# The Lucas Compiler


## Group-3

1. JATIN TARACHANDANI
2. PRASHANTH SRIRAM S
3. S GOUTHAM SAI
4. SHANTANU PANDEY
5. ARAVINDA KUMAR REDDY T
6. ANIRUDH SRINIVASAN
7. T SRIVATSAN
8. JULAKUNTLA MADHURI

# Team Members and Roles

1. JATIN TARACHANDANI - Project Manager
2. PRASHANTH SRIRAM S - System Integrator
3. S GOUTHAM SAI - Language Guru
4. SHANTANU PANDEY - System Architect
5. ARAVINDA KUMAR REDDY T - System Architect
6. ANIRUDH SRINIVASAN - Language Guru
7. T SRIVATSAN - System Tester
8. JULAKUNTLA MADHURI - System Tester

# GOAL OR KEY FEATURES

- Goal is to construct compiler to support Calculus related stuff.

- The following are the USPs of our Language:
  - ➢ Supports Calculus computations
  - ➢ Supports Multi-returning functions
  - ➢ Support arbitrary-precision integers

# ANTLR V4

- ANTLR - ANother Tool for Language Recognition.
- Antlr can be used for both lexing and parsing.
- We need to provide a target language in which the Parser should be generated.
  [For example: C++, C#, Java, Go, Swift, PHP]
- We have used Antlr to generate the Lexer/Parser in Java.

# ANTLR

- ANTLR uses LL(k) parsing to analyse the grammar.
- Parsers, lexers, and tree-parsers are accepted grammar specifications.
- Commands: (‘LucasGrammar.g4’ be the grammar file)
  - antlr4 LucasGrammar.g4
  - javac LucasGrammar*.java
  - grun LucasGrammar compilationUnit test.txt -gui
  - (Or) grun LucasGrammar compilationUnit test.txt -tree

# Initial Plan and Current status

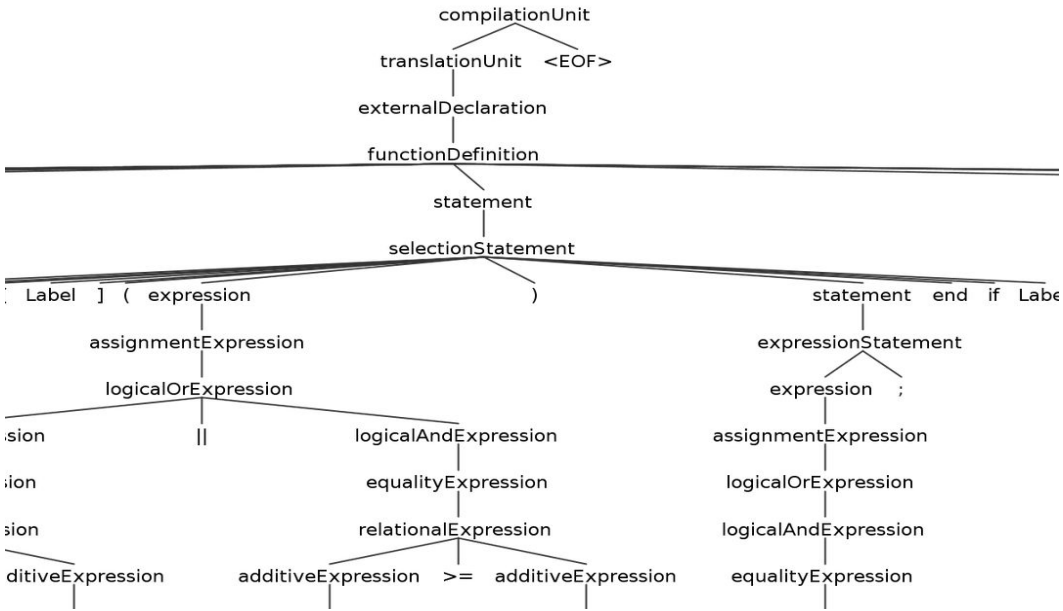
- Our initial plan was to generate lexer, parse using ANTLR. Then write the AST and then generate the LLVM IR taking the help of api's if needed.
- But, due to lack of proper resources and time,  we ended up doing some part of semantic analysis.

# Grammar File(Lexer and Parser)

```
grammar LucasGrammar;


primaryExpression
    :   Identifier
    |   Literal
    |   StringLiteral+
    |   '(' expression ')'
    ;

postfixExpression
    :
    (   primaryExpression
    |   '__extension__'? '(' typeName ')' '{' initializerList ','? '}'
    )
    ('[' expression ']'
    | '(' argumentExpressionList? ')'
    | ('.' | '->') Identifier
    | ('++' | '--')
    )*
    ;

argumentExpressionList
    :   assignmentExpression (',' assignmentExpression)*
    ;

unaryExpression
    :
    ('++' |  '--' |  'sizeof')*
    (postfixExpression
    |   unaryOperator castExpression
    |   'sizeof' '(' typeName ')'
    |   '&&' Identifier // GCC extension address of label
    )
    ;
```

A small snippet of our Grammar

# Sample Parse Tree




This is how our output parse tree looks like

# Semantic Analyser

- We have hand-written java classes following the rule-‘one class for every non-terminal and a subclass for each production rule’ .
- We planned to use visitor pattern to access the classes.
- Consider the grammar rule of declarationSpecifiers:

```
-declarationSpecifiers
                    :   declarationSpecifier+
                    ;
```

- The following two slides will explain how we wrote class for the above rule.

# Class code of declarationSpecifiersNode

```
1  package classdefs;
2
3  import java.util.*;
4
5  public class declarationSpecifiersNode extends ASTNode{
6      List<declarationSpecifierNode> declarationSpecifierNodes;
7
8      public declarationSpecifiersNode(int line_no, List<declarationSpecifierNode> declspecnodes){
9          lineno = line_no;
10         this.declarationSpecifierNodes = declspecnodes;
11     }
12
13     String getNode(){
14         return " declartaionSpecifierNode class defined at line mumber " + lineno;
15     }
16
17     String visit()
18     {
19         return ""; // we will change this later in order to have a visitor of some kind return through the AST
20     }
21 }
```

This is class code of the

“DeclarationSpecifiers”

Node

# Explanation of above class code

- We created a class declarationSpecifiers that extends ASTNode.
- As per the production in the grammar rule we need one or more declaration specifier nodes. So we used list to take care of that.
- The class also consists of  a constructor and getNode(), visit() methods.

# CHALLENGES FACED

- We got an issue due to an indirect left recursion.
- This was caused because we used begin(instead of ‘{‘) and end(instead of ‘}’).
- Fixed this by removing compound statement, and replacing it with (statement|declaration)*.