Lab Terminal

Report

Group Members
Ahmad Raza (FA20-BCS-001)
M.Muzammil Hayat (FA20-BCS-055)

Subject
Compiler Construction

Instructor
Mr. Bilal Haider Bukhari

Question # 1

Introduction

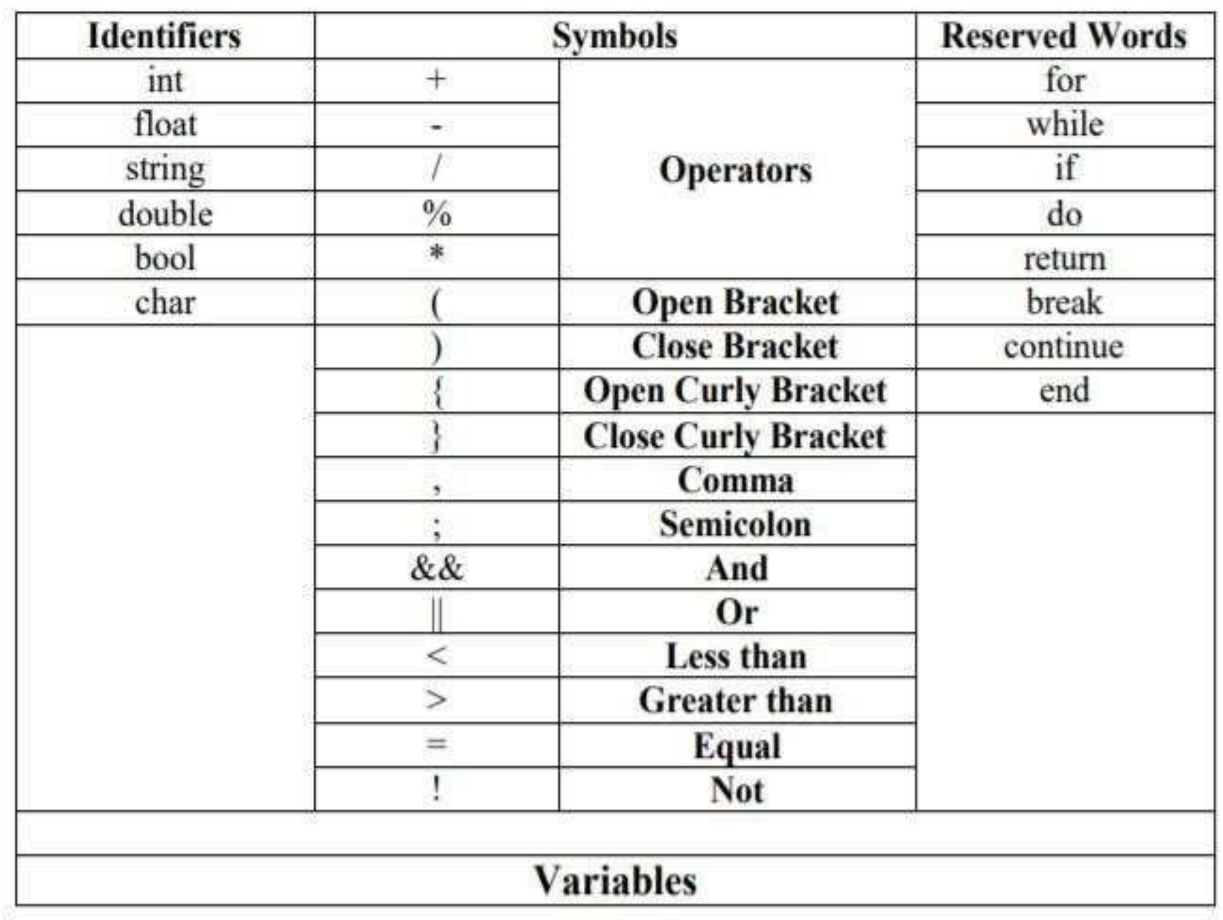
Our Compiler project comprises three integral components: the Lexical Analyzer, Syntax Analyzer, and Semantic Analyzer. The Lexical Analyzer, or scanner, initiates the process by converting the input program into a sequence of tokens using regular expressions. Following this, the Syntax Analyzer, or parser, scrutinizes the syntactical structure of the program to ensure adherence to the language's syntax. It constructs a parse tree based on a predefined grammar and the token sequence provided by the Lexical Analyzer. Finally, the Semantic Analyzer evaluates the program's declarations and statements, verifying their semantic correctness in terms of meaning, control structures, and data types. Together, these phases ensure a comprehensive analysis of input programs, encompassing both syntactic and semantic aspects for accurate compilation.

LEXICAL ANALYZER

Lexical Analysis is the first phase of compiler also known as scanner. It converts the input program into a sequence of Tokens.It can be implemented with the regular expressions.

SYNTAX ANALYZER

Syntax Analysis or Parsing is the second phase,i.e. after lexical analysis. It checks the syntactical structure of the given input,i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not. The parse tree is constructed by using the pre-defined Grammar of the language and the input string.

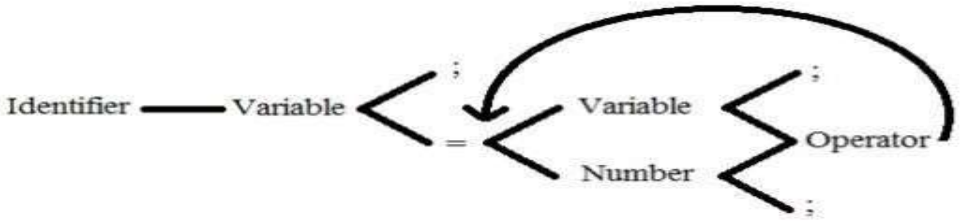


Identifiers	Symbols		Reserved Words
int	+	Operators	for
float	-		while
string	/		if
double	%		do
bool	*		return
char	(	Open Bracket	break
	)	Close Bracket	continue
	{	Open Curly Bracket	end
	}	Close Curly Bracket	
	,	Comma	
	;	Semicolon	
	&&	And	
	||	Or	
	<	Less than	
	>	Greater than	
	=	Equal	
	!	Not	
Variables			

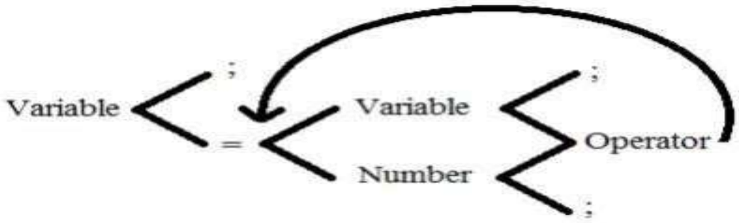
SEMANTIC ANALYZER

Semantic analysis is the task of ensuring that the declarations and statements of a program are semantically correct, i.e,that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.

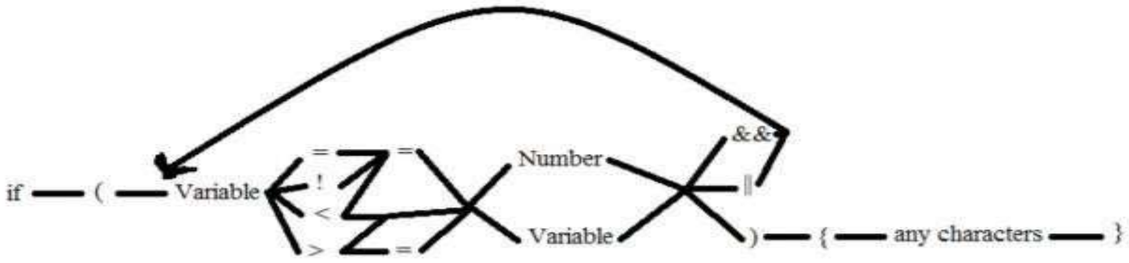
1.


Identifier
Variable
;
=
Variable
Number
;
Operator
;

2.


Variable
;
=
Variable
Number
;
Operator
;

3.

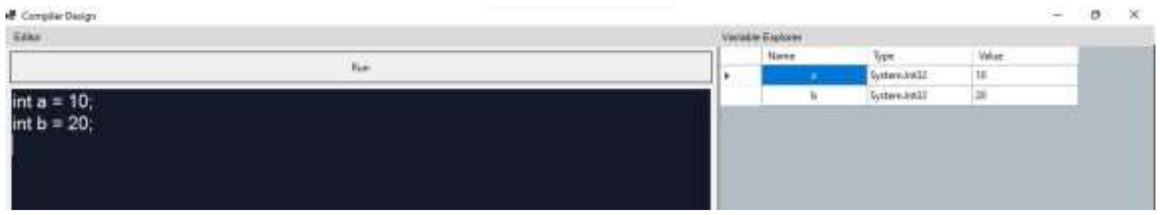

if
(
Variable
=
!
<
>
=
=
Number
Variable
&&
||
)
{
any characters
}

Question # 2

Lexical Analyzer

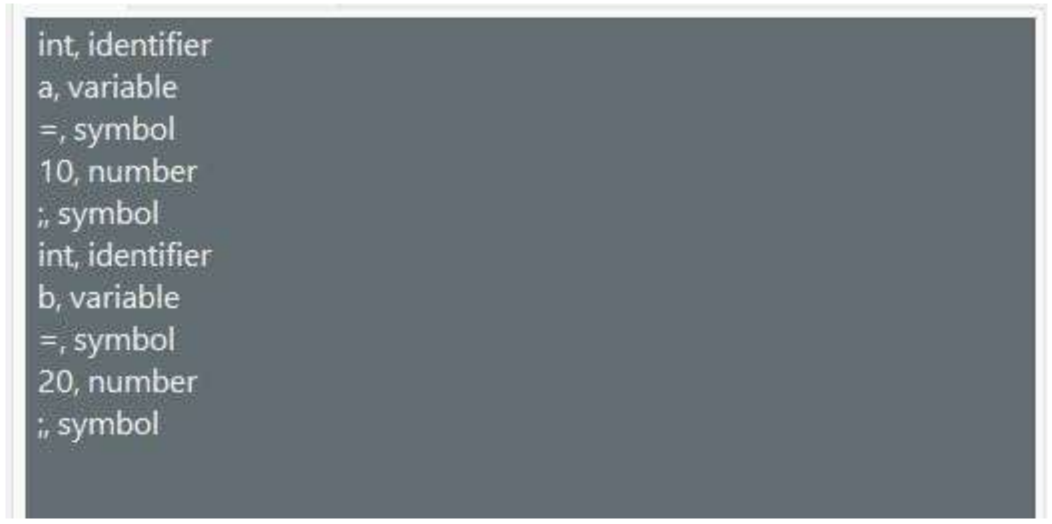
Lexical Analysis is the first phase of compiler also known as scanner. It converts the input program into a sequence of Tokens.It can be implemented with the regular expressions.

The Lexical Analyzer's key role is the identification and classification of language constructs through regular expressions, resulting in a stream of tokens that forms the basis for subsequent phases.

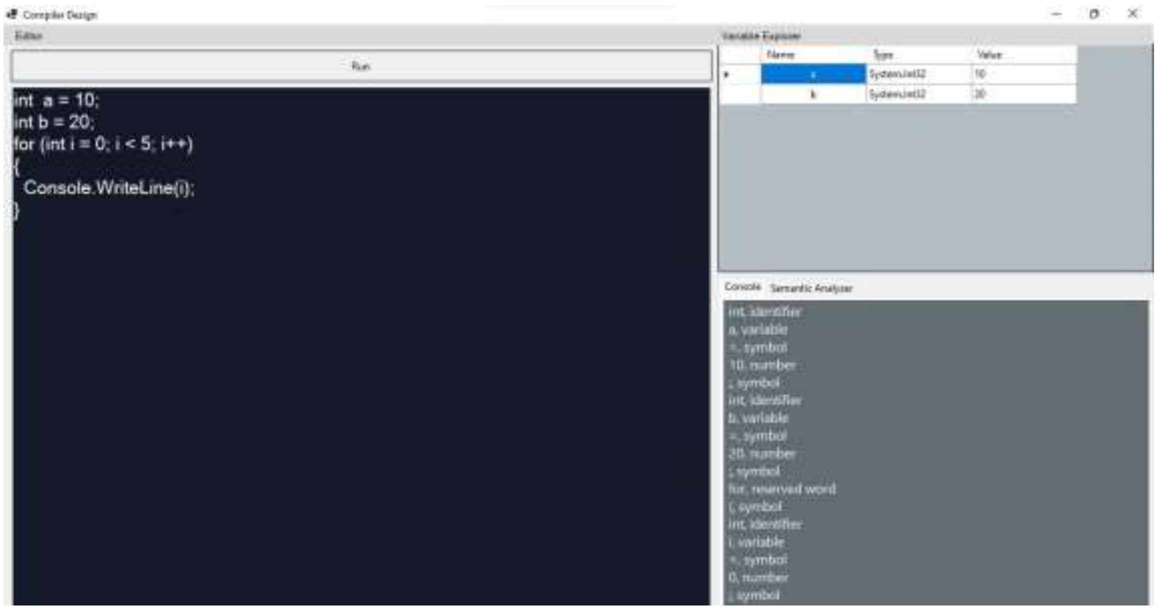


```
int a = 10;
int b = 20;
```

	Name	Type	Value
▸	a	System.Int32	10
	b	System.Int32	20



```
int, identifier
a, variable
=, symbol
10, number
;, symbol
int, identifier
b, variable
=, symbol
20, number
;, symbol
```



```
int  a = 10;
int b = 20;
for (int i = 0; i < 5; i++)
{
  Console.WriteLine(i);
}
```

	Name	Type	Value
▸	a	System.Int32	10
	b	System.Int32	20

```
int, identifier
a, variable
=, symbol
10, number
;, symbol
int, identifier
b, variable
=, symbol
20, number
;, symbol
for, reserved word
(, symbol
int, identifier
i, variable
=, symbol
0, number
;, symbol
```

Token Types Enumeration:

Defines an enumeration TokenType to represent different types of tokens (e.g., Keyword, Identifier, Operator, Constant).

```
enum TokenType
{
    Keyword,
    Identifier,
    Operator,
    Constant,



    // Add more as needed
}
```

Token Structure:

Defines a Token structure to hold information about each token, including its type and value.

```
struct Token
```

```
{ public TokenType Type; public string
    Value;
```

```
}
```

LexicalAnalyzer Class:

Creates a class named LexicalAnalyzer to encapsulate the functionality.

```
class LexicalAnalyzer
{
    // ...
}
```

Analyze Method:

- The Analyze method takes the source code as input and returns a list of tokens.
- Initializes a list to store the tokens.
- Defines regular expressions for various token patterns (e.g., keywords, identifiers, operators, constants).

```
static List<Token> Analyze(string sourceCode)
{
    List<Token> tokens = new List<Token>();

    // Regular expressions for token patterns
    Regex keywordRegex = new Regex(@"¥b(if|else|while|int|float)¥b");



    Regex identifierRegex = new Regex(@"[a-zA-Z_][a-zA-Z0-9_]*");
    Regex operatorRegex = new Regex(@"[+¥-*/=]");
    Regex constantRegex = new Regex(@"¥b¥d+(¥.¥d+)?¥b");

   }
```

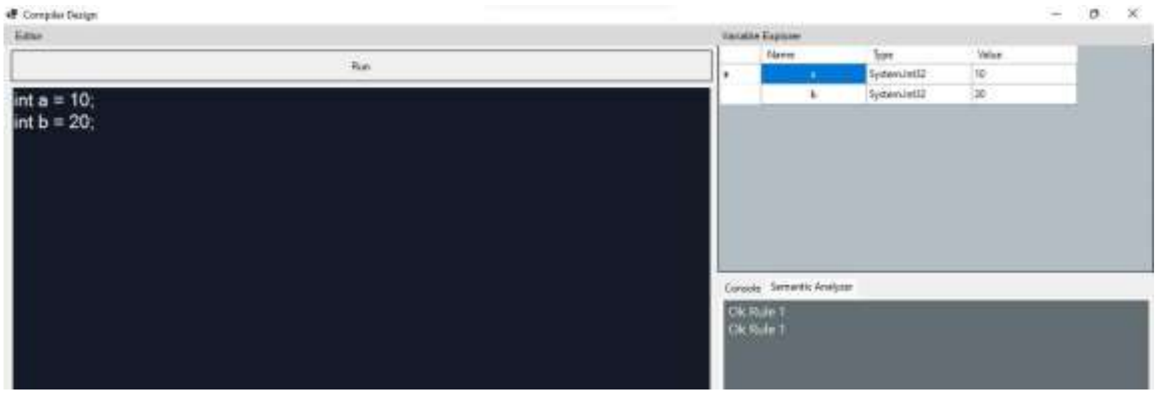
Tokenization Loop:

- Uses a StringReader to iterate through each character in the source code.
- Checks each character against the defined regular expressions to identify token patterns.
- Adds recognized tokens to the list.

```
using (StringReader reader = new StringReader(sourceCode))
{ int currentChar;
    while ((currentChar = reader.Read()) != -1)
    { char currentCharValue = (char)currentChar.ToString();

        // Token matching
        if (char.IsWhiteSpace(currentCharValue))
        {
            // Skip whitespace
        }
        else if (keywordRegex.IsMatch(currentCharValue.ToString()))
        { tokens.Add(new Token { Type = TokenType.Keyword, Value =
currentCharValue.ToString() });
        }
        else if (identifierRegex.IsMatch(currentCharValue.ToString()))
        { tokens.Add(new Token { Type = TokenType.Identifier, Value =
currentCharValue.ToString() });
        }
        else if (operatorRegex.IsMatch(currentCharValue.ToString()))



        { tokens.Add(new Token { Type = TokenType.Operator, Value =
currentCharValue.ToString() });
        }
        else if (constantRegex.IsMatch(currentCharValue.ToString()))
        {

            tokens.Add(new Token { Type = TokenType.Constant, Value =
currentCharValue.ToString() });
        }



        // Add more token types and patterns as needed
        else {



            // Handle unrecognized characters or tokens
            Console.WriteLine($"Error: Unrecognized character '{currentCharValue}'");
        }
    }
}
```

Semantic analysis

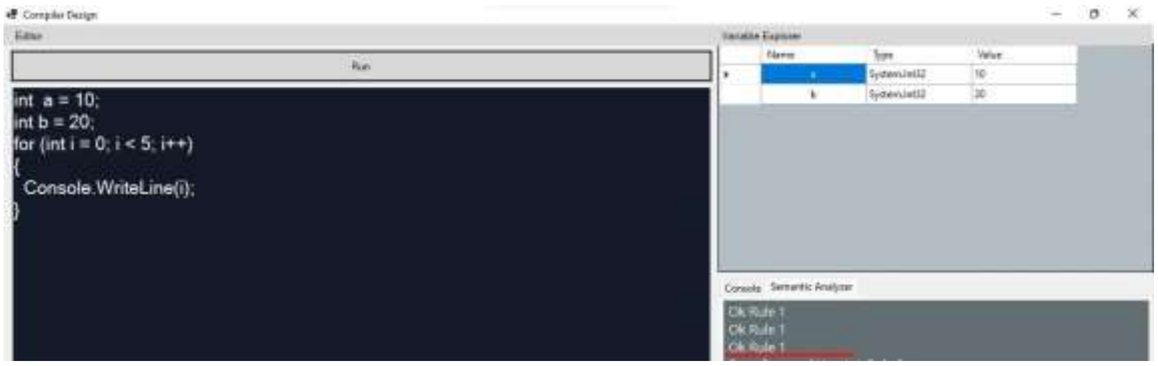
Semantic analysis is the task of ensuring that the declarations and statements of a program are semantically correct, i.e,that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.



```
int a = 10;
int b = 20;
```

	Name	Type	Value
▸	a	System.Int32	10
	b	System.Int32	20

```
Ok Rule 1
Ok Rule 1
```



```
int  a = 10;
int b = 20;
for (int i = 0; i < 5; i++)
{
  Console.WriteLine(i);
}
```

	Name	Type	Value
▸	a	System.Int32	10
	b	System.Int32	20

```
Ok Rule 1
Ok Rule 1
Ok Rule 1
```

Namespace and Class Definition:

The code defines a class named SemanticAnalyzer within the System namespace.

```
using System; using
System.Collections.Generic; class
SemanticAnalyzer
```

```
{ // ...
```

```
}
```

Symbol Table:

A dictionary named symbolTable is declared to store variable symbols along with their types.

```
Dictionary<string, string> symbolTable = new Dictionary<string, string>();
Token Structure:
A Token structure is defined to represent tokens, where each token has a type and a
value. csharp Copy code struct Token
{ public string Type;
    public string
    Value;
}
```

AnalyzeSemantics Method:

This method performs semantic analysis on a list of tokens.

It iterates through each token, checking for specific semantic rules.

For example, if a token is a keyword "int," it calls DeclareVariable to handle the declaration of an integer variable.

If an identifier is encountered and it is not found in the symbol table, it reports an error for an undeclared variable.

```
static void AnalyzeSemantics(List<Token> tokens)



{ foreach (Token token in tokens)
    { if (token.Type == "Keyword" && token.Value == "int")
        {
            DeclareVariable(tokens);
        }
        else if (token.Type == "Identifier" && !symbolTable.ContainsKey(token.Value))
        {
            Console.WriteLine($"Error: Undeclared variable '{token.Value}'");
        }




            }
}
```

DeclareVariable Method:

This method handles the declaration of integer variables.

It assumes that the next token is an identifier and adds the variable to the symbol table.

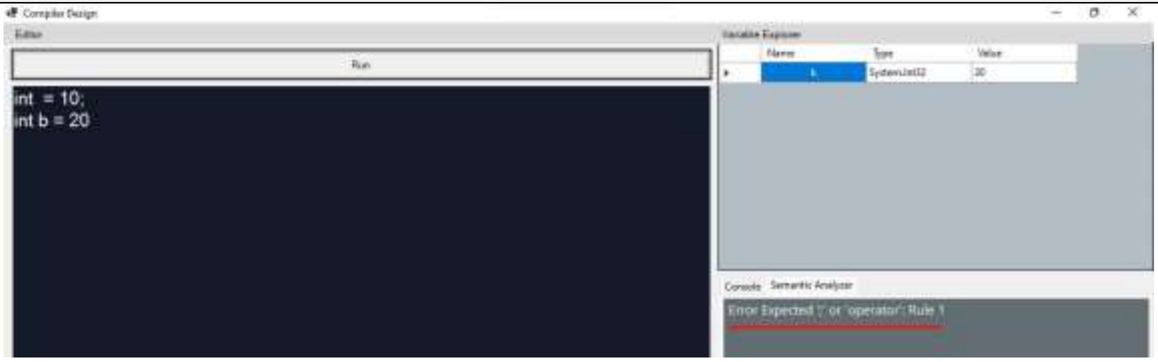
```
static void DeclareVariable(List<Token> tokens)



{
    Token nextToken = GetNextToken(tokens);
    if (nextToken.Type == "Identifier")
    { symbolTable[nextToken.Value] = "int";
```

```
    }
    else
    {
        Console.WriteLine($"Error: Expected identifier after 'int', but found
'{nextToken.Value}'");
    }
}
```

GetNextToken Method:

This method simulates obtaining the next token from the list of tokens for processing.

```
static Token GetNextToken(List<Token> tokens)
{
    Token nextToken = tokens[0];
    tokens.RemoveAt(0);
    return nextToken;
}
```



```
int  = 10;
int b = 20
```

	Name	Type	Value
▸	b	System.Int32	20

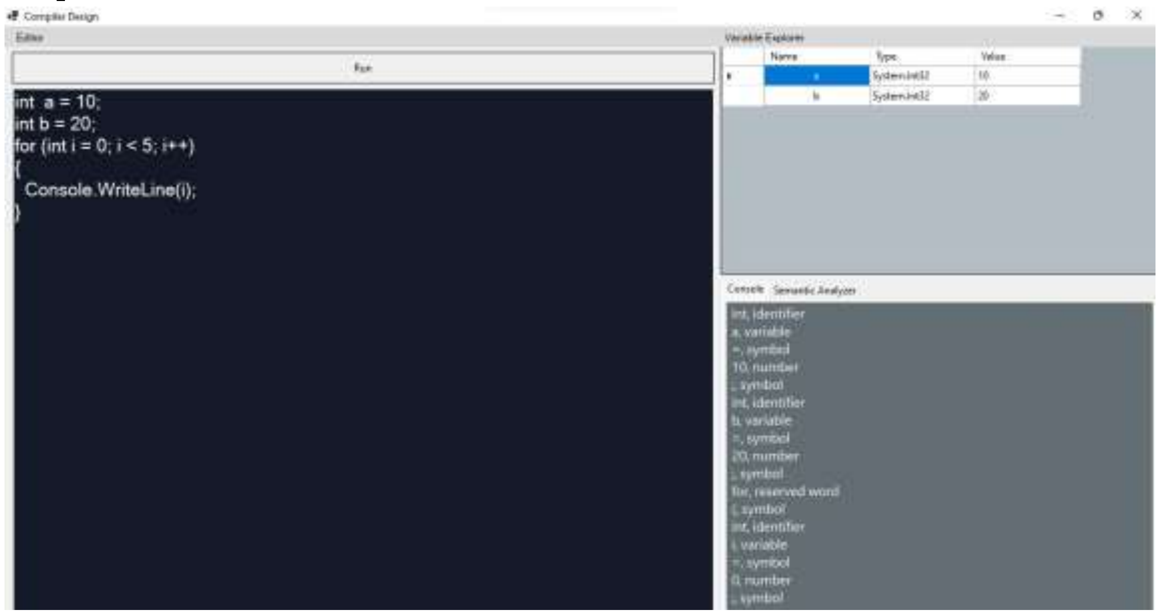
```
Error Expected ';' or 'operator': Rule 1
```

Question #3

Input

```
int a = 10; int
b = 20;
for (int i = 0; i < 5; i++)
{
  Console.WriteLine(i);
}
```

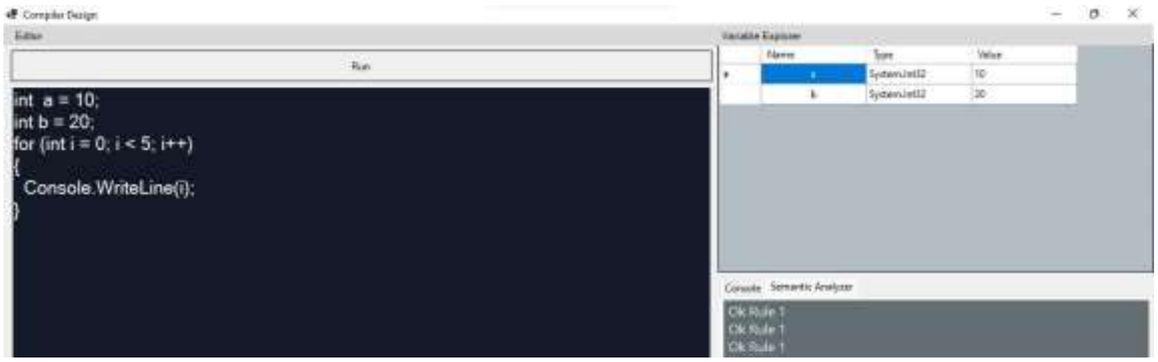
Output



```
int  a = 10;
int b = 20;
for (int i = 0; i < 5; i++)
{
  Console.WriteLine(i);
}
```

	Name	Type	Value
▸	a	System.Int32	10
	b	System.Int32	20

```
int, identifier
a, variable
=, symbol
10, number
;, symbol
int, identifier
b, variable
=, symbol
20, number
;, symbol
for, reserved word
(, symbol
int, identifier
i, variable
=, symbol
0, number
;, symbol
```



```
int  a = 10;
int b = 20;
for (int i = 0; i < 5; i++)
{
  Console.WriteLine(i);
}
```

Name	Type	Value
a	System.Int32	10
b	System.Int32	20

```
Ok Rule 1
Ok Rule 1
Ok Rule 1
```

Code which is responsible for lexical analsis

```
bool isPunctuator(char ch)    //check if the given character is a punctuator
or not
{
    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}' ||
        ch == '&' || ch == '|')
        { return true;
        } return
    false;
}
bool validIdentifier(char* str)       //check if the given identifier is valid
or not
{
    if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
        str[0] == '3' || str[0] == '4' || str[0] == '5' ||
        str[0] == '6' || str[0] == '7' || str[0] == '8' ||
        str[0] == '9' || isPunctuator(str[0]) == true)
        { return false;
        }      //if first character of string is a digit or a special character,
identifier is not valid int i,len = strlen(str);
    if (len == 1)
    { return true;
    }  //if length is one, validation is already completed, hence return true
else
    {
    for (i = 1 ; i < len ; i++)       //identifier cannot contain special
characters
    { if (isPunctuator(str[i]) == true)
        { return false;
        }
    }
    }
    return true;
}
bool isOperator(char ch)      //check if the given character is an operator or
not
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=' || ch == '|' || ch == '&')
    { return true;
    }
    return false;
}
bool isKeyword(char *str)     //check if the given substring is a keyword or
not
{ if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") || !strcmp(str, "continue")
        || !strcmp(str, "int") || !strcmp(str, "double")
        || !strcmp(str, "float") || !strcmp(str, "return")
        || !strcmp(str, "char") || !strcmp(str, "case")
        || !strcmp(str, "long") || !strcmp(str, "short")
        || !strcmp(str, "typedef") || !strcmp(str, "switch")
        || !strcmp(str, "unsigned") || !strcmp(str, "void")
        || !strcmp(str, "static") || !strcmp(str, "struct")
        || !strcmp(str, "sizeof") || !strcmp(str,"long")
        || !strcmp(str, "volatile") || !strcmp(str, "typedef")
        || !strcmp(str, "enum") || !strcmp(str, "const")
        || !strcmp(str, "union") || !strcmp(str, "extern")
        || !strcmp(str,"bool"))
        { return true;
        }
    else
    { return false;
    }
}
bool isNumber(char* str)      //check if the given substring is a number or not
{ int i, len = strlen(str),numOfDecimal = 0;
    if (len == 0)
    { return false;
    }
    for (i = 0 ; i < len ; i++)
    { if (numOfDecimal > 1 && str[i] == '.')
        { return false;
        } else if (numOfDecimal <= 1)
        { numOfDecimal++;
        }
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' || (str[i] == '-' && i > 0))
            { return false;
            }
    }
    return true;
}

char* subString(char* realStr, int l, int r) //extract the required substring
from the main string
{ int i; char* str = (char*) malloc(sizeof(char) * (r - l +
    2));

    for (i = l; i <= r; i++)
    { str[i - l] = realStr[i];
        str[r - l + 1] = '¥0';
    }
    return str;
}


void parse(char* str)                                    //parse the expression
{
    int left = 0, right = 0;
    int len = strlen(str);
    while (right <= len && left <= right) { if (isPunctuator(str[right]) == false)//if
character is a digit or an alphabet { right++; }

        if (isPunctuator(str[right]) == true && left == right)      //if character is a
punctuator {
            if (isOperator(str[right]) == true)
            { std::cout<< str[right] <<" IS AN OPERATOR¥n";
            } right++;
            left = right;
            } else if (isPunctuator(str[right]) == true && left != right
|| (right == len && left != right))//check if
parsed substring is a keyword or identifier or number
            { char* sub = subString(str, left, right - 1); //extract substring

            if (isKeyword(sub) == true) { cout<< sub <<" IS
                        A KEYWORD¥n"; }
            else if (isNumber(sub) == true) { cout<< sub
                        <<" IS A NUMBER¥n"; }
            else if (validIdentifier(sub) == true
                     && isPunctuator(str[right - 1]) == false)
                     { cout<< sub <<" IS A VALID IDENTIFIER¥n";
                     }
            else if (validIdentifier(sub) == false
                     && isPunctuator(str[right - 1]) == false)
                     { cout<< sub <<" IS NOT A VALID IDENTIFIER¥n"; }

            left = right;
            }
    }
    return;
}
```

Question # 4

Lexical Analyzer

- **Input Source Code**
- **Tokenization:** The source code is divided into meaningful units called tokens. (keywords, identifiers, operators, constants, and symbols)
- **Regular Expressions and Patterns:** The Lexical Analyzer uses regular expressions or patterns to define the rules for recognizing different types of tokens.
- **Scanning:** scans the source code character by character.
- **Recognizing Tokens:**
- **Building Tokens:** When a token is recognized, the Lexical Analyzer builds the token by collecting the characters that form it.
- **Ignoring Whitespace and Comments:** The Lexical Analyzer typically ignores whitespace characters
- **Output Token Stream:** The Lexical Analyzer produces a stream of tokens as its output.
- **Passing Tokens to the Syntax Analyzer:** The generated token stream is passed to the next phase of the compiler, the Syntax Analyzer (Parser).

Semantic analysis

- **Input:** Semantic analysis takes the output of the lexical and syntax analysis phases as its input.
- **Symbol Table Construction:** The compiler constructs a symbol table, a data structure that stores information about variables, functions, and other program entities. This table keeps track of the names, types, and scopes of these entities.

- **Type Checking:** it checks that you're not trying to add a string to an integer.
- **Scope Analysis:** The compiler checks for variable scope violations.
- **Declaration Checking:** Semantic analysis verifies that variables and functions are declared before they are used.

- **Function Overloading and Signature Matching:** If the programming language supports function overloading, the compiler checks that functions with the same name have different parameter lists.
- **Constant Folding:** This involves evaluating constant expressions at compile-time to optimize the code.
- **Error Reporting:** If semantic errors are detected, such as type mismatches, undeclared variables, or scope violations, the compiler reports these errors to the user.
- **Intermediate Code Generation:** Depending on the compiler design, semantic analysis may also involve generating intermediate code.
- **Output:** The output of the semantic analysis phase is either an error report indicating the presence and nature of semantic errors or, in the absence of errors, an enhanced representation of the program that captures its semantic meaning.

Question # 5

Challenges Faces

- Managing complex grammars and intricate rules to ensure accurate set calculations.
- Debugging to address potential errors in the code and ensure seamless functionality.
- Coordinating variable scopes and dependencies between different functions within the project.
- Rigorous testing to validate the accuracy of tokenization and set calculations.