EAST WEST UNIVERSITY

Project Report: Parse Tree Generation Using Antlr

ABSTRACT

Our project is about creating parse tree and separating tokens using ANTLR tool. We have mainly focused on how the tool ANTLR is working and by using it creating a parse tree from the written grammar. For using this tool, we also need IDEA . In our project, we have tried to generate a parse tree based on function, if and loop of a new language.

INTRODUCTION:

A parse tree is an entity which represents the structure of the derivation of a compiler. The first step of a compiler is to create a parse tree of the program tree. Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end. ANTLR is a parser generator, a tool that helps you to create parsers. A parser takes a piece of text and transforms it in an organized structure, such as an Abstract Syntax Tree (AST). AST is a tree representation of the abstract syntactic structure of source code written in a language.

CONTEXT-FREE GRAMMAR (CFG):

A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple **(N, T, P, S)** where

- **N** is a set of non-terminal symbols.

- **T** is a set of terminals where **N ∩ T = NULL.**

- **P** is a set of rules, **P: N → (N ∪ T)***, i.e., the left-hand side of the production rule **P** does have any right context or left context.

- **S** is the start symbol.

Context **F**ree **G**rammars (CFG) can be classified on the basis of following two properties:
1. Based on number of strings it generates.

2. Based on number of derivation trees.

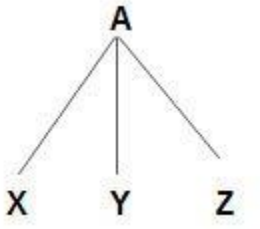
ANTLR TOOL:

ANTLR (Another Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees. A parser takes a piece of text and transforms it in an organized structure, such as an Abstract Syntax Tree (AST). We can think of the AST as a story describing the content of the code, or also as its logical representation, created by putting together the various pieces. From a grammar, ANTLR generates a parser that can build parse trees and also generates a listener interface (or visitor) that makes it easy to respond to the recognition of phrases of interest. Other than lexers and parsers, ANTLR can be used to generate tree parsers.

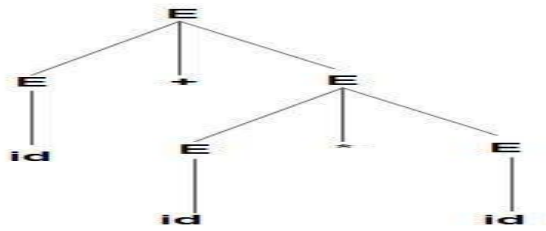
PARSE TREE:

• Parse tree is a hierarchical structure which represents the derivation of the grammar to yield input strings.

• Root node of parse tree has the start symbol of the given grammar from where the derivation proceeds.
• Leaves of parse tree represent terminals.
• Each interior node represents productions of grammar.
• If *A* -> *xyz* is a production, then the parse tree will have *A* as interior node whose children are *x*, *y* and *z* from its left to right



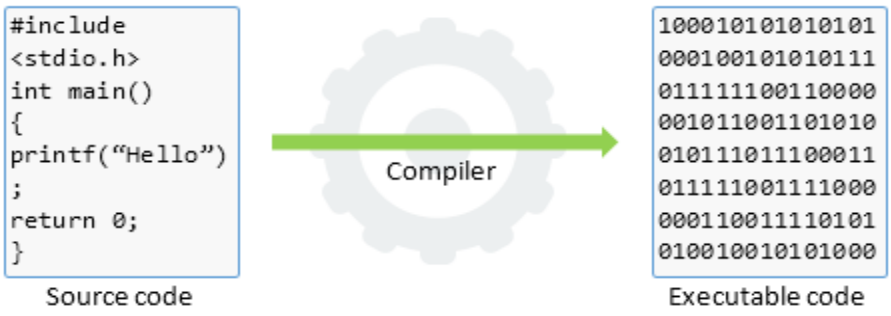
Construct parse tree for E --> E + E I E * E I **id**



COMPILER:

A compiler is a software program that transforms high-level source code that is written by a developer in a high-level programming language into a low level object code (binary code) in machine language, which can be understood by the processor. The process of converting high-level programming into machine language is known as compilation.

The processor executes object code, which indicates when binary high and low signals are required in the arithmetic logic unit of the processor



Correct Input:

```
* include < ters >
int main {}
[
int a = 10;
int b = 5;
int i = 15;
int j = 0;
int k = 25;
double c = 21;
double d = 22;
if(c greater than d)
[
print c;
]
else
[
print d;
]
]
```

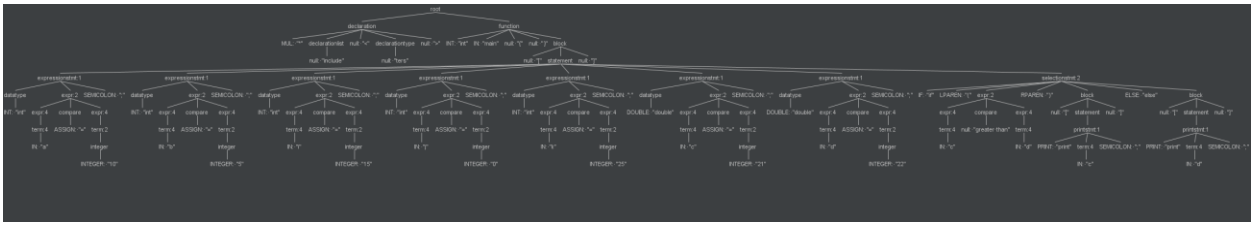
loop(j -> b , 1)

[

print "Hello World";

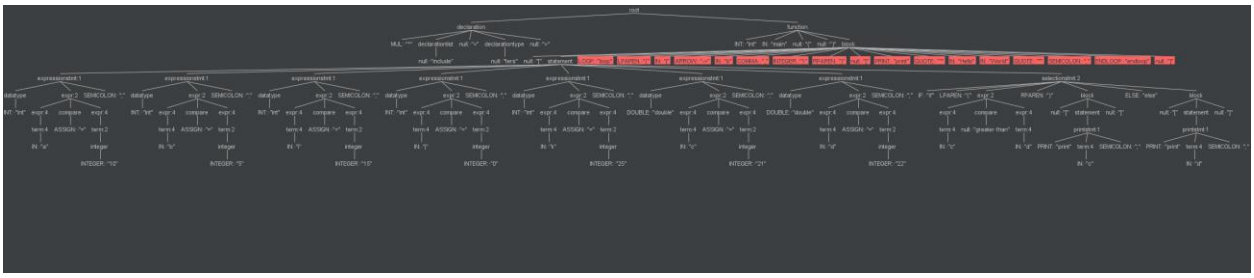
endloop

]



Incorrect input:

include < ters >
int main {}
[
int a = 10;
int b = 5;
int i = 15;
int j = 0;
int k = 25;
double c = 21;
double d = 22;
if(c greater than d)
[
print c;
]
else
[
print d;
]
loop(j -> b , 1)
[
print "Hello World";
endloop
]



OUR GRAMMAR:

grammar a;
root : declaration function;
declaration : '*' declarationlist '<' declarationtype '>' ;
declarationlist : 'include' | 'define' ;
declarationtype : 'ters' ;
function : 'int' IN '{' '}'block;
block : '[' statement ']' ;
statement :(
expressionstmt
|selectionstmt
| printstmt
| assignstmt
|statement_return
| loopstmt)+
;
expressionstmt : datatype expr ';' | ';' ;
statement_return : 'return' expr ';' | 'return' term ';' ;
expr : expr expression expr | expr compare expr | '(' expr ')' | term ;
expression : 'plus' | 'minus' | 'into' | 'by' ;
compare : 'equal to' | 'not equal' | 'greater than equal' | 'less than' | 'greater than' | 'less than equal' | '=' ;
selectionstmt : 'if' '(' expr ')' block | 'if' '(' expr ')' block 'else' block ;
printstmt :
PRINT term SEMICOLON

```
        |
        PRINT QUOTE STRING+ QUOTE SEMICOLON
        ;
        assignstmt :
    NAME ASSIGN expression SEMICOLON
    ;
    loopstmt :
    LOOP LPAREN identifier ARROW integer COMMA integer RPAREN
    (statement | loopstmt)+
    ENDLOOP
    |
    LOOP LPAREN identifier ARROW identifier COMMA integer RPAREN
    (statement | loopstmt)+
    ENDLOOP
    |
    LOOP LPAREN identifier ARROW integer COMMA identifier RPAREN
    (statement | loopstmt)+
    ENDLOOP
    |
     LOOP LPAREN identifier ARROW identifier COMMA identifier RPAREN
     (statement | loopstmt)+
     ENDLOOP
     ;
     term : identifier
     | integer
     | dbl
     |IN
     ;
     identifier : NAME ;
     datatype : INT
     |
     DOUBLE
     |
     FLOAT
;
dbl : DBL
;
integer : INTEGER;
IF: 'if';
ELSE: 'else';
ENDIF: 'endif';
PRINT: 'print';
INT: 'int';
DOUBLE: 'double';
FLOAT: 'float';
LOOP: 'loop';
ARROW: '->';
COMMA: ',';
ENDLOOP: 'endloop';
```

```
PLUS: '+';
MINUS: '-';
DIV: '/';
MUL: '*';
ASSIGN: '=';
POINT: '.';
SEMICOLON: ';';
LPAREN: '(';
RPAREN: ')';
QUOTE: '"';
IN: [a-zA-Z]+ ;
INTEGER: [0-9][0-9]*;
DBL: INTEGER POINT INTEGER
| INTEGER
;
SYMBOL: (','|'.' | '/' | '*' | '+' | '-' | '<' | '>' | '!' | '@' | '#' | '$' | '%' | '&' | '=' | ':');
STRING: ([a-z] | [A-Z] | [0-9] | SYMBOL)+;
WS: [ \t\r\n]+ -> skip ;
```

CONCLUSION:

The goal of our project is to learn about the parser generator, how a parse tree is generated and how the grammar creates a parse tree. We have tried to implement some basic features of C language to see the working procedure of the parser generator.

SOURCES:

- https://en.wikipedia.org/wiki/ANTLR
- https://www.antlr.org/
- https://www.tutorialspoint.com/automata_theory/context_free_grammar_introduction.htm
- https://www.geeksforgeeks.org/classification-of-context-free-grammars/
- http://ecomputernotes.com/compiler-design/parse-tree