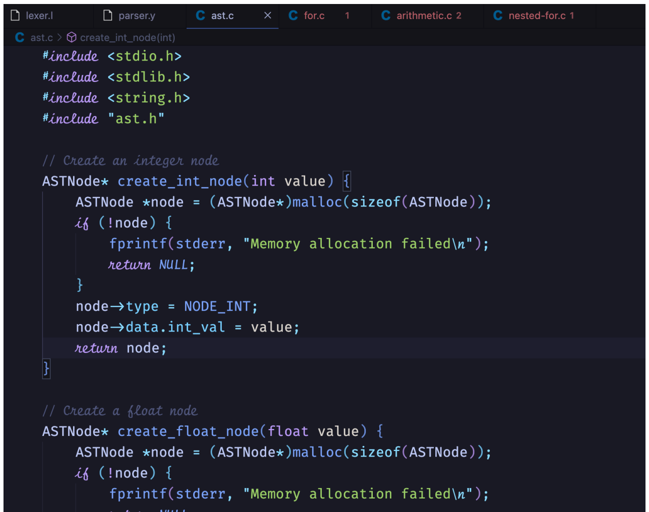
–Defines the grammar of the language using BNF (Backus-Naur Form).

–Constructs AST nodes for constructs like loops, expressions, and function calls.

- **Example**: Input: Tokens from the lexer. Output: AST for `for (i = 0; i < 10; i++) { printf("%d\n", i); }`.

---

3. ast.c



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "ast.h"

// Create an integer node
ASTNode* create_int_node(int value) {
    ASTNode *node = (ASTNode*)malloc(sizeof(ASTNode));
    if (!node) {
        fprintf(stderr, "Memory allocation failed\n");
        return NULL;
    }
    node->type = NODE_INT;
    node->data.int_val = value;
    return node;
}

// Create a float node
ASTNode* create_float_node(float value) {
    ASTNode *node = (ASTNode*)malloc(sizeof(ASTNode));
    if (!node) {
        fprintf(stderr, "Memory allocation failed\n");
```

- **Purpose**: Implements the **Abstract Syntax Tree (AST)** and its evaluation.
- **How It Works**:

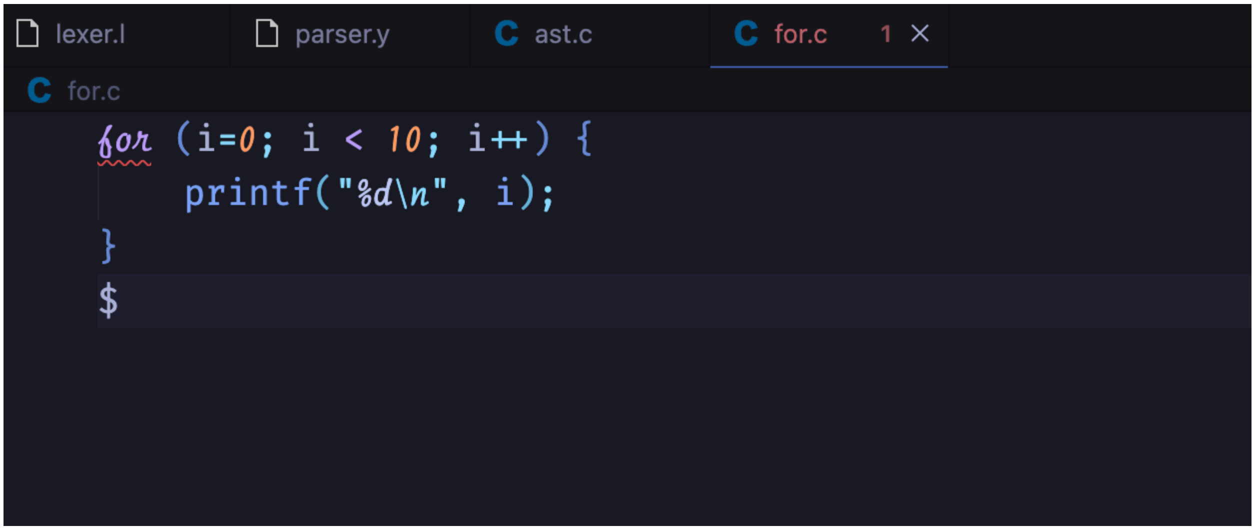
–Defines functions to create and manage AST nodes for different constructs:

- `create_int_node`: Creates an integer node.

        - `create_binop_node`: Creates a binary operation node.
        - `create_unaryop_node`: Creates a unary operation node.
        - `create_assign_node`: Creates an assignment node.
        - `create_funcall_node`: Creates a function call node.
        - `create_argument_list_node`: Creates an argument list node.
    - – Implements evaluate_ast to recursively evaluate the AST and execute the program.
- **Example**: Input: AST for `for (i = 0; i < 10; i++) { printf("%d\n", i); }`. Output: Executes the loop and prints: 0 1 2 3 4 5 6 7 8 9

---

4. for.c



```
for (i=0; i < 10; i++) {
    printf("%d\n", i);
}
$
```

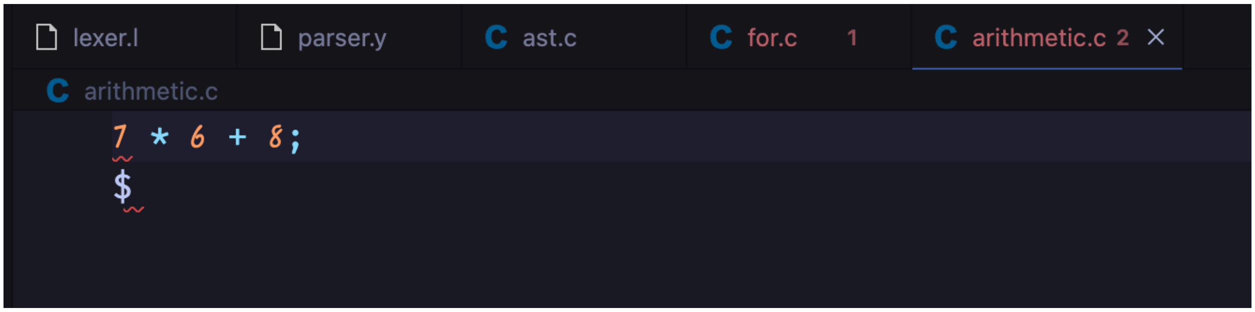
- **Purpose**: Contains test code for the for loop functionality.
- **How It Works**:
- Provides a sample program written in the custom language:

```
for (i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```

- This file is passed as input to the compiler for testing.

---

5. arithmetic.c

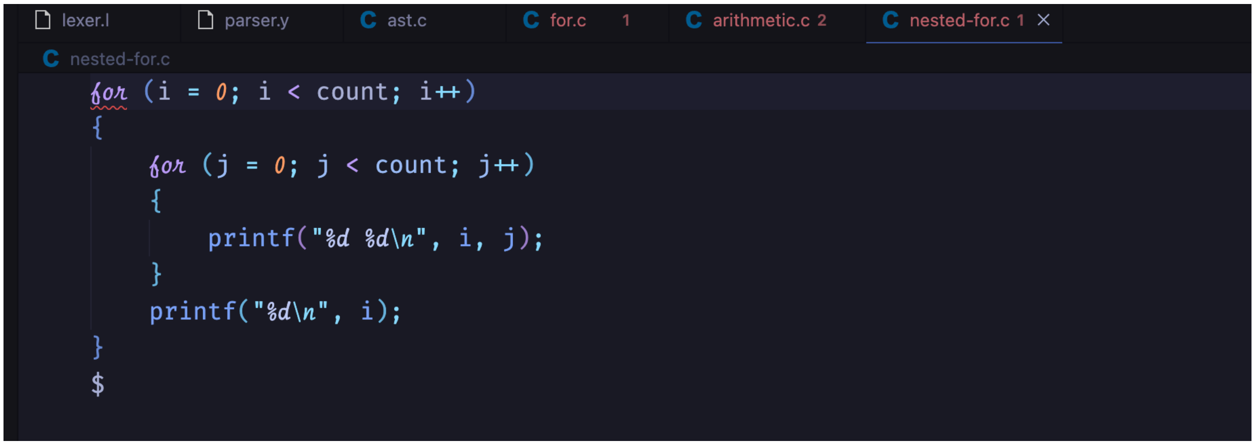


```
7 * 6 + 8;
$
```

- **Purpose**: Handles arithmetic expressions in the custom language.
- **How It Works**:
- Implements evaluation of arithmetic expressions like `7 * 6 + 8`.
- Uses AST nodes for binary operations (`+`, `-`, `*`, `/`) to compute results.
- **Example**: Input: `7 * 6 + 8` Output: `50`

---

6. nested-for.c



```
for (i = 0; i < count; i++)
{
    for (j = 0; j < count; j++)
    {
        printf("%d %d\n", i, j);
    }
    printf("%d\n", i);
}
$
```

- **Purpose**: Tests nested for loops in the custom language.
- **How It Works**:
- Provides a sample program with nested loops:

```
for (i = 0; i < 3; i++) {
    for (j = 0; j < 2; j++) {
        printf("%d %d\n", i, j);
    }
}
```

- This file is passed to the compiler to test nested loop functionality.

---

7. free_ast Function

- **Purpose**: Frees memory allocated for AST nodes.
- **How It Works**:
- Recursively traverses the AST and frees all nodes and their associated data.
- **Example**: Input: AST for a program. Output: Frees all memory used by the AST.

---

8. Symbol Table

- **Purpose**: Stores variable names and their values.
- **How It Works**:
- Maintains a simple array-based symbol table.
- Functions:
  - get_var_value: Retrieves the value of a variable.
  - set_var_value: Sets the value of a variable.
- **Example**: Input: `int x = 5; x = x + 2;` Output: Symbol table entry: `x = 7`.

---

9. compiler

- **Purpose**: The main entry point of the compiler.
- **How It Works**:
- Integrates the lexer, parser, and AST evaluator.

- Reads the input source code file (e.g., for.c), tokenizes it, parses it into an AST, and evaluates the AST.
- **Example**: Command: `./compiler for.c`
- Output: 0 1 2 3 4 5 6 7 8 9

How the Compiler Works

1. **Lexical Analysis**:

- The lexer (lexer.l) tokenizes the input source code into a stream of tokens.

2. **Syntax Analysis**:

- The parser (parser.y) parses the tokens into an AST based on the grammar rules.

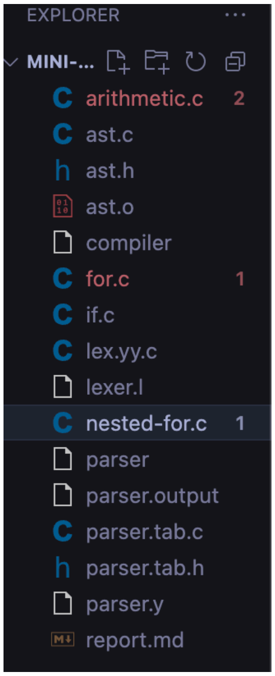
3. **Semantic Analysis**:

- The AST is analyzed for semantic correctness (e.g., variable declarations, type checking).

4. **Execution**:

- The AST is evaluated using evaluate_ast, which executes the program and produces output.

Project Structure

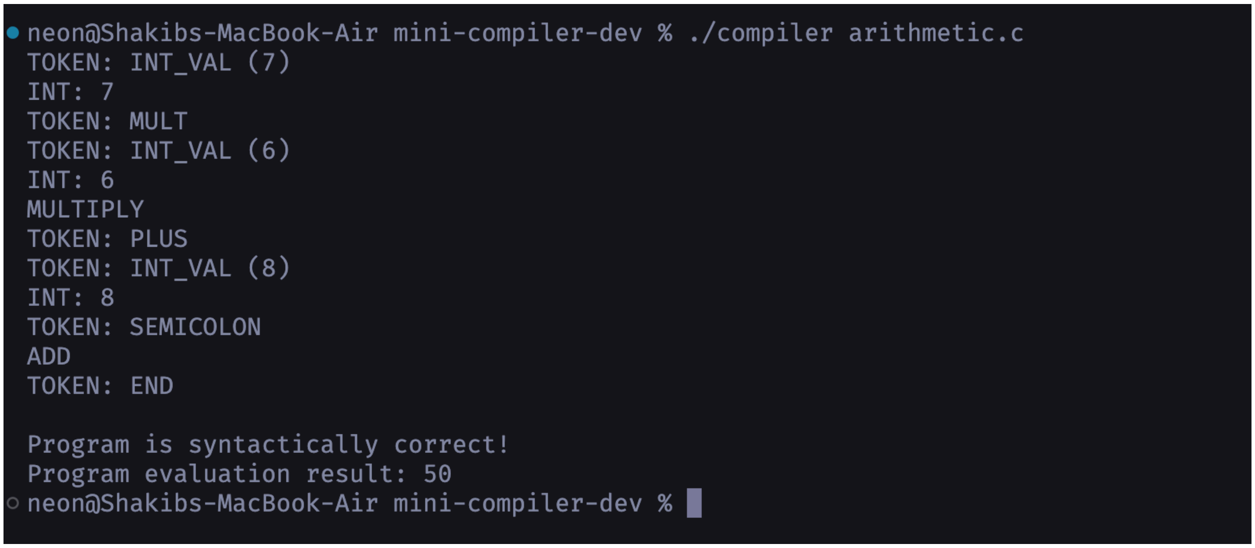


EXPLORER

MINI-...

arithmetic.c 2
ast.c
ast.h
ast.o
compiler
for.c 1
if.c
lex.yy.c
lexer.l
nested-for.c 1
parser
parser.output
parser.tab.c
parser.tab.h
parser.y
report.md

Example Workflow



```
● neon@Shakibs-MacBook-Air mini-compiler-dev % ./compiler arithmetic.c
  TOKEN: INT_VAL (7)
  INT: 7
  TOKEN: MULT
  TOKEN: INT_VAL (6)
  INT: 6
  MULTIPLY
  TOKEN: PLUS
  TOKEN: INT_VAL (8)
  INT: 8
  TOKEN: SEMICOLON
  ADD
  TOKEN: END

  Program is syntactically correct!
  Program evaluation result: 50
○ neon@Shakibs-MacBook-Air mini-compiler-dev % █
```

Which will evaluate the arimathic expression in arithmetic.c (7 * 6 + 8; $).