**CMPN403**

**Compilers**

**Project**

|  |  |  |
| --- | --- | --- |
| Name | Id | Tutorial |
| Abdelrahman Mohamed Ezzat | 1190158 | Wednesday |
| Amr Yasser Salah El-Din | 1190380 | Wednesday |
| Farah Mohamed Abdelfattah | 1190176 | Wednesday |
| Mohamed Hassan Mohamed | 1190118 | Sunday |

Table of Contents

[Project Overview 3](#_Toc166817874)

[Tools and Technologies used: 4](#_Toc166817875)

[List of Tokens 5](#_Toc166817876)

[List of Quadruples 6](#_Toc166817877)

# Project Overview

In this project, we developed a simple programming language compiler based on Lex and Yacc. Our programming language takes inspiration from C++ and includes similar features for variable and function declarations, statements, function calls, mathematical and logical expressions, among others.

To facilitate the usage of our compiler, we have created two scripts to provide different execution modes:

1. GUI Mode: This mode offers a user-friendly graphical interface. To run the compiler in this mode, the user needs to execute the 'GUI.bat' file in the file explorer. Once the GUI is open, the user can easily select the file containing the program to be compiled. The GUI allows direct editing of the program file and recompilation at the user's convenience. Additionally, the generated symbol table and quadruples (intermediate code representation) will be displayed in the GUI.
2. Batch Mode: This mode provides a CLI-like interface for running the compiler. To use this mode, the user needs to execute the 'run.bat' file. The compiler expects the program to be compiled located in the 'input/program.faam' file. After successful compilation, the output file containing the symbol table and other relevant information will be generated at 'output/output.faam'.

In both modes, our compiler leverages the Lex and Yacc tools to parse and translate the provided program written in our custom programming language, which closely resembles C++ syntax and semantics. The compiler generates an intermediate code representation (quadruples) and a symbol table. Our program also produces errors and warnings to allow further analysis if desired.

# Tools and Technologies used:

1. We used **Flex** for the lexer
2. We used **Bison** for the parser
3. We used **C++** for the symbol table & the quadruple handler and some functions in the parser
4. We used **bash** for running the script
5. We used **PyQT** for the GUI

# List of Tokens

|  |  |
| --- | --- |
| **Conditional Tokens** | **Mathematical Operators** |
| if | + |
| else | - |
| switch | \* |
| case | / |
| default | % |
| break | ++ |
| **Loops Tokens** | -- |
| do | **Assignment Operators** |
| while | = |
| for | += |
| **Functions Tokens** | -= |
| return | /= |
| **Data Types** | \*= |
| const | %= |
| int | **Relational Operators** |
| float | == |
| char | != |
| string | > |
| bool | < |
| void | >= |
| **Boolean Values** | <= |
| true | **Logical Operators** |
| false | && |
| **Bitwise Operators** | || |
| & | ! |
| | | **Comments** |
| ^ | **//** (one line comment) |
| **Identifiers** | **/\* \*/** (multi-line comment) |
| [a-zA-z][a-zA-z0-9\_]\* |  |

# List of Quadruples

|  |  |
| --- | --- |
| **Quadruple** | **Description** |
| **Math Operations** | |
| plus op1 op2 t | t = op1 + op2 |
| minus op1 op2 t | t = op1 – op2 |
| mul op1 op2 t | t = op1 \* op2 |
| div op1 op2 t | t = op1 / op2 |
| mod op1 op2 t | t = op1 % op2 |
| **Unary Operators** | |
| inc op1 | op1++ |
| dec op1 | op1-- |
| **Assignment Operators** | |
| assign op1 t | t = op1 |
| add\_assign op1 t | t += op1 |
| sub\_assign op1 t | t -= op1 |
| div\_assign op1 t | t /= op1 |
| mul\_assign op1 t | t \*= op1 |
| mod\_assign op1 t | t %= op1 |
| **Relational Operators** | |
| eq op1 op2 t | t = op1 == op2 |
| neq op1 op2 t | t = op1 != op2 |
| lt op1 op2 t | t = op1 < op2 |
| lte op1 op2 t | t = op1 <= op2 |
| gt op1 op2 t | t = op1 > op2 |
| gte op1 op2 t | t = op1 >= op2 |
| **Logical Operators** | |
| and op1 op2 t | t = op1 && op2 |
| or op1 op2 t | t = op1 || op2 |
| not op1 t | t = !op1 |
| **Bitwise Operators** | |
| bit\_and op1 op2 t | t = op1 & op2 |
| bit\_or op1 op2 t | t = op1 | op2 |
| bit\_xor op1 op2 t | t = op1 ^ op2 |
| **Casting** | |
| CAST op1 type | casting op1 to type *type* |
| **Functions** | |
| proc type funcName type arg1 type arg2 | type funcName(type arg1, type arg2)  this can take any number of arguments |
| return op1 | return op1 |
| call funcName t arg1 arg2 | t = funcName(arg1,arg2) |
| **Jumps** | |
| jmp L | jump to label L (unconditional) |
| jmp L on op1 boolean\_value | jump to label L if op1 is equal to boolean\_value (conditional) |

**Special Examples:**

|  |  |
| --- | --- |
| **if (x > y) {**  **x = 5;**  **}** | gt x y t0  jmp L0 on t0 false  assign 5 x  L0: |
| **int sum(int x, int y) {**  **return x + y;**  **}** | proc INT sum INT x INT y  plus x y t0  return t0 |
| **for(int i=0;i<5;i++) {**    **}** | L0:  lt i 5 t0  jmp L1 on t0 false  inc i  jmp L0  L1: |
| mod op1 op2 t | t = op1 % op2 |