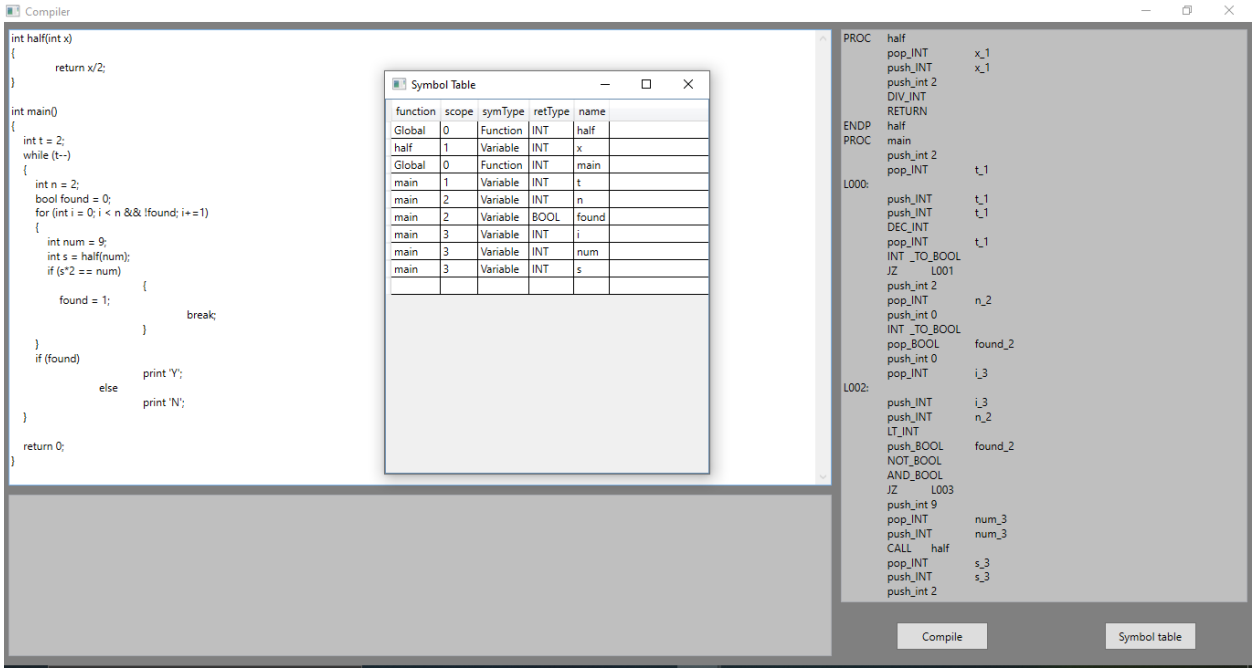
Team 12

Name	Sec	BN
Evram Youssef	1	8
Kareem Osama	2	5
Omar Ahmed	1	36
Muhammed Sayed	2	14

Compiler

A mini C++ compiler built with flex, bison and C++.
The GUI is built with C# and WPF.



Overview

Data types

It supports the following data types

1. void: valid only for functions that doesn't return a value
2. bool: boolean value (*true* or *false).*
3. char: a single character (e.g. 'a', 'b', ...)
4. int: integer value (e.g. 1, 2, …)
5. float: float point (e.g 1.2, 3.14, ...)

The following statements can be used:

1. Variable / Constant declaration

```
bool b = false;
char c = 'c';
int i = 1;
const float PI = 3.14;
```

```
int var;
```

2. Mathematical / Logical expressions

```
1+2*3/4.0-5%2;
!true || false && true;
5 >= 3;
~(1<<2)|(5>>1)^6&(-1);
x++;  ++x;
x--;  --x;
~x;
```

3. Assignment Statements

```
x = true+'c'+2+3.4;
x *= 4+5;
x >>= 4+5;
x |= 4+5;
x ||= false;
```

4. If statements and if-else statements

```
if (x == f)
{
    if (x/2 > 5)
        z = x;
    else
        z = f*2;
}
else
    z = y;
if (found)
    p = 'Y';
```

5. While loops, do-while loops, for loops

```
while (x < 5)
{
    x++;
}

do
{
    x++;
} while (x < 2);
```

```
for (int i = 0; i < 5; i++)
{
    print x;
}
while (x < 5)
    x++;
```

6. Switch statements

```
switch (x)
{
case 1:
    z = f*s;
    break;
case 2:
    z = f/s;
default:
    break;
}
```

7. Block structures and scopes as in C++

```
int x = 10;
{
    int x = 20;
}
```

8. Functions

```
int sum(int x, int y)
{
  return x + y;
}

void main()
{
  int a = sum(10, 20);
  a = sum(a, a);
}
```

Semantic Errors

1. Use of undeclared variables or functions.

```
int main(){
      x = 1;                  // invalid use of undeclared variable
      int x = sum(1,2);       // invalid use of undeclared function
}
```

2. Invalid Global statements: the only valid global statement is declaration of function or variable, anything else will cause a semantic error.

```
x = 1;            // invalid expression in global scope
while(x<3){       // invalid statement in global scope
      x++;
}
int y = 2;        // valid variable declaration in global scope
int main(){       // valid function declaration in global scope
      return 0;
}
```

3. Multiple declaration of the same variable in the same scope

```
int x;
int x = 1;        // multiple declaration of same variable
```

4. Function declaration should be in global scope.

```
int main(){
      int sum(int x, int y){       // invalid function declaration
            return x+y;            // inside a function
      }
      return 0;
}
```

5. Invalid default values for function parameters.

```
int sum(int x, int y = 1){ // invalid default value for parameter y
      return x+y;
}
```

6. Invalid return statements:
    a) Expected return value: for functions of any type except void.
    b) Invalid return statement: for void functions.

```
void fun_1(){
      return 2+3;       // invalid return statement for void function
}
int main(){
```

```
      int x = 5;
      return;           // expected return expression of type integer
}
```

7. More or less arguments than function declaration.

```
int sum(int x, int y){
      return x+y;
}
int main(){
      int x = sum(1)      //less arguments than function parameters
      int y = sum(1,x,x) //more arguments than function parameters
      return 0;
}
```

8. Float usage in integer operation

```
1.0%2;                  //invalid mod
1.0<<2;                 //invalid shift left
4.0>>1;                 //invalid shift right
1.0|3;                  //invalid bitwise or
1.0^5;                  //invalid xor
1.0&6;                  //invalid bitwise and
~1.0;                   //invalid bitwise not
```

9. break statement: should be inside for, while, do while, switch.

```
int main(){
      break;      // invalid break statement outside a loop or switch
      return 0;
}
```

10. continue statement: should be inside for, while, do while.

```
int main(){
      continue;   // invalid break statement outside a loop.
      return 0;
}
```

11. Constant assignment after declaration.

```
const int c = 1;
int main(){
      c = 5;      // invalid constant assignment after declaration
      return 0;
}
```

12. Calling a void function in a mathematical expression or assignment statement.

```
void func_1(){
```

```
      return;
}
int main(){
      int x = 1+func_1();    // unable to convert void to integer
      return 0;
}
```

13. Using pre/postfix increment/decrement with a boolean variable;

```
int main(){
      bool x = true;
      x++;  // invalid usage of increment operator with bool type
      return 0;
}
```

Tokens

Regex	Description
`[a-zA-Z_][a-zA-Z_0-9]*`	Identifiers (variable and function names)
`0|[1-9][0-9]*`	Integers
`(0|[1-9][0-9]*)\.[0-9]*`	Floats
`'[^'\n]'`	Characters
“while”, “for”, “do”, “if”, “else”, “switch”, “case”, “continue”, “break”, “default”, “print”, “const”, “return”, “true”, “false”	Reserved words
“int”, “float”, “char”, “bool”, “void”	Data types
“>=”, “<=”, “==”, “!=”, “<”, “>”, “&&”, “||”, “!”, “~”, “|”, “&”, “^”, “<<”, “>>”, “+”, “-”, “*”, “/”, “%”, “=”, “+=”, “-+”, “*=”, “/=”, “%=”, “|=”, “&=”, “^=”, “<<=”, “>>=”, “++”, “--”	Symbols used in expressions (almost all expression symbols of C++)
“(“, “)”, “;”, “{“, “}”, “:”, “,”	Other symbols

Production Rules

We used bison to determine the associativity of the expressions and precedence to make the grammar unambiguous. (they are not listed here)
Uppercase words are terminals, lowercase words are non-terminals.

Production Rule
`program -> function`
`function -> function stmt` `function -> ε`
`typ -> INT_TYPE | CHAR_TYPE | BOOL_TYPE | FLOAT_TYPE`

`decl -> typ VARIABLE`
`decl -> typ VARIABLE '=' expr`
`decl -> CONST typ VARIABLE`
`decl -> CONST typ VARIABLE '=' expr`

`stmt -> ';' | CONTINUE ';' | BREAK ';' | expr ';'| decl ';'`
`stmt -> PRINT exrp ';'`
 	
`stmt -> typ VARIABLE '(' param_list ')'  stmt`
`stmt -> VOID VARIABLE '(' param_list ')' stmt`
`stmt -> RETURN expr ';'`
`stmt -> RETURN ';'`
 	
`stmt -> '{' stmt_list '}'`
`stmt -> WHILE '(' expr ')' stmt`
`stmt -> DO stmt  WHILE '(' expr ')' ';'`
`stmt -> FOR '(' expr ';' expr ';' expr ')' stmt`
`stmt -> FOR '(' decl ';' expr ';' expr ')' stmt`
 	
`stmt -> SWITCH '(' expr ')' '{' switch_stmt '}'`
`stmt -> IF '(' expr ')' stmt`
`stmt -> IF '(' expr ')' stmt ELSE stmt`

`param_list -> decl | decl ',' param_list | ε`
`arg_list -> expr | expr ',' arg_list | ε`
`switch_stmt -> CASE const_expr ':' stmt_list`

```
switch_stmt -> CASE const_expr ':' stmt_list switch_stmt
switch_stmt -> DEFAULT ':' stmt_list
```

```
stmt_list -> stmt | stmt_list stmt
```

```
const_expr -> INTEGER | FLOAT | CHAR | BOOL
const_expr -> '!' const_expr
const_expr -> '~' const_expr
const_expr -> '-' const_expr
const_expr -> '+' const_expr
const_expr -> const_expr OR const_expr
const_expr -> const_expr AND const_expr
const_expr -> const_expr '|' const_expr
const_expr -> const_expr '^' const_expr
const_expr -> const_expr '&' const_expr
const_expr -> const_expr '+' const_expr
const_expr -> const_expr '-' const_expr
const_expr -> const_expr '*' const_expr
const_expr -> const_expr '/' const_expr
const_expr -> const_expr '%' const_expr
const_expr -> const_expr '<' const_expr
const_expr -> const_expr '>' const_expr
const_expr -> const_expr GE const_expr
const_expr -> const_expr LE const_expr
const_expr -> const_expr NE const_expr
const_expr -> const_expr EQ const_expr
const_expr -> const_expr SHIFT_LEFT const_expr
const_expr -> const_expr SHIFT_RIGHT const_expr
const_expr -> '(' const_expr ')'
```

```
expr -> INTEGER | FLOAT | CHAR | BOOL | VARIABLE
expr -> INCR VARIABLE
expr -> VARIABLE INCR
expr -> DECR VARIABLE
expr -> VARIABLE DECR
expr -> VARIABLE '(' arg_list ')'
expr -> VARIABLE '=' expr
expr -> VARIABLE PLUS_EQ expr
expr -> VARIABLE MINUS_EQ expr
expr -> VARIABLE MUL_EQ expr
```

```
expr -> VARIABLE DIV_EQ expr
expr -> VARIABLE MOD_EQ expr
expr -> VARIABLE SH_LE_EQ expr
expr -> VARIABLE SH_RI_EQ expr
expr -> VARIABLE AND_EQ expr
expr -> VARIABLE OR_EQ expr
expr -> VARIABLE XOR_EQ expr
expr -> '!' expr
expr -> '~' expr
expr -> '+' expr
expr -> '-' expr
expr -> expr OR expr
expr -> expr AND expr
expr -> expr '|' expr
expr -> expr '^' expr
expr -> expr '&' expr
expr -> expr '+' expr
expr -> expr '-' expr
expr -> expr '*' expr
expr -> expr '/' expr
expr -> expr '%' expr
expr -> expr '<' expr
expr -> expr '>' expr
expr -> expr GE expr
expr -> expr LE expr
expr -> expr NE expr
expr -> expr EQ expr
expr -> expr SHIFT_LEFT expr
expr -> expr SHIFT_RIGHT expr
expr -> '(' expr ')'
```

Quadruples

- We are using stack based quadruples.
- Quadruples are concerned with the data types of variables and expressions.
  Ex: PUSH_INT: pushes integer to stack.

- Quadruples can be concluded into four main categories.
  a) PUSH_(type): pushes a variable or constant to the top of the stack.
  b) POP_(type): pops a variable from the top of the stack.
  c) OPR_(type): it could be a unary or binary operation, that pops last pushed values in the stack and does the operation, then pushes the result of the operation to the top of the stack.
  d) (type)_TO_(type): pops the last pushed value from stack and converts its type, then pushes the value after conversion to the top of the stack.
- Let the first element on the top of the stack is x_1, second element on the top of the stack under x_1 is x_2.

Quadruple	Description
`PUSH_(type)` x_1	Pushes variable x_1 (or constant) on the top of the stack.
`POP_(type)` x_1	pops a variable from the top of the stack to variable x_1.
`PRINT_(type)`	pops x_1 from the top of the stack and print it in the console.
`(type1)_TO_(type2)`	x_1 = type2(x_1)
`ADD_(type)`	$x_1 = x_1 + x_2$
`MUL_(type)`	$x_1 = x_1 * x_2$
`DIV_(type)`	$x_1 = x_2 / x_1$
`SUB_(type)`	$x_1 = x_2 - x_1$
`MOD_(type)`	$x_1 = x_2 \% x_1$
`OR_BOOL`	$x_1 = x_1 \| \| x_2$
`AND_BOOL`	$x_1 = x_1 \&\& x_2$
`NOT_BOOL`	$x_1 = !x_1$

`BIT_AND_(type)`	$x_1 = x_1 \,\&\, x_2$
`BIT_OR_(type)`	$x_1 = x_1 \mid x_2$
`BIT_XOR_(type)`	$x_1 = x_1 \,\hat{}\, x_2$
`SHL_(type)`	$x_1 = x_2 << x_1$
`SHR_(type)`	$x_1 = x_2 >> x_1$
`GT_(type)`	$x_1 = x_2 > x_1$
`LT_(type)`	$x_1 = x_2 < x_1$
`GE_(type)`	$x_1 = x_2 \geq x_1$
`LE_(type)`	$x_1 = x_2 \leq x_1$
`EQ_(type)`	$x_1 = x_1 == x_2$
`NE_(type)`	$x_1 = x_1 \neq x_2$
`LE_(type)`	$x_1 = x_2 \leq x_1$
`NEG_(type)`	$x_1 = -x_1$
`INC_(type)`	$x_1 = x_1 + 1$
`DEC_(type)`	$x_1 = x_1 - 1$
`PROC (function name)`	Refer to the beginning of function scope.
`ENDP (function name)`	Refer to the end of function scope.
`CALL (function name)`	Call a procedure(function).
`RETURN`	Returns from a procedure(function).
`JMP (label)`	Unconditional jump to the label.
`JZ  (label)`	Jumps to label if the last value in stack x_1 is equal to zero.
`JNZ (label)`	Jumps to label if the last value in stack x_1 is not equal to zero.