Abstract Syntax Tree
Project Report

Programming Language CS-306

Ayush Sawlani
International Institute of Information Technology, Bangalore
IMT2018014
Ayush.Sawlani@iiitb.org

Manan Bansal
International Institute of Information Technology, Bangalore
IMT2018039
Manan.Bansal@iiitb.org

Soham Kolhe
International Institute of Information Technology, Bangalore
IMT2018073
Soham.kolhe@iiitb.org

I. Introduction

A. Abstract Syntax Tree

An abstract syntax tree (AST) is a tree that is used to represent the abstract syntactic structure of the source code written in a programming language which in this case is C. The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. The following are the advantages of an Abstract Syntax Tree:

- Easy to visualise the flow of the source code.
- An AST usually contains extra information about the program, due to the consecutive stages of analysis by the compiler.
- An AST can be edited and enhanced with information such as properties and annotations for every element it contains.
- Compared to the source code, an AST does not include inessential punctuation and delimiters (braces, semi-colons, parentheses, etc.).

II. Construction Modules

A. Lexer

Lexer is a software that performs lexical analysis, which is the process of converting a sequence of characters into a sequence of tokens. Tokens are strings with an assigned value and are meaningful. The tokens are then passed on to the parser as inputs.

We used the lexing tool flex to generate lexing C code. We use the command "sudo apt install flex" in the terminal to download the tool. Flex takes a lex file as an input. The lex file contains the rules for matching strings with their tokens. The tokens are defined in the parsing file which is discussed later. The output of running flex on a lex file as C file which has yylex and other related functions defined.

1) Parser: The job of a parser is to convert the stream of tokens produced by the lexer into a parse tree representing the structure of the parsed language.

We used bison as a tool to generate parsing C code. We use the command "sudo apt install bison" in the terminal to download the tool. bison takes a yacc file as an input. The yacc file contains the context free grammar of the language for which we are building our parse tree. It calls the function yylex() from the lexing code repeatedly to get tokens and uses LALR(1) algorithm to build the parse tree

Figure 1: Example for Parsing

III. Implementation Overview

- Before creating the AST for the given string of code, we first check whether the initial parsing is correct or not. (control.c)
- So for the above step, we have created 2 files, check_lexer.l and check_parser.l. These files are executed first using lex and yacc respectively.
- The check_lexer.l file creates the file lex.yy.c which contains the function yylex().
- The check_parser.l file creates the files y.tab.h and y.tab.c which contain the function yyparse(). This function returns 0 if parsing is successful else returns 1 if parsing is unsuccessful.
- The check_lex.l file contains the tokens associated with the keywords of the language.

- The check_parser.l file contains the Grammar for C language.
- The y.tab.c file contains the numerical values associated with each token present in check_lexer.l
- If the parsing is successful, the main function then calls the graph_lexer.l and graph_parser.l files which are created by taking the check files as the basis.
- graph_lexer.l and graph_parser.l files treat the tokens as a struct representing the node of the tree. For each token and expression, we are calling the create_node function.
- Generate_graph.c after running yyparse() gets the root node of the tree in mtree. After that we run dfs to generate a file which will represent the tree in the form of adjacency lists. This file will be read by our display program to generate the tree.

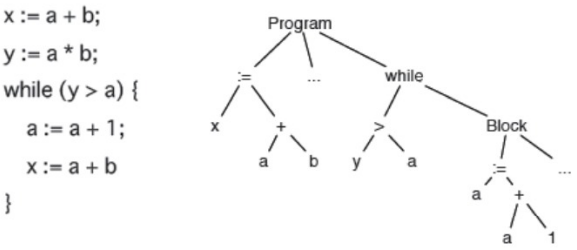

Figure 2: Example for an AST

IV. Running Our Program

Follow these steps to run our program:

- Download the zip file and decompress it.
- Open terminal and navigate to the project directory.
- make sure you have flex and bison installed in your pc. If not, use ”sudo apt install flex” and then ”sudo apt install bison”.
- Open the ”hello.txt” file and copy your input program into it. (We are sorry for not making this interactive)
- enter ”make clean” command.
- enter ”make all” command.
- enter ”./Generate” command.
- Copy numerical values from output.txt to this.tree variable in tree.js.
- Copy strings values from output.txt to this.value variable in tree.js file.
- This is hard coded as JavaScript does not have access to local files.
- run index.html.

Our program does not parse every C code as of yet. Only the the input programs with the following properties are parsed as of now:

- It must not contain define or include and other directives. The gcc compiler in principle expands these directives and then does the parsing.
- It must not have complex data types. This functionality will be implemented later in our project. During our demonstration too we failed to run a C program which had structs. This functionality will be implemented later.

V. Conclusion

Overall, we think we did a good job to generate the Abstract Syntax Tree for C programming language. We would like to thank Prof. Sujit and the TAs to teach us the basic concepts of parsing and also help us out in our project when we were stuck. We would also like to thank our batchmates for having fruitful discussions with us on and off the class timings. Lastly, We would like to thank people on stackoverflow, YouTube and article writers on GeeksForGeeks to provide us solutions to our other problems that we encountered while writing code.

VI. References

- Lex and Yacc tutorial click here
- Prof Sujit’s Github programs click here
- Basic GeeksForGeeks yacc program click here
- Lex grammar for C click here
- Yacc grammar for C click here