Report on C parser

Group No.: 29

Name	Roll Number
Aashish Raj	19MT10001
Tanmay Vijay Vaghale	19CH30034
Gaurav Jha	19CH10074

Table of Contents

Introduction **2**

Grammar **3**

Understanding the Grammar **10**
- Syntax of Regular Expression 10
- Terminals 10
- Non-Terminals 11
- Production Rules 11
- Miscellaneous 11

Thought Process **12**

How to use **12**

Results **13**

Discrepancies **21**

Observations **23**

Contributions **23**

Bibliography **23**

Introduction:

Context-free grammar (also called phrase structure grammar) is a set of rules or productions, each expressing how symbols of the language can be grouped and ordered together. It is equivalent to Backus-Naur Form, or BNF. Context-free rules can be hierarchically embedded so that we can combine the previous rules with others. There are two types of symbols in any context-free grammar of any language. These are terminals and non-terminals. The symbols correspond to words in the language. Non-terminals express abstraction over these terminals.

In this project, we developed concise context-free grammar for the C language. Given the C program, the parser will try to check if the program follows the rules or not. In other words, it checks for the syntactic correctness of the given C program. We used **Lark** parser in Python to tokenise and parse the given code. The lark parser reads the grammar in **EBNF** (Extended Backus-Naur Form) and provides us with the freedom to choose from a varied range of parsing algorithms like **CYK, Earley, LALR** and so on to parse the code and generate its respective parse-tree / **Abstract Syntax Tree** (AST).
We chose the Lark Parser because of the various features it provides, namely :

- Beginner-friendly
- Supports grammar in EBNF, which is easy to read and write
- Allows to choose different parsers
- Automatically resolves collisions of terminals
- Automatically track lines and columns in both rules and input code
- Has a standard library of terminals (like strings, numbers)

Grammar:

```
!Pip install lark
```

```python
from lark import Lark

main_grammar = r'''
preprocessor_hook: "%{$PREPROCESSOR" constant_expression "}%"

program: (external_declaration* | preprocessor_hook)

pragma: "_Pragma" "(" STRING_LITERAL ")" | "printf" "(" STRING_LITERAL (","
expression)* ")" ";" | "scanf" "(" STRING_LITERAL ("," "&" expression)* ")"
";"

?external_declaration: function_definition | declaration | ";" | pragma

function_definition: decl_specifier declarator block_statement

declaration: decl_specifier init_declarator_list ";"
   | decl_specifier ";"

init_declarator_list: init_declarator ("," init_declarator)*

init_declarator: declarator
   | declarator "=" initializer

initializer: assignment_expression
   | "{" initializer_list ","? "}" -> compound_initializer

initializer_list: initializer_item ("," initializer_item)*
initializer_item: designation? initializer
?designation: designator_list "="
?designator_list: designator designator_list | designator
designator: "[" constant_expression "]" -> index_member_reference
   | "." identifier_expr -> name_member_reference

declarator: STAR* direct_declarator

direct_declarator: identifier_expr
   | identifier_expr "[" assignment_expression? "]" -> array_declarator
   | identifier_expr? "(" param_list? ")" -> func_declarator

param_list: param_declaration ("," param_declaration)* ("," VARARGS)?

param_declaration: decl_specifier declarator
   | decl_specifier STAR* -> unnamed_param_declaration

// Use epsilon in child rules rather than make optional so we don't
```

```
// loose it from the tree
decl_specifier: storage_class_specifier type_qualifier type_specifier
storage_class_specifier: TYPEDEF | STATIC |
type_qualifier: CONST |

?type_specifier: struct_specifier | typedef_name

typedef_name: TYPEDEF_NAME
TYPEDEF_NAME.2: IDENT

?struct_specifier: struct_spec_reference | struct_spec_declaration
struct_spec_reference: "struct" identifier_expr
struct_spec_declaration: "struct" identifier_expr? "{"
struct_declaration_list "}"

struct_declaration_list: struct_declaration+
struct_declaration: type_specifier struct_declarator_list? ";"

struct_declarator_list: struct_declarator ("," struct_declarator)*

?struct_declarator: declarator

type_name: type_specifier STAR*

// So the transformer can distinguish between block-level declarations
// and global
block_declaration: declaration
block_statement: "{" (statement | block_declaration)* "}"

?statement: block_statement | statement_no_block

?statement_no_block: labelled_statement
   | expression ";" -> expression_statement
   | "if" "(" expression ")" statement ["else" statement] -> if_statement
   | "switch" "(" expression ")" "{" switch_case_fragment* "}" ->
switch_statement
   | "while" "(" expression ")" statement -> while_statement
   | "do" statement "while" "(" expression ")" ";" -> do_while_statement
   | "for" "(" expression? ";" expression? ";" expression? ")" statement ->
for_statement
   | "goto" identifier_expr ";" -> goto_statement
   | "continue" ";" -> continue_statement
   | "break" ";" -> break_statement
   | "return" expression? ";" -> return_statement
   | "sync" ";" -> sync_statement
```

```
   | pragma

switch_case_fragment: "case" constant_expression ":" switch_case_body?
   | "default" ":" switch_case_body? -> switch_default_fragment
?switch_case_body: block_statement | statement_no_block+

?labelled_statement: identifier_expr ":" statement -> label_statement
//    | "case" constant_expression ":" statement -> case_statement
//    | "default" ":" statement -> default_statement

?expression: assignment_expression
   | expression "," assignment_expression

?constant_expression: conditional_expression

?assignment_expression: conditional_expression
   | unary_expression (ASSIGN | ASSIGN_OP) assignment_expression

?conditional_expression: logical_or_expression ["?" expression ":"
conditional_expression]

?logical_or_expression: logical_and_expression
   | logical_or_expression LOG_OR_OP logical_and_expression -> binop_expr

?logical_and_expression: inclusive_or_expression
   | logical_and_expression LOG_AND_OP inclusive_or_expression ->
binop_expr

?inclusive_or_expression: exclusive_or_expression
   | inclusive_or_expression OR_OP exclusive_or_expression -> binop_expr

?exclusive_or_expression: and_expression
   | exclusive_or_expression XOR_OP and_expression -> binop_expr

?and_expression: equality_expression
   | and_expression AND_OP equality_expression -> binop_expr

?equality_expression: relational_expression
   | equality_expression EQ relational_expression -> binop_expr
   | equality_expression NEQ relational_expression -> binop_expr

?relational_expression: shift_expression
   | relational_expression REL_OP shift_expression -> binop_expr

?shift_expression: additive_expression
```

```
    | shift_expression SHIFT_OP additive_expression -> binop_expr

?additive_expression: multiplicative_expression
    | additive_expression ADD_OP multiplicative_expression -> binop_expr

?multiplicative_expression: cast_expression
    | multiplicative_expression (STAR | MUL_OP) cast_expression ->
binop_expr

?cast_expression: "(" type_name ")" cast_expression
    | unary_expression

?unary_expression: postfix_expression
    | INCREMENT_OP unary_expression -> pre_increment_expr
    | UNARY_OP cast_expression
    | "sizeof" ( "(" type_name ")" | unary_expression ) -> sizeof_expr

?postfix_expression: primary_expression
    | postfix_expression "[" expression "]" -> array_subscript_expr
    | postfix_expression "(" (assignment_expression (","
assignment_expression)*)? ")" -> function_call_expr
    | postfix_expression (DOT | ARROW) identifier_expr -> member_access_expr
    | postfix_expression INCREMENT_OP -> post_increment_expr

?primary_expression: identifier_expr
    | INT_CONSTANT -> int_literal
    | STRING_LITERAL+ -> string_literal
    | "(" expression ")"


identifier_expr: IDENT

VARARGS: "..."

TYPEDEF: "typedef"
STATIC: "static"
CONST: "const"

STAR.0: "*"
EQ.2: "=="
ASSIGN: "="
ASSIGN_OP: ASSIGN | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>=" | "&="
| "^=" | "|="

LOG_OR_OP: "||"
```

```
LOG_AND_OP: "&&"
OR_OP: "|"
XOR_OP: "^"
AND_OP: "&"
NEQ: "!="
REL_OP: "<=" | ">=" | "<" | ">"
SHIFT_OP: "<<" | ">>"
ADD_OP: "+" | "-"
MUL_OP: "*" | "/" | "%"
UNARY_OP: "&" | "*" | "+" | "-" | "~" | "!"
INCREMENT_OP: "++" | "--"
DOT: "."
ARROW: "->"

INT_CONSTANT: DEC_CONSTANT | OCT_CONSTANT | HEX_CONSTANT | BIN_CONSTANT |
CHAR_CONSTANT | "0"
DEC_CONSTANT: NON_ZERO_DIGIT DIGIT*
OCT_CONSTANT: "0" ("0".."7")+
HEX_CONSTANT: "0" ("x" | "X") ("0".."9" | "A".."F" | "a".."f")+
BIN_CONSTANT: "0" ("b" | "B") ("0" | "1")+

SINGLE_CHAR: (/(<?!\\)./ | "\\" /([abefnrtv\\'"?]|x[\da-fA-F]+|[0-7]{1,3})/
)

CHAR_CONSTANT: "'" SINGLE_CHAR "'" | "'" LETTER "'"

%import common.ESCAPED_STRING
STRING_LITERAL: ESCAPED_STRING

DIGIT: "0" .. "9"

NON_ZERO_DIGIT: "1" .. "9"

LETTER: "a".."z" | "A".."Z"
%import common.SIGNED_NUMBER
ID_START: LETTER | "_"

IDENT: ID_START (ID_START | DIGIT)*

WHITESPACE: " " | "\t" | "\f" | "\n"
%ignore WHITESPACE+

COMMENT: "//" /[^\n]/* | "/*" /(\S|\s)*?/ "*/"
%ignore COMMENT
'''
```

```
initial_grammar = r'''start : header_files
preprocessor_commands : "#include" | definition
definition : def (string)+ (header_files)*
def : "#define" | "#undef" | "#ifdef" | "#ifndef" | "#if" | "#else" |
"#elif" | "#endif" | "#error" | "#pragma"
header_files : preprocessor_commands (file_names)? (header_files)*
file_names : "<stdio.h>" | "<math.h>" | "<conio.h>" | "<stdlib.h>" |
"<string.h>" | "<ctype.h>" | "<time.h>" | "<float.h>" | "<limits.h>" |
"<wctype.h>"

%import common.SIGNED_NUMBER
letter : "a".."z" | "A".."Z"
char : letter | SIGNED_NUMBER | "}" | ")" | "]"
string : /[a-zA-Z0-9_.-]{2,}/ | (char)*

WHITESPACE: " " | "\t" | "\f" | "\n"
%ignore WHITESPACE+

COMMENT: "//" /[^\n]/* | "/" /(\S|\s)?/ "*/"
%ignore COMMENT
'''
```

Code:

```
par = Lark(grammar=initial_grammar, parser='earley', start='start')
parser = Lark(grammar=main_grammar, parser="lalr", start="program")
code = r'''
#include<stdio.h>
#include<math.h>
#define PI 3.14
void main(){
float a,b,c;float x=4.5;
scanf("%f%f%f",&a,&b,&c);
float d=(-b+sqrt(pow(b,2)-4*a*c))/(2*a);
printf("%f",d);
return;
}
'''
if (code.find('void') == -1):
   i_void = 10000
else:
   i_void = code.find('void')
if (code.find('int') == -1):
   i_int = 10000
else:
```

```
    i_int = code.find('int')
if (code.find('static') == -1):
    i_static = 10000
else:
    i_static = code.find('static')
if (code.find('long') == -1):
    i_long = 10000
else:
    i_long = code.find('long')
if (code.find('float') == -1):
    i_float = 10000
else:
    i_float = code.find('float')
if (code.find('double') == -1):
    i_double = 10000
else:
    i_double = code.find('double')
if (code.find('string') == -1):
    i_string = 10000
else:
    i_string = code.find('string')
if (code.find('bool') == -1):
    i_bool = 10000
else:
    i_bool = code.find('bool')
idx = [i_void, i_bool, i_int, i_static, i_long, i_float, i_double, i_string]
idx.sort()
min = idx[0]
code1 = code[:min]
code2 = code[min:]


print(par.parse(code1).pretty())
print(parser.parse(code2).pretty())
```

Understanding the Grammar:

The EBNF follows a syntax that is similar to that of the regular expression. It is explained in the following lines.

Syntax of Regular Expression:

There is some meaning to the special characters like ?,*,+ in the regular expression. These characters are used frequently in the grammar rules. That is why it is crucial to understand them first before reading the grammar rules.

- **.** : The special character '.' (dot) matches any character except a new line.
- ^: This symbol '^' (caret) matches the start of a string.
- **()**: The parenthesis are used to group some rules. For example, the "," (comma) and the rule 'param_declaration' are grouped in the following rule.
  `param_list: param_declaration ("," param_declaration)* ("," VARARGS)?`
- *****: It matches to zero or more repetitions of the preceding symbol. It means that in the above rule, the group ("," param_decalration) occur any number of times.
- **+:** This character is similar to *, except that it matches one or more repetitions of the preceding symbol. In the following example, the string can occur one or more times.
  `definition : def (string)+ (header_files)*`
- **?:** It matches to zero or one repetition of the preceding symbol. An alternative for this is to enclose the concerned symbol in square brackets ("[]").
- {m}: It specifies that the preceding symbol will occur exactly m times.
- {m,n}: It specifies that the preceding symbol will occur m to n times.
- \: It either escapes special characters and permits to match characters like '*' or '?' or signals a special sequence.
- \d: Matches any digit. It includes [0-9]
- \s: It matches Unicode whitespace characters. (that includes [\t\n\r\f\v])
- \S: Matches any character which is not a whitespace character. It is the opposite of \s.
- ->: This symbol is used to give an alternate name to a rule.

Terminals:

Terminals are used to match text into symbols. They must be enclosed within double quotes ("). For example, in the following rule, 'LETTER' matches any capital or small alphabets. The vertical bar ('|') is the OR operator. Names of the Terminal symbols must always be capitalised, i.e. Names of terminal symbols must be in Upper Case.

```
LETTER: "a".."z" | "A".."Z"
```

Non-Terminals:

The non-terminals match other non-terminals or terminals. For example, in the following rule, the non-terminal 'param_list' matches to another non-terminal 'param_declaration' followed by zero or more occurrences of "," and 'param_declaration' followed by either zero or one occurrence of "," and 'VARARGS'.

```
param_list: param_declaration ("," param_declaration)* ("," VARARGS)?
```

Production Rules:

The production rules follow the standard EBNF grammar rules. A top-down approach is followed where we start the program with a start rule, including references to non-terminal rules and terminal symbols that follow, often using a recursive call to specific repetitive rules/symbols that can occur multiple times throughout the code. The terminal symbols depict the termination of the rule where they are used.

Miscellaneous:

Lark contains standard libraries that offer certain Terminal symbols such as strings, numbers, names, etc., and these do not need to be defined additionally in the grammar. These libraries are used by importing the concerned library using the "%import" command.
Strings are present in the "common.ESCAPED_STRING" library, and numbers are present in the "common.SIGNED_NUMBER" library which are among the most common terminals used.

```
%import common.ESCAPED_STRING
%import common.SIGNED_NUMBER
```

Thought Process:

To develop concise grammar, we first read and observed the grammar rules of the ANSI C grammar, which can be found on This link. Then we observed how most of the C programs start and the general structure of any C program. In general, the C program will start with preprocessor commands like #include or #define. Then we made some rules that can handle these lines of code. After this much part of the code, mainly function definition or global variable declarations, will start. For that part, we had to incorporate the string terminals from the library of Lark because the function's name can be anything. Also, we had to make some

rules which took care of the data type as the function declaration starts with the data type that the function will return.
The further grammar starts for the actual program, which starts from the function definition or the global variable declaration has been inspired from the ANSI C grammar referenced above. The program starts with the first line of code being a function definition/the main definition or an "external_declaration", i.e., essentially a global variable declaration. It then moves on following production rules to non-terminal rules, recursive rules brought into effect by using the "*" or "+" symbols and terminal rules/symbols depicting termination of the rule. These rules cover everything comprising declarations, data types, statements such as loops, conditional statements, print and scan statements, expressions such as arithmetic and logical expressions and so on.
The test code is split into two parts, the preprocessing part that includes the "#include", "#define" and other such commands/preprocessors and the actual code part starting from the function definition or global variable declaration as stated above. Two separate grammars were generated to individually handle the split parts of the input, resulting in the formation of two parse trees relating to each part. The reason behind this was to avoid collisions in the LALR parser. LALR parsing does not allow the same terminal symbols to be used in other terminal rules. This problem could only be solved by generating two different grammars and splitting the code to parse each part individually. This would not affect the accuracy of the parse tree as the preprocessor and program part are syntactically unrelated with respect to the structure of the parse tree.

How to use:

To successfully use the parser, the user must download the Lark library by the command

pip install lark

The user must have pip installed. If it is not installed, download it from [here](). Or the user can install lark directly from the IDE settings.
To test your code, copy it and paste it into the variable named 'code' in the main.py file.

Results:

We tried 15 C programs out of which 10 were syntactically correct and 5 were incorrect. The programs and the result of the parser are as shown below.

1. ```
   #include<stdio.h>
   #define PI 3.14
   void main(){
   ```
   ```

```
printf("Hello world!!!");
}
```

Result : Output_1.pdf

2.
```
#include<stdio.h>
#define myGlobalInt 5

int main()
{
   int myLocalInt = 7;
   int myProd = myLocalInt * myGlobalInt;
   printf("My Calculator => %d * %d = %04d \n", myGlobalInt, myLocalInt,
myProd);
   return 0;
}
```

Result : Output_2.pdf

3.
```
#include<stdio.h>
#include<math.h>
#define PI 3.14
void main(){
int a=5,b=6,c=10;
if(a<b && a<c){
printf("%d",a);
}
if(b>a && b<c){
printf("%d",b);
}
else
printf("%d",c);
return;
}
```

Result: Output_3.pdf

4.
```
#include<stdio.h>
#include<math.h>
#define PI 3.14
void main(){
int a=5,b=6,c=10;
if(a<b && a<c){
printf("%d",a);
```

```
}
else if(b>a && b<c){
printf("%d",b);
}
else if(c<a && c>b)
printf("%d",c);
return;
}
```

Result: Output_4.pdf

5. ```
   #include<stdio.h>
   #include<math.h>
   #define PI 3.14

   int a=3;
   char c='c';
   float b=4.5;
   void main(){
   int n;
   scanf("%d",&n);
   switch(n){
   case 1:
      printf("hello world!");
      break;
   case 2:
      printf("hello peter");
      break;
   default:
      printf("%f",b);
   }
   return;
   }
   ```

   Result: Output_5.pdf

6. ```
   #include<stdio.h>
   #include<math.h>
   #define PI 3.14

   int a=3;
   char c='c';
   float b=4.5;
   ```

```
void main(){
int i=0,n;
scanf("%d",&n);
for(;i<n;i++){
if(i<a)
   continue;
else
   printf("%d",i);
}
return;
}
```

Result: Output_6.pdf

7.
```
#include<stdio.h>
#include<math.h>
#define PI 3.14

int a=3;
char c='c';
float b=4.5;
void main(){
int i=0,n;
scanf("%d",&n);
while(i<n){
if(i<a)
   continue;
else
   printf("%d",i);
++i;
}
return;
}
```

Result: Output_7.pdf

8.
```
#include<stdio.h>
#include<math.h>
#define PI 3.14

int a=3;
char c='c';
float b=4.5;
```

```
void main(){
int i=0,n;
scanf("%d",&n);
do{
if(i<a)
   continue;
else
   printf("%d",i);
++i;
}
while(i<n);
return;
}
```

Result: Output_8.pdf

9.
```
#include<stdio.h>
#include<math.h>
#define PI 3.14

int a=3;
char c='c';
float b=4.5;

int max(int a,int b){
   if(a<b)
   return b;
   else
   return a;
}

/*void swap(int *x,int *y)
{
   int t;
    t   = *x;
   *x   = *y;
   *y   =  t;
}*/

void main(){
int i=0,n1,n2;
scanf("%d%d",&n1,&n2);
printf("%d",max(n1,n2));
```

```
//swap(&n1,&n2);
//printf("After  Swapping: num1 is: %d, num2 is: %d\n",n1,n2);
printf("%d",min(n1,n2));
return;
}
int min(int a,int b){
   if(a<b)
   return a;
   else
   return b;
}
```

Result: Output_9.pdf

10.
```
#include<stdio.h>
#include<math.h>
#define PI 3.14
void swap(int *x,int *y)
{
   int t=*x;
   *x=*y;
   *y=t;
}

void main(){
int n1,n2;
scanf("%d%d",&n1,&n2);
printf("%d",fib(n1));
int *a=&n1,*b=&n2;
swap(a,b);
}

int fib(int n){
if(n==0 || n==1)
   return n;
return fib(n-1)+fib(n-2);
}
```

Result: Output_10.pdf

Following are the syntactically incorrect programs.

11.
```
#include <stdio.h>
#define myGlobalInt

int main()
{
   int myLocalInt = 7
   int myProd = myLocalInt myGlobalInt;
   printf("My Calculator => %d * %d = %04d \n", myGlobalInt, myLocalInt, myProd);
   return 0;
}
```

Result: Output_11.pdf

12.
```
#include<stdio.h>
#include<math.h>
#define PI 3.14
void main(){
int a=5,b=6,c=10;
if(a<b && a<c{
printf("%d",a);
}
else if(b>a && b<c){
printf("%d",b);
}
else if(c<a && c>b)
printf("%d",c);
return;
}
```

Result: Output_12.pdf

13.
```
#include<stdio.h>
#include<math.h>
#define PI 3.14

int a=3;
char c='c';
float b=4.5;
void main(){
int n;
scanf("%d",&n);
switch(n){
```

```
case 1:
   printf("hello world!");
   break;
case :
   printf("hello peter");
   break;
default:
   printf("%f",b);
}
return;
}
```

Result: Output_13.pdf

14.
```
#include<stdio.h>
#include<math.h>
#define PI 3.14

int a=3;
char c='c';
float b=4.5;
void main(){
int i=0,n;
scanf("%d",&n);
for(i<n;i++){
if(i<a)
   continue;
else
   printf("%d",i);
}
return;
}
```

Result: Output_14.pdf

15.
```
#include<stdio.h>
#include<math.h>
#define PI 3.14

int a=3;
char c='c';
float b=4.5;
```

```
int max(int a,int b){
    if(a<b)
    return b;
    else
    return a;
}

void main(){
int i=0,n1,n2;
scanf("%d%d",n1,&n2);
printf("%d",max(n1,n2));
printf("%d",min(n1,n2));
return;
}
int min(int a,int b){
    if(a<b)
    return a;
    else
    return b;
}
```

Result: Output_15.pdf

Discrepancies:

The following codes are those which the grammar cannot handle, i.e. even though they are syntactically correct, the parser will throw an error. For the first code below, the variable i is defined in the for loop, which is still a valid syntax, throwing an error. In the second one, variable i is not initialised at the time of declaration, which is also valid syntax, but the parser still gives an error. The third code is syntactically wrong because the data type of variable i is not specified; the parsed tree is still generated. In the fourth code, the swap function is called with the arguments as &n1 and &n2, which is valid still; the parser throws an error. If we use some pointers to store the address of n1 and n2 and then pass these pointers to the arguments to the function (as in example 10 above), it will be parsed successfully. Float/Double type variables can be declared but cannot be initialised with decimal values.

1. ```
   #include<stdio.h>
   #include<math.h>
   #define PI 3.14

   void main(){
   for(int i=0;i<4;i++){
   ```
   ```

```
printf("%d",i);
}
return;
}
```

Result: disc_1.pdf

2. ```
   #include<stdio.h>
   #include<math.h>
   #define PI 3.14

   void main(){
   int i;
   i=0;
   for(;i<4;i++){
   printf("%d",i);
   }
   return;
   }
   ```

   Result: disc_2.pdf

3. ```
   #include<stdio.h>
   #include<math.h>
   #define PI 3.14

   void main(){
   for(i=0;i<4;i++){
   printf("%d",i);
   }
   return;
   }
   ```

   Result: disc_3.pdf

4. ```
   #include<stdio.h>
   #include<math.h>
   #define PI 3.14
   void swap(int *x,int *y)
   {
      int t=*x;
      *x=*y;
      *y=t;
   ```
   ```

```
}

void main(){
int n1,n2;
scanf("%d%d",&n1,&n2);
printf("%d",fib(n1));

swap(&n1,&n2);
}

int fib(int n){
if(n==0 || n==1)
   return n;
return fib(n-1)+fib(n-2);
}
```

Result: disc_4.pdf

5.
```
#include<stdio.h>
#include<math.h>
#define PI 3.14
void main(){
float a,b,c;float x=4.5;
scanf("%f%f%f",&a,&b,&c);
float d=(-b+sqrt(pow(b,2)-4*a*c))/(2*a);
printf("%f",d);
return;
}
```

Result: disc_5.pdf

Observations:

Although we said that the grammar is concise, still it contains many rules which are complicatedly linked with each other. The above discrepancies are nothing but the bugs in these rules, which we tried to resolve, but we cannot understand precisely at which rule the problem is as of now. The parser works reasonably well with simple as well as complex C programs. While developing these rules, we learned how to write the grammar in EBNF, the syntax of regular expressions, Different parsers like CYK and Earley, The complexity and efficiency, and limitations of these parsers on different grammar.

Contributions:

Aashish Raj :

- Finding valuable resources to decide which parser to use.
- Studied about Lark.
- Worked on developing the grammar.
- Debugging the final code and writing the report.

Tanmay Vaghale :

- Studied about lex and yacc.
- Worked on developing the grammar.
- Debugging the final code.
- writing the report.

Gaurav Jha:

- Helped in debugging the code.
- Did preliminary research.

Bibliography:

- https://lark-parser.readthedocs.io/en/latest/: Documentation of Lark
- https://github.com/eliben/pycparser: pycparser
- https://lark-parser.readthedocs.io/_/downloads/en/latest/pdf/: Lark Documentation by Erez Shinan
- Information technology - Syntactic metalanguage - Extended BNF INTERNATIONAL STANDARD ISO/IEC First edition 1996-l 2-l 5
- https://github.com/antlr/grammars-v4/blob/master/c/C.g4: ANSI C grammar

THANK YOU!