



# Compilers Phase 2 Team 11

## **Team Members**

Num	Full Name in ARABIC	SEC	BN
1	مروان سامي محمد	2	18
2	مروان عماد مصطفي	2	19
3	سيف الدين هاني عبد الفتاح	1	20
4	محمد عاطف احمد	2	11





## **Project Overview**

A compiler project for our programming language, using lex and bison from GNU, which work as the base of the lexical analysis and syntax analysis. We wrote our tokens for the lex file and our grammar in the parser file. Using a symbol table to make semantic and syntax analysis. We made quadruples to work as an intermediate language between source and compiled code. You can also view a machine code in the assembly.txt. There's also a GUI!, where you can edit your code, save it, or compile it! And you can see from, the machine code, quadruples, warnings, errors, symbolTable, and any output from our code!

# **Tools and Technologies**

- Lex and Bison from GNU
- C
- Python (for GUI)
- Make (for automating tests)

## Lexer

#### **Tokens**

- int, float, string, char, bool, const, void
- for, while, do, break, continue
- if, else, switch, case, default
- print, true, false
- func ,return





### **Operators**

• >, <, >=, ==, !=, <=

• &&, ||,!

|, &, ^, ~

+, -, /, \*, %, ;

#### **Variables**

• [a-zA-Z\_][a-zA-Z0-9\_]\*

• int value : [-]?[0-9][0-9]\*

• float value: [-]?[0-9]+\.[0-9]\* or [-]?[0-9]+\.[0-9]\*[eE][-+]?[0-9]+

• char value: \'.\'

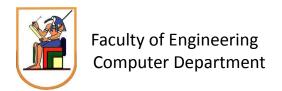
string value: \"[^"\n]\*\"

• comments: #[\s\t]\*.\*[\s\t]\*

• other chars: [(){},:]

## **Tokens**

Token	Description
INT_VALUE	Return the integer value
BOOL_VALUE	Return the boolean value
CHAR_VALUE	Return the char value
STRING_VALUE	Return the string value
INT	Return the integer token (int x;)
FLOAT	Return the float token





Token	Description
STRING	Return the string token
CHAR	Return the char token
BOOL	Return the bool token
CONSTANT	Return the const token (const int x = 1;)
VOID	Return the void token (func void test())
VARIABLE	Handle the variable and function names
FOR	Handle the for token
WHILE	Handle the while token
DO	Handle the do token
BREAK	Handle the break token
CONTINUE	Handle the continue token
IF	Handle the if token
ELSE	Handle the else token
GT	Handle the greater than (>) token
LT	Handle the less than (<) token
GE	Handle the greater than or equal (>=) token
LE	Handle the less than (<=) token
EQ	Handle the equal (==) token





Token	Description
NE	Handle the not equal (!=) token
AND	Handle the logical and (&&) token
OR	Handle the logical or (  ) token
NOT	Handle the logical not (!) token
BITWISE_OR	Handle the bitwise or ( ) token
BITWISE_AND	Handles the bitwise and (&) token
BITWISE_XOR	Handles bitwise xor (^) token
BITWISE_NOT	Handles the bitwise not (~) token
LEFT_SHIFT	Handles the bitwise left shift (<<) token
RIGHT_SHIFT	Handles the bitwise right shift (>>) token
ASSIGN	Handle the assign (=) token
INC	Handle the post and pre-increment (++) token
DEC	Handle the post and pre-decrement () token
ADD	Handle the addition (+) token
SUB	Handle the subtraction (-) token
MUL	Handle the multiplication (*) token
DIV	Handle the division (/) token
MOD	Handle the modulus (%) token





Token	Description
SEMICOLON	Handle the of the line (;) token
FUNCTION	Handle the function declaration (func) token
RETURN	Handle the return (return) token
PRINT	Handle the printing token (print(x))
SWITCH	Handle the switch (switch) token
CASE	Handle the case (case) token
DEFAULT	Handle the default (default) token

## **Parser**

For now, the parser has all the rules for syntax checking ONLY

#### How to define:

```
    Variable:

            int x = 42;
            int x;

    function:

            func int add(int a, int b = 0)
            {
            return a + b;
            }
            func void x(){
            if(true) {
```



# Faculty of Engineering Computer Department



```
};
      0 };
 for loop:
      o func void testForLoop() {
           for(int x = 0; x < 5; x = x+1) {
              print(x);
      0
           }
      0
      0 }
  do while:
      o func void testDoWhile() {
           int i = 0;
      0
           int sum = 0;
      0
           do {
             if (i % 2 == 1) {
                sum = sum + i;
      0
              }
      0
            i = i + 1;
      0
           } while(i < 10);
           print(sum);
      0 }
• if else:
      o func void ifElse() {
           int x = 0;
           if(true) {
      0
           x = 1;
      0
           } else {
      0
              x = 2;
      0
           }
      0
      0 }
```

• switch:





```
    func int testInt(int x) {
    switch (x) {
    case 1: {return 10;} # any case must be in a block
    case 2: {print(2); break;}
    default: {print(0); break;}
    }
```

# **Quadruples**

Quadruple	Description
assign	To assign a variable or a constant variable
print	To print a constant or variable
add	To add two expressions and the result to be in the 4th operand
sub	To sub two expressions and the result to be in the 4th operand
mul	To mul two expressions and the result to be in the 4th operand
div	To div two expressions and the result to be in the 4th operand
mod	To mod two expressions and the result to be in the 4th operand
and, or	Apply logical AND or OR between two expressions





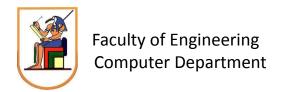
not	Apply logical not on an expression
label	To put a label for the if, switch, for, while, do while, and functions
jmp	To jmp to some label of the above quadruple
if_false	To check if the result of the logical operation is false or not then decide to where to jmp (used in the if statements specifically but can be use else where)
jf	Jump to some labe if the given bool expression is false
pop_param	To pop an argument when enter a function to assign it
push	Push variable value passed to the functions
push_const	Same as above put for constant values (just a rename)
func_label	A label to show the start of the function
return	Returns from a function and put its 4th operand as the returned value from the function if it's not a void function
jmp <i>call</i>	Jump to a specific label (not used anymor)
jmp func_ <function_n ame&gt;</function_n 	Jump to a function label

## How to run:

We have a Makefile for build, clean, run, and test the project

So to run it:

1. cd lex-parser





2. make // that will clean, build, and run the exe to test interactively

#### Run our test cases:

- 1. cd lex-parser
- 2. make test

#### Run our gui:

3. make gui

Tests are located at lex-parser/tests/inputs where valid cases must have a prefix of valid, and it will check for the return code from the parser process