Requirements

You are to develop a l exer, parser, i nterpreter and an I LOC i nstructions gen-erator for a s mall programming l anguage - *MniLang*. The s pecification of this l anguage can be f ound i n the next sections. The assignment i s composed of three major parts: i) a FSA-based l exer and hand-crafted top-down recursive-descent parser, i i) visitor classes to perform XML generation, s emantic analysis and i nterpreter execution on the abstract s yntax tree (AST) produced by the parser and i ii) a report detailing how you designed and i mplemented the different tasks. These parts are s ubdivided in tasks and explained i n detail i n the Task Breakdown s ection.

MiniLang is an expression-based strongly-typed programming language. The language has C-style comments, that is, `//...` for line comments and `/*...*/` for block comments. The language is case-sensitive and each function is expected to return a value. MiniLang has 3 types: 'float', 'int' and 'bool'. Binary operators, such as '+', require that the operands have matching types and the language does not perform any implicit/automatic typecast.

The following is a syntactically and semantically correct MiniLang program:

```
fn Square(x:float) : float {
 return x*x;
}

fn XGreaterThanY(x:float, y:float) : bool {
 var ans:bool = true;
 if (y > x) { ans = false; }
 return ans;
}

fn AverageOfThree(x:float, y:float, z:float) : float {
  var total:float = x + y + z;
  return total / 3;
}

var x:float = 2.4;
var y:float = Square(2.5);
print y;                                  //6.25
print XGreaterThanY(x, 2.3);              //true
print XGreaterThanY(Square(1.5), y);      //false
print AverageOfThree(x, y, 1.2);          //3.28
```

MiniLang (in EBNF)

```
⟨Letter⟩             ::= [A-Za-z]

⟨Digit⟩              ::= [0-9]

⟨Printable⟩          ::= [\x20-\x7E]

⟨Type⟩               ::= 'float' | 'int' | 'bool'

⟨BooleanLiteral⟩     ::= 'true' | 'false'

⟨IntegerLiteral⟩     ::= ⟨Digit⟩ { ⟨Digit⟩ }

⟨FloatLiteral⟩       ::= ⟨Digit⟩ { ⟨Digit⟩ } '.' ⟨Digit⟩ { ⟨Digit⟩ }

⟨Literal⟩            ::= ⟨BooleanLiteral⟩
                      |  ⟨IntegerLiteral⟩
                      |  ⟨FloatLiteral⟩

⟨Identifier⟩         ::= ( '_' | ⟨Letter⟩ ) { '_' | ⟨Letter⟩ | ⟨Digit⟩ }

⟨MultiplicativeOp⟩   ::= '*' | '/' | 'and'

⟨AdditiveOp⟩         ::= '+' | '-' | 'or'

⟨RelationalOp⟩       ::= '<' | '>' | '==' | '!=' | '<=' | '>='

⟨ActualParams⟩       ::= ⟨Expression⟩ { ',' ⟨Expression⟩ }

⟨FunctionCall⟩       ::= ⟨Identifier⟩ '(' [ ⟨ActualParams⟩ ] ')'

⟨SubExpression⟩      ::= '(' ⟨Expression⟩ ')'

⟨Unary⟩              ::= ( '-' | 'not' ) ⟨Expression⟩

⟨Factor⟩             ::= ⟨Literal⟩
                      |  ⟨Identifier⟩
                      |  ⟨FunctionCall⟩
                      |  ⟨SubExpression⟩
                      |  ⟨Unary⟩

⟨Term⟩               ::= ⟨Factor⟩ { ⟨MultiplicativeOp⟩ ⟨Factor⟩ }

⟨SimpleExpression⟩   ::= ⟨Term⟩ { ⟨AdditiveOp⟩ ⟨Term⟩ }

⟨Expression⟩         ::= ⟨SimpleExpression⟩ { ⟨RelationalOp⟩ ⟨SimpleExpression⟩ }

⟨Assignment⟩         ::= ⟨Identifier⟩ '=' ⟨Expression⟩

⟨VariableDecl⟩       ::= 'var' ⟨Identifier⟩ ':' ⟨Type⟩ '=' ⟨Expression⟩

⟨PrintStatement⟩     ::= 'print' ⟨Expression⟩

⟨ReturnStatement⟩    ::= 'return' ⟨Expression⟩

⟨IfStatement⟩        ::= 'if' '(' ⟨Expression⟩ ')' ⟨Block⟩ [ 'else' ⟨Block⟩ ]
```

```
⟨ForStatement⟩    ::= ‘for’ ‘(’ [ ⟨VariableDecl⟩ ] ’;’ ⟨Expression⟩ ’;’ [ ⟨Assignment⟩ ] ‘)’ ⟨Block⟩

⟨FormalParam⟩     ::= ⟨Identifier⟩ ‘:’ ⟨Type⟩

⟨FormalParams⟩    ::= ⟨FormalParam⟩ { ‘,’ ⟨FormalParam⟩ }

⟨FunctionDecl⟩    ::= ‘fn’ ⟨Identifier⟩ ‘(’ [ ⟨FormalParams⟩ ] ‘)’ ‘:’ ⟨Type⟩ ⟨Block⟩

⟨Statement⟩       ::= ⟨VariableDecl⟩ ‘;’
                    | ⟨Assignment⟩ ‘;’
                    | ⟨PrintStatement⟩ ‘;’
                    | ⟨IfStatement⟩
                    | ⟨ForStatement⟩
                    | ⟨ReturnStatement⟩ ‘;’
                    | ⟨FunctionDecl⟩
                    | ⟨Block⟩

⟨Block⟩           ::= ‘{’ { ⟨Statement⟩ } ‘}’

⟨Program⟩         ::= { ⟨Statement⟩ }
```

Additional Requirements:

- Line by line comments
- Full custom implementation
- 2000 word report about the implementation of each task(code snippets in report should be a screenshot)
- Deadline: 3rd May 2019

Task Breakdown

Task 1 - A table-driven lexer

In this first task you are to develop the lexer for the *MiniLang* language using the C++ programming language. The lexer is to be implemented using the table-driven approach, which encodes the DFA transition function of the *MiniLang* micro-syntax. The lexer should be able to report any lexical errors in the input program.

Task 2 - Hand-crafted recursive descent parser

In this task you are to develop a hand-crafted predictive parser for the *MiniLang* language using the C++ programming language. The Lexer and Parser classes interact through the function *GetNextToken*() which the parser uses to get the next valid token from the lexer. The parser should be able to report any syntax errors in the input program. A successful parse of the input should produce an abstract syntax tree (AST) describing the structure of the program.

Task 3 - AST XML Generation Pass

In OOP programming, the Visitor design pattern is used to describe an operation to be performed on the elements of an object structure without changing the classes on which it operates. In our case this object structure is the AST produced by the parser in Task 2. For this task you are to implement a visitor class to output a properly indented XML representation of the generated AST.

```
var x : float = 8.1 + 2.2;

<Decl>
    <Var Type="float">x</Id>
    <BinExprNode Op="+">
        <FloatConstant>8.1</FloatConstant>
        <FloatConstant>2.2</FloatConstant>
    </BinExprNode>
</Decl>
```

Task 4 - Semantic Analysis Pass

For this task, you are to implement another visitor class to traverse the AST and perform type-checking (e.g. checking that variables are assigned to appropriately typed expressions, variables are not declared multiple times in the same scope, etc.). In addition to the global program scope, scopes are created whenever a block is entered and destroyed when control leaves the block. Note that blocks may be nested and that to carry out this task, it is essential to have a proper implementation of a symbol table.

Task 5 - Interpreter Execution Pass

For this task, you are to implement another visitor class to traverse the AST which simulates an interpreter and executes the test program. The *'print'* <Expression> statement can be used in your test programs to output the value of <Expression> to the console and determine whether the computation carried out by the interpreter visitor is correct.

Report

In addition to the source and class files, you are to deliver a report for each Task. In your report include any deviations from the original EBNF, the salient points on how you developed the lexer / parser / interpreter (and reasons behind any decisions you took) including semantic rules and code execution, and any sample *MiniLang* programs you developed for testing the outcome of your compiler. In your report, state what you are testing for, insert the program AST and the outcome of your test. As an example, the *MiniLang* source script below, computes the answer of a real number raised to an integer power:

```
//Function definition for Power
fn Pow(x:float , n:int) : float
{
    var y : float = 1.0;          //Declare y and set it to 1.0
    if( n>0 )
    {
        for(; n>0 ; n=n-1)
        {
            y = y * x;            //Assignment y = y * x;
        }
    }
    else
    {
        for(; n<0 ; n=n+1)
        {
            y = y / x;            //Assignment y = y / x;
        }
    }
    return y;                     //return y as the result
}
```