FOOL Programming Language

Eduart Uzeir[1]

[1] *Dipartimento di Informatica - Scienze e Ingegneria. Email: eduart.uzeir@studio.unibo.it*

Abstract

In this report we show the steps needed to build a simple compiler for an educational programming language called Functional Object Oriented Language (FOOL). Nowadays very few people build compilers or interpreters for mainstream programming languages. On the other hand knowing the theory and the practice behind this discipline should be an important achievement on the life of a computer scientist. Not only because we can understand better the functions and the peculiarities of the languages that we use every day but we can be better developers by knowing all the processes that our code passes through until it become machine code and get executed by the hardware.

Keywords: Fool Programming Languages, ANTLR4, Compiler, Interpreter, Grammar, Assembly Code, C#.

1 Introduction

In the present work we describe the main steps needed to create a compiler for a toy programming language called FOOL. This project is developed as part of the exam *Compilers and Interpreters*. The grammar and some code snippets for the project are provided by the professor. You can see the grammar in the Appendix of this report.
The objective is to construct a compiler that given in input a file written using Fool syntax (.fool) to generate some *assembly code* in a given specific assembly language for a custom virtual machine called *Simple Virtual Machine, SVM*.
The first step of the development is to prepare your development tools and input files.

- **Programming Language:** C# version 8.0 [1].
- **Development IDE:** We have used Microsoft Visual Studio Community 2017 version:15.9.35[2] with Microsoft .NET Framework version 4.8.04084 to write, test and debug the project.
- **Lexer/Parser generator:** We used Antlr4.Runtime.Standard version: 4.8.0[3] to generate the following files: *FOOL.interp, FOOL.tokens, FOOLLexer.cs, FOOLLexer.interp, FOOLLexer.tokens*. This files are automatically generated by ANTLR4 using an input grammar file *FOOL.g4*.
- **FOOL.g4:** Is the input grammar used by ANTLR4 to generate the previous files.
- **Helper Package:** System.Configuration.ConfigurationManager version:v6.0.0[4] is used to configure the custom project directory paths.

Once prepared the environment and the tools we have to create the project structure and the input files generated by ANTLR4.
In the second step we produce the project directory structure as shown in fig. 1. and populate the folder *Lexer* with the ANTLR4 generated files [5]. In order to generate those files we have to use the following command from the console:

```
>java -jar antlr-4.8.0.jar -Dlanguage=Csharp FOOL.g4 -visitor
```

Compilers and Interpreters (2021)

Corresponding author
Eduart Uzeir

Edited by
Eduart Uzeir

1. https://docs.microsoft.com/it-it/dotnet/csharp/whats-new/csharp-8
2. https://visualstudio.microsoft.com/it/vs/older-downloads/#visual-studio-2017-and-other-products
3. https://www.nuget.org/packages/Antlr4.Runtime.Standard/4.8.0
4. https://www.nuget.org/packages/system.configuration.configurationmanager/
5. https://hayeol.tistory.com/45

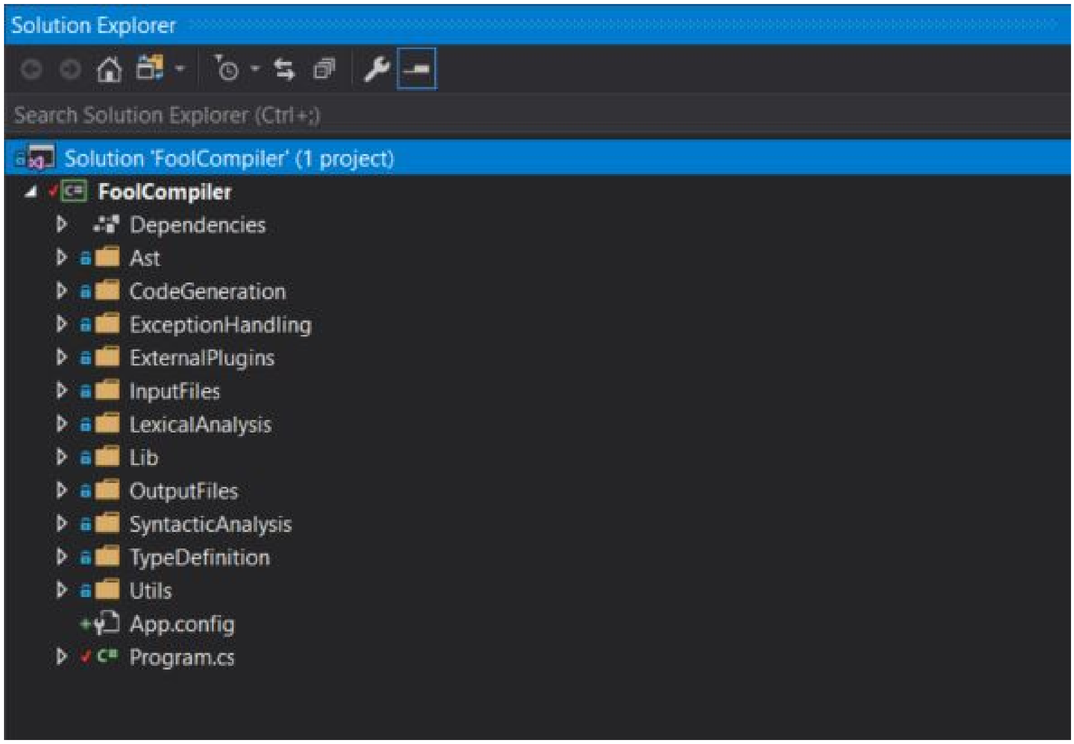

Figure 1. The folder structure of the Project

We have to copy the generated files in the folder *Lexer* of our project.

Now let's have a view in the folder structure of the project: Now according to .NET terminology the project consists of one *solution* calle **FoolCompiler** that contains only one *project* called Fool-Compiler as well. FoolCompiler project is populated by the following folders:

- **Abstract Syntax Tree (Ast)** contains 22 **.cs**. The first 20 are the implementation of all possible nodes used in the Ast. We have used the following naming convention: *Fool**NodeName**Node*, for example: *FoolMethodNode.cs* or *FoolProgramLetInExpressionNode.cs*. The remaining 2 files are the visitor implementation (*FoolVisitor.cs*) and the interface from which all the nodes inherit (*IFoolNode.cs*).
- **CodeGeneration** folder contains 16 files, some of them are classes that we have write others are files generated in the process of code generation. In this folder are placed all the code that guide the code generation process, the function of the heap and the dispatch table.
- **ExceptionHandling** contains the custom error handling mechanisms. This folder contains 8 files.
- **ExternalPlugins** contains the ANTLR4 jar used to generate the lexer and the parser starting from the grammar.
- **InputFiles** contains 42 .fool testing files that we have use to test the compiler.
- **LexicalAnalysis** contain 6 files, the grammar FOOL.g4 and 5 other files generated by ANTLR4.
- **Lib** is a library folder that contain only one helper class.
- **SyntacticAnalysis** contain 3 ANTLR4 generated files.
- **TypeDefinition** contain 9 files, each of them is the definition of one Fool type, such as *FoolInt-Type.cs, FoolBoolType.cs* etc. All this types inherit from the interface *IFoolType.cs*.
- **Utils** contain the utility classes and functions used in the whole project. This file contain 6 files.

Also in the root directory of the project we have 2 very important files:

- **Program.cs** is the starting point of the execution of the project. Here is where the *main* method is defined.

```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="inputFilePath" value="C:\Users\uzeir\Desktop\FoolCompiler\FoolCompiler\InputFiles\" />
    <add key="outputFilePath" value="C:\Users\uzeir\Desktop\FoolCompiler\FoolCompiler\OutputFiles\code.svm" />
  </appSettings>
</configuration>
```

Figure 2. Web.config

- **App.config** is the file that we use to set the absolute path to the input and output directories on your local machine. This is needed in order to avoid hard-coding of all the paths in the code. Fig. 2. show the format of this file.

2 How to run the project

The project can be download from the Github[6] repository and executed locally. Here is a step by step tutorial how to run the project in Windows using Visual Studio 2017. In case you want to run the project using Linux you can use Mono[7] or install manually .NET[8] in your computer.
Image 3 shows a possible output when the execution terminates.

- **Step 1:** Download and unzip the project folder form GitHub.
- **Step 2:** Open the folder and modify *Web.config* with your own paths.
- **Step 3:** Open the project in the IDE of your choice .
- **Step 4:** Run the project.
- **Step 5:** Choose the input file from the default folder or create your own input file.
- **Step 6:** Click *Enter* to execute the project.

3 Lexical Analysis - Lexer

The whole process of code generation, starting from the input file pass through a number of well defined phases. *Lexical Analysis* or *Lexer* is the first of those steps. In our case the Lexer is generated automatically by the *ANTLR4 tool*. The following listing shows the piece of code in the *Program.cs* that implement the creation and the use of the lexer.
The job of the Lexer is to consume the input file written in the FOOL language and to produce a set of **Tokens**, one for each lexeme in the original input. The Lexer is also called a *tokenizer*.

```csharp
//LEXER
Console.WriteLine("[<<<<< LEXICAL ANALYSIS >>>>>]");
FOOLLexer lexer = new FOOLLexer(input);
if (lexer.lexicalErrors.Count > 0)
    {
        foreach (var e in lexer.lexicalErrors)
            {
                throw new FoolLexerException(e);
            }
    }
CommonTokenStream tokens = new CommonTokenStream(lexer);
Console.WriteLine("\nOK, DONE!");
Console.WriteLine("****************************");
```

Fig. 4. show an example of tokenization for the input code 437 + 743.
In order to generate the tokens, Lexer make use of the input grammar (FOOL.g4) where we have

6. https://github.com/euzeir/FoolCompiler
7. https://www.mono-project.com/
8. https://opensource.com/article/17/11/net-linux

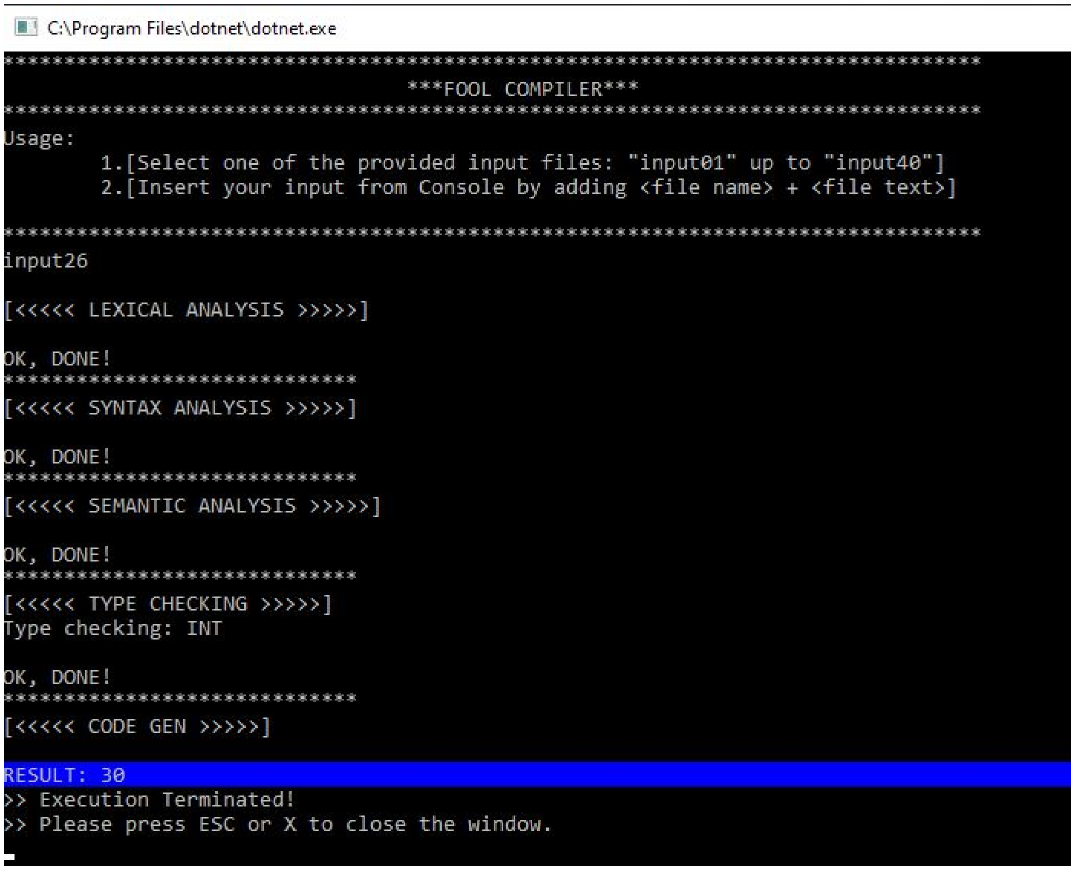

Figure 3. Example of Execution

initialize all the needed rules (parser and lexer rules).
We want to point out the fact that the rules are divided in 2 classes, the parser rules that comes first in the grammar and the lexer rules that comes later. Even inside each class the order of the rules is important and helps in case of ambiguity. Another distinction is that lexer rule names are in uppercase letters, and usually expressed in the form of regular expressions or one-to-one matching. Our lexer has also specific rules that ignore comments and white spaces. This code inform the lexer how to generate the tokens and in the case of errors to report. As you can see in the previous code listing the errors are added in a list called *lexicalErrors* and if this list is empty we proceed with the following steps.
The part of lexical rules in the grammar is shown in the Appendix.

```
exp    : left=term ((PLUS | MINUS) right=exp)?
       ;
//some lines further

PLUS    : '+' ;

//Numbers
fragment DIGIT : '0'..'9';
INTEGER        : DIGIT+;
```

Once the input files is tokenized and the error list is empty we pass to the second phase, the *Parsing*.

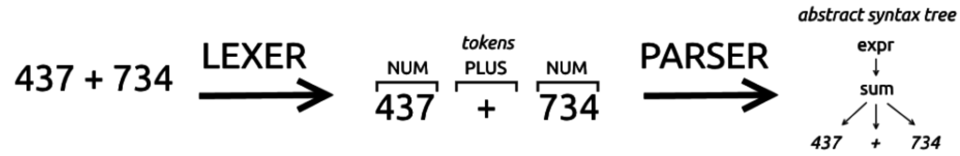

Figure 4. Example of Tokenization and Parsing

4 Syntax Analysis - Parser

Parsing is the second step in the process of input analysis. It is automatically performed by the ANTLR4 plugin and has as final result a tree representation of the input called *Abstract Syntax Tree* or *AST*.
The snippet of code shows the creation on the parser and it's use in out *Program.cs* file

```
//PARSER
Console.WriteLine("[<<<<< SYNTAX ANALYSIS >>>>>]");
FOOLParser parser = new FOOLParser(tokens);
FOOLParser.ProgContext progContext = parser.prog();
if (parser.NumberOfSyntaxErrors > 0)
throw new FoolParserException("Errors: " + parser.NumberOfSyntaxErrors + "\n");
Console.WriteLine("\nOK, DONE!");
Console.WriteLine("****************************");
```

The main purpose of the parsing process is to determine if a given input is part of the language syntax or not based on the rules specified on the grammar. When running ANTLR4 plugin with the *-visitor* option we get back a set of output files. Three of them are particularly interesting in this phase. *FOOLBaseVisitor.cs* and *FOOLVisitor.cs* that represent a blueprint of the visitor pattern applied to our grammar rules. The first is an abstract class that define the signature of all virtual methods that then are implemented by us in the second file. This is the point where we insert the logic of how the process has to progress and which controls has to be made.
Although ANTLR44 allow the use of two patterns, *Listener* and *Visitor*, we used the latest because it permit us to control the manner of how the nodes are parsed and also to get information about each node. This information are then used in the next phase for the *Semantic Analysis* and the *Type checking*. The way the *Visitor* works is very simple, it starts from the root node and execute a depth-first search (DFS) on the AST of the chosen rule based on the input and the grammar.
Keep in mind that the visitor produced by ANTLR4 has one method for each rule in the grammar file.
The following listing is the virtual default method produced by ANTRL4 in the *FOOLBaseVisitor.cs* file for the *LetInExp* rule in the grammar.

```
public virtual Result VisitLetInExp([NotNull] FOOLParser.LetInExpContext context)
{ return VisitChildren(context); }
```

Here is the starting rule in the grammar from which ANTRL4 generated the previous method. As you can see is the second option in the starting rule.

```
prog    : exp SEMIC                                        #singleExp
        | let IN (exp SEMIC | stms)                        #letInExp
        | (classdec)+ let? IN (exp SEMIC | stms)           #classExp
        ;
```

Now in our visitor implementation we have to insert the logic of how the visitor will behave in each node it reaches.
The following listing represents the code we implemented for our project for the *LetInExp rule*. As you can see in the code snippet the method get as parameter a *context* object of the type *LetInExpContext* in this case. The logic of how the analysis has to proceed is explicitly express in the code. Inside the method we check different conditions, in this case if we are dealing with an *expression* of a *statement* and base on it we create an instance of *FoolProgramLetInExpressionNode* class with different parameters.

```
        public override IFoolNode VisitLetInExp(LetInExpContext context)
        {
            List<IFoolNode> declarations = new List<IFoolNode>();

            foreach (DecContext dec in context.let().dec())
            {
                declarations.Add(Visit(dec));
            }

            if (context.exp() != null)
            {
                return new FoolProgramLetInExpressionNode(declarations, Visit(
    context.exp()));
            }
            else
            {
                List<IFoolNode> statements = new List<IFoolNode>();
                foreach (StmContext stm in context.stms().stm())
                {
                    statements.Add(Visit(stm));
                }
                return new FoolProgramLetInExpressionNode(declarations, statements)
    ;
            }
        }
```

Keep in mind that all the classes defined inherit by the single interface *IFoolNode* that define three methods as shown in the following snippet.

```
    public interface IFoolNode
    {
        IFoolType TypeCheck();
        List<string> CheckSemantics(FoolEnvironment environment);
        string CodeGeneration();
    }
```

If the parsing finish correctly; no errors found the flow goes throw the *Semantic Analysis* that check the logical meaning of our input code.

5 Semantic Analysis

Semantic Analysis is a fundamental part on the whole code analysis process that try to find that kind of errors that the previous two types of analyser couldn't find, semantic/meaning errors. During this process, the semantic analyser makes use of an important structure called *Symbol Table*. The symbol table in our case is implemented as a list of *dictionaries* in the C# terminology.
More formally. we are going to list the four type of checks that semantic analyser perform:

- check for multiple declarations of the same *name* in a given *scope*. In case of error an exception is raised.
- check for use of undeclared variables. As general rule in Fool a variable has to be declared before being used.
- check for type mismatch. Errors of type: $int x; bool y; x = y$;
- check if the methods are called with the right type and number of parameters.

This checks are done in two sequential steps by traversing the AST. The first step is called *Scope Checking* and the second is called *Type Checking*. We know that the scoping rules are fundamental for the correctness of the code. Fool language uses *static scoping*. The following listing show the entry point in our *Program.cs* file where we perform semantic checking.

```
//SEMANTIC
Console.WriteLine("[<<<<< SEMANTIC ANALYSIS >>>>>]");
FoolVisitor visitor = new FoolVisitor();
IFoolNode ast = visitor.Visit(progContext);
FoolEnvironment environment = new FoolEnvironment();
List<string> error = ast.CheckSemantics(environment);
if (error.Count > 0) throw new FoolSemanticException(error);
Console.WriteLine("\nOK, DONE!");
Console.WriteLine("******************************");
//TYPE CHECKING
Console.WriteLine("[<<<<< TYPE CHECKING >>>>>]");
IFoolType type = ast.TypeCheck(); //type-checking bottom-up
Console.WriteLine("Type checking: " + type.GetFoolType().ToUpper());
Console.WriteLine("\nOK, DONE!");
Console.WriteLine("******************************");
```

As you can see in the code snippet, type checking is done by calling $ast.TypeCheck()$ on the tree. Is important to mention that we have defined 8 different types in the project. All the types derive and implement *IFoolType.cs* interface that defines the signature of 3 methods.

```
    public interface IFoolType
    {
        string GetId();
        bool IsSubType(IFoolType type);
        string GetFoolType();
    }
```

The logic of *GetId()* and *GetFoolType()* is very simple. They just return the type name and the type.
IsSubType() is more elaborate. It takes as input a given type and determines if it is a sub-type of current type.
At the end if there are not errors found we go forward with the *Code Generation*. Keep in mind that at this point the helper structures that we used previously such as Symbol Tables are no needed anymore.

6 Code Generation

The purpose of this step is to generate an assembly-like code that later will be executed by our stack machine called SVM (Simple Virtual Machine). Stack machines are very simple models where the operand are always on the top of the stack.
The grammar for the assembly language, *SVM.g4* is provided by the professor and further enriched by our work adding some more instructions to it, such as *new*, *loadm* etc. Keep in mind that in this

grammar we have removed *actions* and *attributes*. We perform for this grammar all the previous analysis except for the semantic analysis and return the assembly code that has to be executed later on. The code snippet below show the initialization of the *virtualMachine* object. We call on it the *cpu()* method that has the task to execute the assembly code and produce a value as input or an error if any.
As we mention previously the virtual machine used in the project is a stack machine, where the operators are always in the top of the stack and the common *push* and *pop* stack operations are defied. In the case of the objects, this memory arrangement is not enough. We need another data structure called *heap* to deal with object. As in all other OOP languages even in Fool the object are instantiate using *new* keyword. The logic of how heap is implemented is defined in the file *FoolHeap.cs*.
Another important part of the OOP is the inheritance. Fool programming language permits inheritance and method overriding. This feature is implemented using the *dispatch table*. Itself the dispatch table is organized as a dictionary where the Keys are the name of the classes and the Values are a list of dispatch table entries called *FoolDispatchTableEntry*. The following code snippet show the structure of each DispatchTableEntry.

```
public class FoolDispatchTableEntry
{
    private string _methodId;
    private string _methodLabel;

    public FoolDispatchTableEntry(string methodId, string methodLabel)
    {
        _methodId = methodId;
        _methodLabel = methodLabel;
    }
    public string GetMethodId()
    {
        return _methodId;
    }
    public string GetMethodLabel()
    {
        return _methodLabel;
    }
}
```

The two methods that are defined in this class are used to get the the name of the method (GetMethodID()) and to get the label (GetMethodLabel). The labels are used to show the part of the assembly file where the methods code start.

At the end of the executing we will be able to see in the terminal a value or an error.

The resource that i most used to prepare this project is mentioned in the bibliography and is the book "The definitive ANTLR 4 Reference" and the slides of the course.

7 Apendix

The following listing shows the Parser rules used in the Project.

```
/*------------------------------------------------------------
 * PARSER RULES
 *------------------------------------------------------------*/
prog    : exp SEMIC
        | let IN (exp SEMIC | stms)
        | (classdec)+ let? IN (exp SEMIC | stms) ;
let     : LET (dec SEMIC)+ ;
classdec : CLASS ID (EXTENDS ID)?
            (LPAR vardec (COMMA vardec)* RPAR)?
            (CLPAR met* CRPAR)? SEMIC ;
letVar  : LET (varasm)+ ;
vardec   : type ID ;
varasm   : vardec ASM (exp | NULL) ;
fun     : (VOID | type) ID LPAR ( vardec ( COMMA vardec)* )?
            RPAR  CLPAR (letVar IN)? (exp SEMIC | stms) CRPAR ;
dec    : varasm
       | fun ;
met : fun ;
type    : INT
         | BOOL
         | ID ;
exp    : left=term ((PLUS | MINUS) right=exp)? ;
term   : left=factor ((TIMES | DIV) right=term)? ;
factor : left=value ( logicoperator=
                    (EQ|
                    GREATERTHAN|
                    LESSERTHAN|
                    GREATEREQUAL|
                    LESSEREQUAL|AND|OR) right=factor)? ;
value   : (MINUS)? INTEGER
       |  (NOT)? ( TRUE | FALSE )
       |  LPAR exp RPAR
       |  IF cond=exp THEN CLPAR thenBranch=exp
            SEMIC CRPAR (ELSE CLPAR elseBranch=exp SEMIC CRPAR)?
       |  (MINUS| NOT)? ID
       |  functioncall
       |  ID DOT functioncall
       |  NEW ID LPAR (exp (COMMA exp)* )? RPAR ;
functioncall : ID LPAR (exp (COMMA exp)* )? RPAR ;
stm :  ID ASM (NULL | exp)
    | IF LPAR cond=exp RPAR THEN CLPAR thenBranch=stms CRPAR
          (ELSE CLPAR elseBranch=stms CRPAR)? ;
stms :  (stm SEMIC)+ ;
```

The following listing shows the Lexer rules used in the Project.

```
/*------------------------------------------------------
 * LEXER RULES
 *------------------------------------------------------*/
SEMIC   : ';' ;
COLON   : ':' ;
COMMA   : ',' ;
EQ      : '==' ;
ASM     : '=' ;
PLUS    : '+' ;
MINUS   : '-' ;
TIMES   : '*' ;
DIV     : '/' ;
TRUE    : 'true' ;
FALSE   : 'false' ;
LPAR    : '(' ;
RPAR    : ')' ;
CLPAR   : '{' ;
CRPAR   : '}' ;
IF      : 'if' ;
THEN    : 'then' ;
ELSE    : 'else' ;
LET     : 'let' ;
IN      : 'in' ;
VAR     : 'var' ;
FUN     : 'fun' ;
INT     : 'int' ;
BOOL    : 'bool' ;

//ADDED FOR PROJECT.

OR              : '||' ;
AND             : '&&' ;
NOT             : 'not' ;
GREATERTHAN     : '>' ;
LESSERTHAN      : '<' ;
GREATEREQUAL    : '>=' ;
LESSEREQUAL     : '<=' ;


VOID             : 'void';
CLASS            : 'class';
THIS             : 'this';
NEW              : 'new';
DOT              : '.';
EXTENDS          : 'extends';
NULL             : 'null';

//Numbers
fragment DIGIT : '0'..'9';
INTEGER        : DIGIT+;

//IDs
fragment CHAR  : 'a'..'z' |'A'..'Z' ;
ID               : CHAR (CHAR | DIGIT)* ;

//ESCAPED SEQUENCES
WS               : (' '|'\t'|'\n'|'\r')-> skip;
LINECOMENTS     : '//' (~('\n'|'\r'))* -> skip;
```