

SPL-1 Project Report, [2022]

Abstract Syntax Tree

SE 305: Software Project Lab 1

Submitted by

Mosamma Sultana Trina

BSSE Roll No. : 1313

BSSE Session: 2020-21

Supervised by

Dr. Sumon Ahmed

Designation: Associate Professor

Institute of Information Technology



Institute of Information Technology

University of Dhaka

[21-05-2023]

Table of Contents

1. Introduction..... 3

 1.1. Background Study 3

 1.2. Challenges.....6

2. Project Overview.....6

3. User Manual.....12

4. Conclusion.....15

References.....16

1. Introduction

My project is to build Abstract Syntax Tree for C source code. An abstract syntax tree (AST) is a data structure commonly used in computer science and programming language theory to represent the structure of a program. It is a hierarchical tree-like representation that captures the syntax and organization of the program's code, but abstracts away details that are not relevant to its structure, such as semicolon or specific formatting. The AST represents the syntactic structure of the program by breaking it down into individual nodes, where each node corresponds to a specific construct in the programming language. For example, in a language like C, an AST node might represent a variable declaration, assignment, or a loop construct. ASTs are useful for several reasons. Such as Compilation and Interpretation, Static Analysis, Code Transformation and Refactoring.

1.1. Background Study

To lay a solid foundation for the project, an extensive background study was conducted to understand the concepts and techniques related to lexical analysis, parsing, and abstract syntax trees.

Lexical Analysis:

The first phase of building the abstract syntax tree is the lexical analysis. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. A token is an object describing a lexeme. Along with the value of the lexeme (the actual string of characters of the lexeme), it contains information such as token ID, its type (is it a keyword? an identifier? an operator? ...)

```
int main()
{
    int x = 0, y = 5;
    printf("First number is %d and the second number %d", x, y);
    return 0;
}
```

In the above example of lexical analysis, we can easily recognize that there are 27 tokens in the above code. Tokens in the code are 'int' 'main' '(' ')' '{' 'int' 'x' '=' '0' ',' 'y' '=' '5' ';' 'printf' '(' '"First number is %d and the second number %d"' ',' 'x' ',' 'y' ')' ';' 'return' '0' ';' '}'.

Parsing or Syntax Analysis:

In syntax analysis, the compiler checks the syntactic structure of the input string, whether the given string follows the grammar or not. It uses a data structure called a Parse Tree or Syntax Tree to make comparisons. The parse tree is formed by matching the input string with the pre-

defined grammar. If the parsing is successful, the given string can be formed by the grammar, else an error is reported.

The parsing techniques can be divided into two types:

Top-down parsing: The parse tree is constructed from the root to the leaves in top-down parsing. Some most common top-down parsers are Recursive Descent Parser and LL parser.

Bottom-up parsing: The parse tree is constructed from the leaves to the tree's root in bottom-up parsing. Some examples of bottom-up parsers are the LR parser, SLR parser, CLR parser, etc.

Parse Tree:

Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal. In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol. It is the graphical representation of symbol that can be terminals or non-terminals. It is a hierarchical structure which represents the derivation of the grammar to yield input strings.

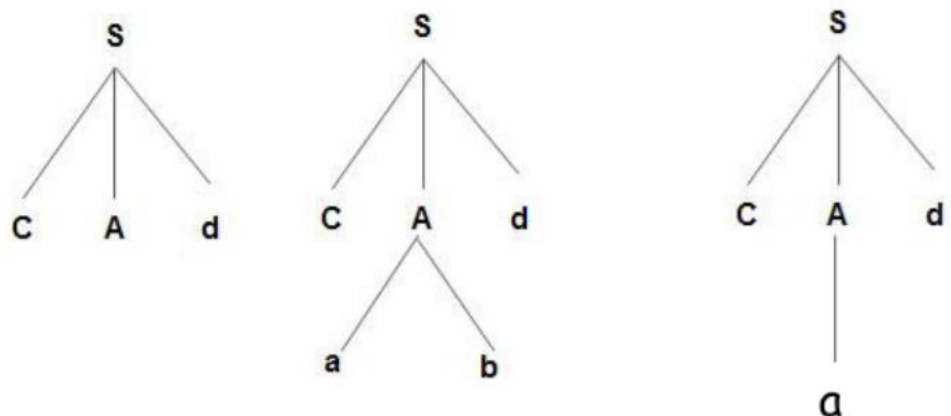
- Root node of parse tree has the start symbol of the given grammar from where the derivation proceeds.
- Leaves of parse tree represent terminals.
- Each interior node represents productions of grammar.

Consider the grammar:

$S \rightarrow cAd$

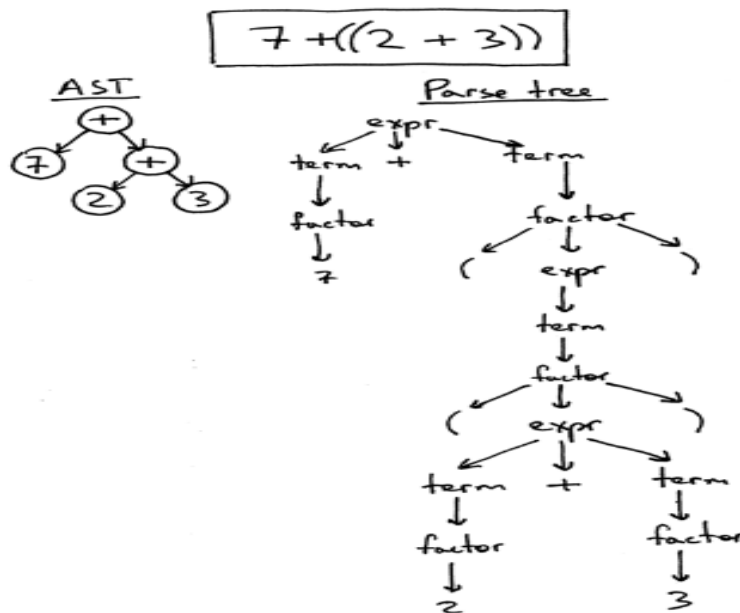
$A \rightarrow ab \mid a$

To derive the string cad,



Abstract Syntax Tree:

Some compilers use an abstract syntax tree (AST) to represent the program being compiled. The AST has the essential structure of the parse tree but eliminates some of the unnecessary nodes. Some parse tree nodes may not be required in the AST representation, such as punctuation or parentheses. Exclude these nodes from the AST during the construction process.



Print Tree:

To print the parse tree and abstract syntax tree, I use the Depth-First Search (DFS) algorithm.

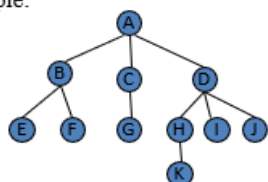
Additionally, I represent the tree using the general tree representation, known as the left child right sibling representation.

Memory Representation of General Tree

- Suppose T is a general tree. T is maintained in memory by means of a linked representation that uses following three parallel arrays:

- INFO[K] = Information at node N
- CHILD[K] = location of the first child of N.
- SIBL[K] = location of next sibling of N

- Here K is the location of node N of T.
- Here ROOT is used as the root of T.
- Example:



ROOT [6]

	INFO	CHILD	SIBL
1	C	3	13
2	B	5	1
3	G	0	0
4	K	0	0
5	E	0	9
6	A	2	0
7	I	0	12
8			
9	F	0	0
10			
11	H	4	7
12	J	0	0
13	D	11	0

Figure: General Tree and Its Memory Representation

1.3 Challenges:

New Challenges are always faced while implementing a software solution. While implementing the Abstract Syntax Tree, a lot of challenges were faced namely:

- **Parsing Complexity:** Understanding the concepts of parsing and dealing with source code proved to be quite challenging, especially for someone new to this area. Parsing code character by character was a time-consuming task that required careful handling of various cases and situations.
- **Learning Context-Free Grammar:** Mastering the intricacies of Context-Free Grammar (CFG), which defines the syntax rules of the programming language, required a significant time investment. It involved understanding components like terminals, non-terminals, production rules, and their relationships.
- **Implementing Production Rules:** Implementing the detailed production rules was the most difficult aspect of building the Parse Tree. It demanded a deep understanding of the code's structure and semantics to construct Parse Tree and AST.
- **Handling Nested "if" Statements:** Dealing with nested "if" statements adds complexity to the Parse Tree construction process. Managing the indentation levels, tracking the conditionals, and accurately representing the hierarchical structure of nested "if" statements requires careful consideration and implementation. Ensuring the proper nesting and capturing the correct semantics of the code within the parse tree was challenging, especially when there are multiple levels of nested conditionals.
- **Limited Resources:** Finding comprehensive and reliable resources online was a challenge. This scarcity hindered progress and required resourcefulness to explore alternative avenues for acquiring the necessary knowledge and guidance.
- **Parse Tree to Abstract Syntax Tree:** Converting parsed code into a suitable AST representation was a confusing task. Designing a logical structure that accurately captured the code's semantics and relationships was a little bit challenging.
-

2. Project Overview

The project has mainly Three Steps:

1. Lexical Analysis
2. Construct Parse Tree
3. Build Abstract Syntax Tree

Lexical Analysis:

Lexing.cpp file uses the C source code and breaks it into tokens by removing the comments and extra white spaces used in the code. It also checks the invalid tokens after creating them. If there are any invalid tokens, the lexical analyzer stops analyzing the code and gives an error. Mainly, the Lexical analyzer checks the legal tokens and then forwards the code to the other phases. Some of the steps of tokenization is given below:

```
ere X Lexing1.cpp X
4 struct token{
5
6     string name;
7     string type;
8     int id;
9
10 };
11
12 token tok [10000];
13
14 void removeAllComments(FILE *input_file, FILE *output_file) {
15
16     char c, next_c;
17
18     while ((c = fgetc(input_file)) != EOF) {
19         if (c == '/') {
20             next_c = fgetc(input_file);
21             if (next_c == '/') {
22                 while ((c = fgetc(input_file)) != EOF && c != '\n') {}
23                 if (c == '\n') {
24                     fputc(c, output_file);
25                 }
26             } else if (next_c == '*') {
27                 while ((c = fgetc(input_file)) != EOF) {
28                     if (c == '*') {
29                         next_c = fgetc(input_file);
30                         if (next_c == '/') {
31                             break;
32                         }
33                     }
34                 }
35             }
36             fputc(c, output_file);
37         }
38     }
39 }
```

Each token contains its unique token ID.

```

input_sc.get(ch);

while(true){
    if(ch=='\n' || ch==' ' || ch=='\t'){

        if(type_specifier(str)==true){

            tok[token].name=str;
            tok[token].type="type_spec";

            if(str=="int"){
                tok[token].id=81;
            }
            else if(str=="float"){
                tok[token].id=82;
            }
            else if(str=="bool"){
                tok[token].id=83;
            }
            str="";
            token++;
            input_sc.get(ch);
            break;
        }

        if(str[0]=='#' && str[str.size()-1]=='>'){

            tok[token].name=str;
            tok[token].type="Header";
            tok[token].id=1;
            token++;
        }
    }
}

```

Here, we can see type specifiers and header files are being lexed from the source code.


```
rt here X Lexing1.cpp X
277
278     tok[token].name=str;
279     tok[token].type="while_stmt";
280     tok[token].id=10;
281     str="";
282     token++;
283 }
284 else if(str=="for"){
285
286     tok[token].name=str;
287     tok[token].type="for_stmt";
288     tok[token].id=15;
289     str="";
290     token++;
291 }
292 else if(str=="void"){
300 else if(break_statement(str)==true){
301
302     tok[token].name=str;
303     tok[token].type="break_stmt";
304     tok[token].id=20;
305
306     str="";
307     token++;
308 }
309 else if(ifElse_stmt(str)==13){
310
311     tok[token].name=str;
312     tok[token].type="if_stmt";
313     tok[token].id=22;
314
```

Here, we have detected break, if, while, for and converted them into tokens for further use.

Construct Parse Tree:

A parse tree will be formed with the tokens. Any irregularities with the Context Free Grammar will be determined from here. The prototypes of functions used in the Analyzer will be shown below whose activities can be guessed from their name:

```

art here X ParseTree1.cpp X
22
23     Tree tree [10000];
24
25     int indx=0;
26     int level=0;
27     int open=0;
28
29     void printParseTree(node newnode, int depth);
30     void printTreeDetail();
31     node parse (string str);
32     node program (string str);
33     node decl_list(string str);
34     node decl (string str);
35     node var_decl(string str);
36     node type_spec(string str);
37     node main_func(string str);
38     node stmt_list(string str);
39     node stmt (string str);
40     node st_list (string str);
41     node st (string str);
42     node compound_stmt(string str);
43     node local_decls(string str);
44     node local_decl(string str);
45     node if_stmt (string str);
46     node for_stmt (string str);
47     node for_expr(string str);
48     node while_stmt (string str);
49     node print (string str);
50     node expr_stmt(string str);
51     node expr (string str);
52     node break_stmt (string str);
53     node return_stmt (string str);
54
55     node parse (string str){
56

```

These are the functions used for constructing Parse Tree. Now the Context Free Grammar whose help has been taken in building Parse Tree:

```

Program      → decl_list main_func
decl_list    → decl_list decl | E
decl         → var_decl
var_decl     → type-spec IDENT ; | type-spec IDENT[ ] ;
type_spec    → VOID | BOOL | INT | FLOAT
main_func    → int main ( ) { stmt_list } | int main ( ) compound_stmt
stmt_list    → stmt_list stmt | E
stmt         → if_stmt | while_stmt | return_stmt | expr_stmt |
              for_stmt | break_stmt | print_stmt | var_decl
expr_stmt    → expr ; | ;
while_stmt   → WHILE ( expr ) { st_list }
st_list      → st_list st | E
st           → if_stmt | break_stmt | expr_stmt | print_stmt | var_decl
for_stmt     → FOR ( for_expr ; for_expr ; for_expr ) { st_list }
for_expr     → expr | E
compound_stmt → { local_decls stmt_list }
local_decls  → local_decls local_decl | E
local_decl   → type-spec IDENT ; | type-spec IDENT[ ] ;
print_stmt   → printf (STRING_LITERAL);

```

if_stmt → IF (expr) { st_list }
 break_stmt → BREAK ;
 return_stmt → RETURN ; | RETURN expr ;

The following expressions are listed in order of increasing precedence:

expr → IDENT = expr
 → expr EQ expr | expr NE expr
 → expr LE expr | expr < expr | expr GE expr | expr > expr
 → expr + expr | expr - expr
 → (expr)
 → IDENT
 → BOOL_LIT | INT_LIT | FLOAT_LIT | STRING_LIT

Build Abstract Syntax Tree:

To build an abstract syntax tree from a parse tree, we need to remove unnecessary nodes such as brackets, semicolons, and any node whose child is "E" (epsilon).

```

void deleteNode(node& n)
{
    for (auto it = n.children.begin(); it != n.children.end(); )
    {
        if (it->label == "E" || it->label == "(" || it->label == ")" || it->label == "{" || it->label == "}" || it->label == ";" || it->label == ",")
        {
            it = n.children.erase(it);
        }
        else
        {
            deleteNode(*it);
            ++it;
        }
    }
}

```

Traversing and printing a tree by calling a recursive function, it is commonly referred to as performing a depth-first search (DFS) traversal. DFS starts at the root of the tree and explores as far as possible along each branch before backtracking.

```

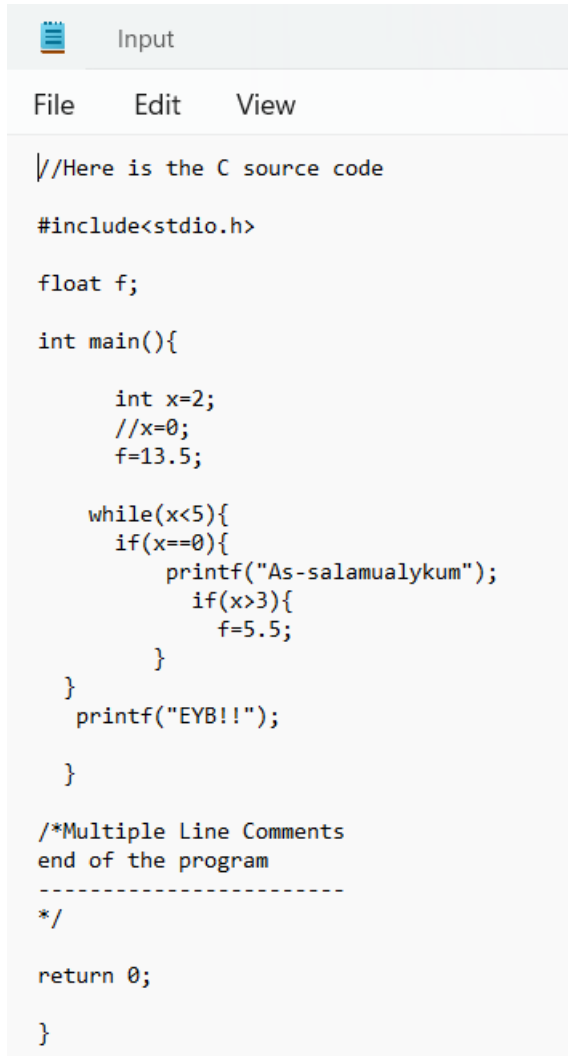
4
5 void printAbstractSyntaxTree(node n, int depth)
6 {
7     output3 << string(depth, ' ') << n.label << endl;
8
9     for (node child : n.children)
0     {
1         printAbstractSyntaxTree(child, depth + 4);
2     }
3
4 }
5

```

printAbstractSyntaxTree function takes two parameters: n represents the current node being visited, and depth represents the current depth level in the tree. The function prints the label of the current node with an indentation based on the depth level.

3. User Manual

Before running the project, you need to create an input file that contains the C source code.



```
//Here is the C source code

#include<stdio.h>

float f;

int main(){

    int x=2;
    //x=0;
    f=13.5;

    while(x<5){
        if(x==0){
            printf("As-salamualykum");
            if(x>3){
                f=5.5;
            }
        }
        printf("EYB!!");

    }

    /*Multiple Line Comments
end of the program
-----
*/

    return 0;

}
```

The project consists of four C files: Lexing1.cpp, ParseTree1.cpp, AST1.cpp, and drive.cpp. To run the project, start by executing the drive.cpp file. This will trigger the execution of Lexing1.cpp, which will generate the following outputs:

Token			
File Edit View			
NO:	Token	Token_Type	Token_ID
1.	#include<stdio.h>	Header	1
2.	float	type_spec	82
3.	f	IDENT	28
4.	;	SEMICOLON	36
5.	int main	main_func	9
6.	(F_BRAC_O	37
7.)	F_BRAC_C	38
8.	{	S_BRAC_O	39
9.	int	type_spec	81
10.	x	IDENT	28
11.	=	ASSIGNMENT	50
12.	2	INT_LIT	3
13.	;	SEMICOLON	36
14.	f	IDENT	28
15.	=	ASSIGNMENT	50
16.	13.5	FLOAT_LIT	2
17.	;	SEMICOLON	36
18.	while	while_stmt	10
19.	(F_BRAC_O	37
20.	x	IDENT	28
21.	<	LESSER	49
22.	5	INT_LIT	3
23.)	F_BRAC_C	38
24.	{	S_BRAC_O	39
25.	if	if_stmt	22
26.	(F_BRAC_O	37
27.	x	IDENT	28
28.	==	EQUAL	51
29.	0	INT_LIT	3
30.	}	F_BRAC_C	38

Afterward, execute ParseTree1.cpp, which will produce a parse tree containing detailed information about the source code.

Brackets are balanced!

Parse Tree:

No:	Parent	Number of Childs	Is LEAF	Childs
1:	NULL	1	NO	program
2:	program	2	NO	decl_list main_func
3:	decl_list	2	NO	decl_list decl
4:	decl	1	NO	var_decl
5:	var_decl	2	NO	type_spec IDENT ;
6:	type_spec	1	NO	FLOAT
7:	FLOAT	X	YES	NO CHILD
8:	IDENT	X	YES	NO CHILD

Finally, run AST1.cpp to construct an abstract syntax tree (AST) from the parse tree, eliminating unnecessary nodes. The program will then print the AST using depth-first traversal.

No:	Parent	Number of Childs	Is LEAF	Childs
1:	NULL	1	NO	program
2:	program	2	NO	decl_list main_func
3:	decl_list	2	NO	decl_list decl
4:	decl	1	NO	var_decl
5:	var_decl	1	NO	type_spec IDENT
6:	type_spec	1	NO	FLOAT
7:	FLOAT	X	YES	NO CHILD
8:	IDENT	X	YES	NO CHILD
9:	main_func	2	NO	int main () compound_stmt
10:	int main ()	X	YES	NO CHILD



Therefore, the abstract syntax tree has been constructed in the above stated manner.

4. Conclusion

In conclusion, this project has been instrumental in enhancing my understanding of compiler operations and has provided me with valuable insights into various aspects of code analysis and parsing. Through this project, I have gained a deeper understanding of how compilers work and the importance of managing code effectively. Handling large lines of code has become more manageable. I have built the Abstract Syntax Tree (AST) for a specific domain (if statements, nested if statements, while loops, and for loops). The AST accurately represents the syntactic structure of the code within this domain. However, I have plans to extend the functionality of the AST in the future so that it can handle and build ASTs from any valid C source code, regardless of

the specific domain or constructs used. This expansion will involve enhancing the parser and AST construction algorithms to handle a broader range of C language features and syntax

References

- [1] <https://ruslanspivak.com/lrbasi-part7/>, ruslanspivak.com, last accessed on 17.05.2023
- [2] Research paper on Compiler Design, Jatin Chhabra, Hiteshi Chopra, Abhimanyu Vats, INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH IN TECHNOLOGY, 2014, 151
- [3] <https://www.twilio.com/blog/abstract-syntax-trees>, twilio.com, last accessed on 22.04.2023
- [4] <https://www.cs.wcupa.edu/rkline/fcs/parse-trees.html>, .cs.wcupa.edu last accessed on 11.04.2023