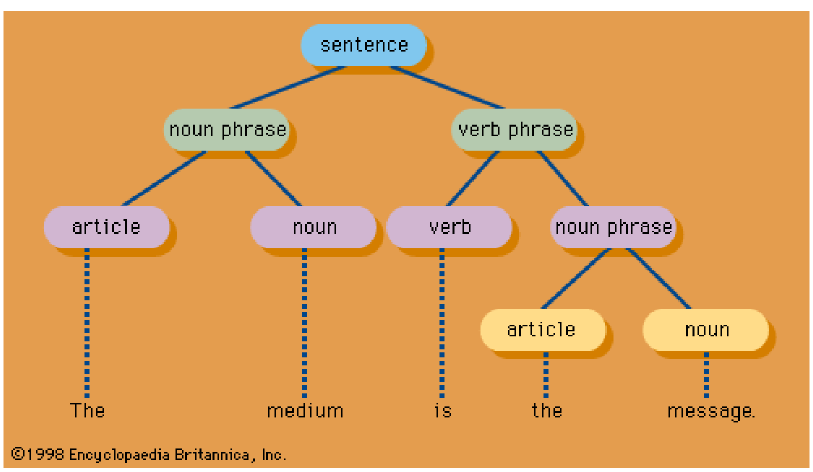
Problem Statement:

With regard to this topic, your team has to come up with a grammar (ideally, a reduced and optimized one) that can parse a C-language program. Given a C-program, your code has to parse it and output the parsing steps (which terminal symbol is reached or which production rule is applied for a non-terminal symbol, in each step). After successfully parsing a C-code, your grammar/parser has to tell whether the input C-code was syntactically correct or not. Your project report should contain the grammar (make sure that it's well-structured, so that someone reading the report can understand well), a list of terminal and non-terminal symbols, production rules, etc. Explain clearly your thought process on how you came up with this grammar and elaborate on how your grammar works. You are encouraged to add anything else that you feel is important.

Parsing

Parsing or syntactic analysis is the process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of formal grammar. In simple words, parsing is the process to break down a set of information or instructions containing complex steps into smaller and simple steps for example A simple parsing of a sentence in accordance with English grammar rules, "The medium is the message" would look something like:



The keyword here is 'Grammar' and the picture represents a way of showing the parsed sentence in the form of a tree.

Why Parsing?

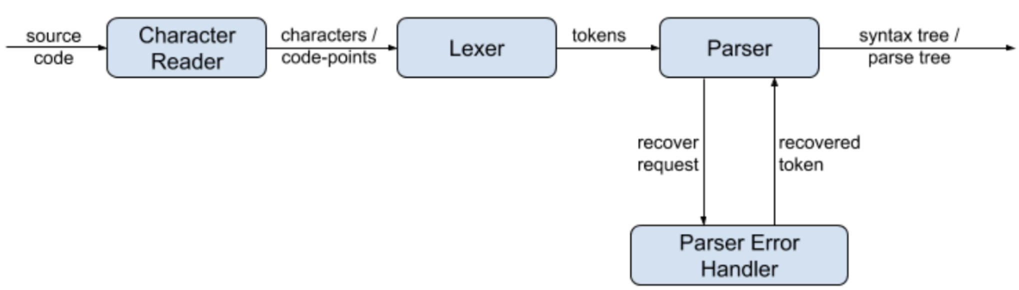
So the obvious question then is that where does this "Parsing" process exactly fits in the fields of Natural language processing and software engineering. Many data-driven parsing approaches developed for natural languages are robust and have quite a high accuracy when applied to the parsing of software. software maintenance is supported by numerous program analysis tools. Maintenance tasks include program comprehension (understanding unknown code for fixing bugs or further development), quality assessment (judging code, e.g., in code reviews), and reverse-engineering (reifying the design documents for given source code). To extract information from the programs, the tools first parse the program code and produce an abstract syntax tree (AST) for further analysis and abstraction.

Plan of Action

There are basically three major steps involved in the Parsing of the C-code:

Context-Free Grammar - Formulating a series of context-free grammars that can potentially parse a complete C program taking various cases into consideration such as block statements, expressions, loops, etc. While writing the grammar we also had to take into account that cases don't overlap each other like the grammar for "while" loop and "do while" loop had to be different. This grammar forms the base of our program and is the most important part considering that the "Parsing" of the code depends a lot on the grammar used.

Lexing - The very first step after forming the context-free grammar is to divide the input program code into a group of semantically meaningful units called tokens. Intuitively each word and symbol in the source code conveys some meaning and we want to extract these tokens into an ordered list for further processing. The output of a lexical analyzer is this sequence of tuples (line numbers are only relevant for error reporting). Each tuple contains the token type and the name of the token.



As we can see in the figure, our input C-code is regarded as datastream into the lexer which outputs tokens that are then crucial for the Parser to output a (parsed) syntax tree.

Parsing - The Syntax Analyzer (aka Parser) is the heart of the compiler. It calls the other modules, such as the lexical analyzer to fetch the next token, and acts as the main loop of the compiler. The parser is responsible for reading the tokens from the lexer and producing the parse-tree. It gets the next token from the lexer, analyzes it, and compares it against a defined grammar. Then decide which of the grammar rules should be considered, and continue to parse according to the grammar rule chosen. However, this is not always very straightforward, as sometimes it is not possible to determine which path to take only by looking at the next token. Thus, the parser may have to check a few tokens into the future as well, to decide the path or the grammar rule which has to be considered.

Parsing of an Expression:

We take up an expression as an example to parse so that we can understand the steps involved in parsing and how we were able to formulate grammar for the parsing of C-code:

Consider the following expression to be parsed:

$$\frac{\cos\left(x^2 + 2\pi\theta\right)}{\sqrt{a^2 + b^2}}$$

Constraints:
To keep the code interesting yet simple, we limit the supported math syntax to the following items (for this particular example):

- Latin letter variables: a through z and A through Z.
- Greek letter variables, lowercase alpha through omega and uppercase Alpha through Omega.
- Numeric values in the form of integers (724), floating point numbers (4.39254), and scientific notation (6.27e+20).
- The binary operators addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (^).
- The unary prefix operators + (positive) and - (negative). Note that the same symbol can mean different things depending on context!

- The functions cosine (cos), sine (sin), absolute value (abs), and square root (sqrt). It will be easy for you to add more functions later, if you want.
- The use of parentheses () to override operator precedence.

Grammar:
We will be using BNF Grammar notation for representing grammar for this particular example as it is easy to read and interpret the meaning.

Grammar definitions:
Each definition starts with a name like expr followed by the ::= symbol, meaning "is defined as." To the right of each ::= is the syntax allowed for the concept named to the left of the ::=.

Here is the BNF grammar for our expression parser:

```
expr ::= mulexpr { addop mulexpr }
addop ::= "+" | "-"
mulexpr ::= powexpr { mulop powexpr }
mulop ::= "*" | "/"
powexpr ::= "-" powexpr | "+" powexpr | atom ["^" powexpr ]
atom ::= ident [ "(" expr ")" ] | numeric | "(" expr ")"
numeric ::= /[0-9]+(\.[0-9]*)?([eE][\+\-]?[0-9]+)?/
ident ::= /[A-Za-z_][A-Za-z_0-9]*/
```

Explanation:

We will start with the first definition, expr, which represents any mathematical expression to be parsed. When you see --->>> `mulexpr { addop mulexpr } --->>>` it means that an expression consists of a mulexpr, followed by zero or more addop mulexpr pairs. In other words, these are all valid expr strings:

```
mulexpr
mulexpr addop mulexpr
mulexpr addop mulexpr addop mulexpr
mulexpr addop mulexpr addop mulexpr addop mulexpr
```

There are cases where ParseExpr needs to recognize an expression inside another expression. For example, consider the boldfaced part of this grammar rule:

`atom ::= ident [ "(" expr ")" ] | numeric |` **`"(" expr ")"`**

Precedence is implicit in the grammar based on how the rules are nested. The fact that

```
expr ::= mulexpr { addop mulexpr }
```

is defined in terms of

```
mulexpr ::= powexpr { mulop powexpr }
```

tells you that this grammar places multiplication at a higher precedence than addition, because multiplication is processed at a deeper level of recursion by the parser.

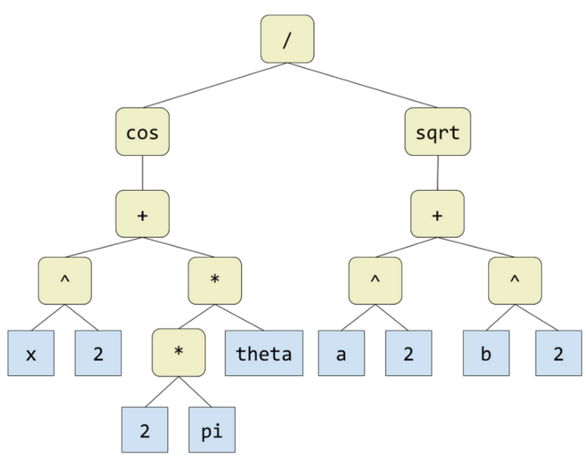
Left associative operators group left-to-right. This is most important with subtraction and division, where the order matters. For example: a-b-c-d is interpreted as ((a-b)-c)-d. This emerges from the grammar using simple iteration via the “zero or more” grammar rules using braces, as in

```
expr ::= mulexpr { addop mulexpr }
```

When the parser sees a^b^c^d, we want it to interpret this string as a^(b^(c^d)). The grammar uses recursion rather than iteration to make sure the right side is deeper in the resulting parse tree. See the boldfaced part of the following rule.

powexpr ::= "-" powexpr | "+" powexpr | **atom ["^" powexpr]**

The code starts by parsing the user’s input text into a parse tree. Here is what the parse tree looks like for the expression shown above:



Program Structures that our code works on successfully and on which it fails to Parse:

Bison reads tokens, it pushes them onto a stack along with their semantic values. The stack is called the parser stack. Pushing a token is traditionally called shifting. The parser tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol. This kind of parser is known in the literature as a bottom-up parser. Bison reads a specification of a context-free language, warns about any parsing ambiguities, and generates a parser that reads sequences of tokens and decides whether the sequence conforms to the syntax specified by the grammar.

Program structures that our Code works on perfectly:

- Variable Declaration
- Variable arithmetic
- Arithmetic expressions
- Expressions with Logical Operators
- Print statements
- Scan statements
- Nested Arithmetic expressions
- Complex mathematical expressions
- ‘For’ loop
- ‘Do-while’ loop
- While loop
- Nested loops (All combinations)
- Switch-case iteration statements
- Function declarations and definitions
- Function Calls
- Jump statements (like break, continue and return statements)
- Pointer declaration
- Pointer arithmetic

Program structures that our Code is unable to parse on:

1. Header Files and Libraries:
   Unfortunately, bison fails to recognise the declaration of standard header files and libraries (like <stdio.h>, <stdlib.h>, <string.h>, <math.h> etc) and as a consequence of which, it is unable to find a suitable token for the functions that are enveloped inside the libraries mentioned in the code. Examples being acos(), acosh(), ceil(), floor(), exp(), etc functions from <math.h>; strcat(), strcmp(), strlen(), strcpy(), etc functions from the <string.h> library for strings; etc. the code fails to recognize these (and many more that libraries declare as their set of functions) functions and thus cannot match them with any of the predefined tokens after lexing during parsing.

2. Due to time constraint, we failed to come up with an appropriate grammar for the following scenarios:
   a. Struct data type and it's consequent implementation structures.
   b. Multiple function calls inside a function
   c. Recursion

Future Plans:

Our future plan for this project includes tackling the challenges of including functions from different header files and libraries and also introduce 'struct' as a data type to parse more C functions that can include direct implementation of 'struct' data type like stacks and queues.

For including a new data type, we would have to formulate new grammar for the said data type and due to which the affecting grammars might change.

For inclusion of header file and library functions, we plan to hard code all the functions from the main header files (<stdio.h>, <stdlib.h>, <string.h>, <math.h>) as tokens into our lexer and assume them to behave as functions for the code to parse.

Lexical Grammar

1. **Lexical Elements**

token:
- *keyword*
- *identifier*
- *constant*
- *string-literal*
- *punctuator*

2. **Keywords**

keyword: one of

auto	**enum**	**restrict**	**unsigned**
break	**extern**	**return**	**void**
case	**float**	**short**	**volatile**
char	**for**	**signed**	**while**
const	**goto**	**sizeof**	**_Bool**
continue	**if**	**static**	**_Complex**
default	**inline**	**struct**	**_Imaginary**
do	**int**	**switch**	
double	**long**	**typedef**	
else	**register**	**union**	

3. **Identifiers**

identifier:
- *identifier-nondigit*
- *identifier identifier-nondigit*
- *identifier digit*

identifier-nondigit: one of

_	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**
	n	**o**	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**	**x**	**y**	**z**
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**
	N	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**

digit: one of

0 1 2 3 4 5 6 7 8 9

4. **Constants**

constant:
- *integer-constant*
- *floating-constant*
- *enumeration-constant*
- *character-constant*

integer-constant:
- *nonzero-digit*
- *integer-constant digit*

nonzero-digit: one of

1 2 3 4 5 6 7 8 9

floating-constant:
- $\textit{fractional-constant exponent-part}_{opt}$
- *digit-sequence exponent-part*

fractional-constant:
- $\textit{digit-sequence}_{opt}$ **.** *digit-sequence*
- *digit-sequence* **.**

exponent-part:
- **e** $\textit{sign}_{opt}$ *digit-sequence*
- **E** $\textit{sign}_{opt}$ *digit-sequence*

sign: one of

+ −

digit-sequence:
digit
digit-sequence digit

enumeration-constant:
identifier

character-constant:
' *c-char-sequence* '

c-char-sequence:
c-char
c-char-sequence c-char

c-char:
any member of the source character set except
the single-quote ', backslash \, or new-line character
escape-sequence

escape-sequence: one of
**\'   \"   \?   **
\a   \b   \f   \n   \r   \t   \v

5. String literals

string-literal:
" $s\text{-}char\text{-}sequence_{opt}$ "

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any member of the source character set except
the double-quote ", backslash \, or new-line character
escape-sequence

6. Punctuators

punctuator: one of

```
[  ]  (  )  {  }  .  ->
++  --  &  *  +  -  ~  !
/  %  <<  >>  <  >  <=  >=  ==  !=  ^  |  &&  ||
?  :  ;  ...
=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=
,  #
```

Grammar Definitions used to Parse the C codes:

Rule 0 S' -> translation_unit
Rule 1 translation_unit -> external_declaration
Rule 2 translation_unit -> translation_unit external_declaration
Rule 3 external_declaration -> function_definition
Rule 4 external_declaration -> declaration
Rule 5 function_definition -> declaration_specifiers declarator declaration_list compound_statement
Rule 6 function_definition -> declaration_specifiers IDENTIFIER LEFTBRACE RIGHTBRACE compound_statement
Rule 7 function_definition -> declarator declaration_list compound_statement
Rule 8 function_definition -> declarator compound_statement
Rule 9 declaration_list -> declaration
Rule 10 declaration_list -> declaration_list declaration
Rule 11 declarator -> pointer direct_declarator
Rule 12 declarator -> direct_declarator
Rule 13 direct_declarator -> IDENTIFIER
Rule 14 direct_declarator -> LEFTBRACE declarator RIGHTBRACE
Rule 15 direct_declarator -> direct_declarator LSQPAREN constant_expression RSQPAREN
Rule 16 direct_declarator -> direct_declarator LSQPAREN RSQPAREN
Rule 17 direct_declarator -> direct_declarator LEFTBRACE parameter_type_list RIGHTBRACE
Rule 18 direct_declarator -> direct_declarator LEFTBRACE identifier_list RIGHTBRACE
Rule 19 direct_declarator -> direct_declarator LEFTBRACE RIGHTBRACE
Rule 20 primary_expression -> IDENTIFIER
Rule 21 primary_expression -> CONSTANT
Rule 22 primary_expression -> STRING_LITERAL
Rule 23 primary_expression -> LEFTBRACE expression RIGHTBRACE
Rule 24 postfix_expression -> primary_expression
Rule 25 postfix_expression -> postfix_expression LSQPAREN expression RSQPAREN
Rule 26 postfix_expression -> postfix_expression LEFTBRACE RIGHTBRACE
Rule 27 postfix_expression -> postfix_expression LEFTBRACE argument_expression_list RIGHTBRACE
Rule 28 postfix_expression -> postfix_expression DOT IDENTIFIER
Rule 29 postfix_expression -> postfix_expression PTR_OP IDENTIFIER

Rule 30 postfix_expression -> postfix_expression INC_OP
Rule 31 postfix_expression -> postfix_expression DEC_OP
Rule 32 argument_expression_list -> assignment_expression
Rule 33 argument_expression_list -> argument_expression_list COMMA assignment_expression
Rule 34 unary_expression -> postfix_expression
Rule 35 unary_expression -> INC_OP unary_expression
Rule 36 unary_expression -> DEC_OP unary_expression
Rule 37 unary_expression -> unary_operator cast_expression
Rule 38 unary_expression -> SIZEOF unary_expression
Rule 39 unary_expression -> SIZEOF LEFTBRACE type_name RIGHTBRACE
Rule 40 unary_operator -> ADDRESS
Rule 41 unary_operator -> MULT
Rule 42 unary_operator -> PLUS
Rule 43 unary_operator -> MINUS
Rule 44 unary_operator -> COMPLEMENT
Rule 45 unary_operator -> NOT
Rule 46 cast_expression -> unary_expression
Rule 47 cast_expression -> LEFTBRACE type_name RIGHTBRACE cast_expression
Rule 48 multiplicative_expression -> cast_expression
Rule 49 multiplicative_expression -> multiplicative_expression MULT cast_expression
Rule 50 multiplicative_expression -> multiplicative_expression DIVIDE cast_expression
Rule 51 multiplicative_expression -> multiplicative_expression MODULO cast_expression
Rule 52 additive_expression -> multiplicative_expression
Rule 53 additive_expression -> additive_expression PLUS multiplicative_expression
Rule 54 additive_expression -> additive_expression MINUS multiplicative_expression
Rule 55 shift_expression -> additive_expression
Rule 56 shift_expression -> shift_expression LEFT_OP multiplicative_expression
Rule 57 shift_expression -> shift_expression RIGHT_OP multiplicative_expression
Rule 58 relational_expression -> shift_expression
Rule 59 relational_expression -> relational_expression LT shift_expression
Rule 60 relational_expression -> relational_expression GT shift_expression
Rule 61 relational_expression -> relational_expression LE_OP shift_expression
Rule 62 relational_expression -> relational_expression GE_OP shift_expression
Rule 63 equality_expression -> relational_expression
Rule 64 equality_expression -> equality_expression EQ_OP relational_expression
Rule 65 equality_expression -> equality_expression NE_OP relational_expression

Rule 66 and_expression -> equality_expression
Rule 67 and_expression -> and_expression ADDRESS equality_expression
Rule 68 exclusive_or_expression -> and_expression
Rule 69 exclusive_or_expression -> exclusive_or_expression XOR and_expression
Rule 70 inclusive_or_expression -> exclusive_or_expression
Rule 71 inclusive_or_expression -> inclusive_or_expression OR exclusive_or_expression
Rule 72 logical_and_expression -> inclusive_or_expression
Rule 73 logical_and_expression -> logical_and_expression AND_OP inclusive_or_expression
Rule 74 logical_or_expression -> logical_and_expression
Rule 75 logical_or_expression -> logical_or_expression OR_OP logical_and_expression
Rule 76 conditional_expression -> logical_or_expression
Rule 77 conditional_expression -> logical_or_expression C_OP expression SIGNIFY conditional_expression
Rule 78 assignment_expression -> conditional_expression
Rule 79 assignment_expression -> unary_expression assignment_operator assignment_expression
Rule 80 assignment_operator -> ASSIGN
Rule 81 assignment_operator -> MUL_ASSIGN
Rule 82 assignment_operator -> DIV_ASSIGN
Rule 83 assignment_operator -> MOD_ASSIGN
Rule 84 assignment_operator -> ADD_ASSIGN
Rule 85 assignment_operator -> SUB_ASSIGN
Rule 86 assignment_operator -> LEFT_ASSIGN
Rule 87 assignment_operator -> RIGHT_ASSIGN
Rule 88 assignment_operator -> AND_ASSIGN
Rule 89 assignment_operator -> XOR_ASSIGN
Rule 90 assignment_operator -> OR_ASSIGN
Rule 91 expression -> assignment_expression
Rule 92 expression -> expression COMMA assignment_expression
Rule 93 constant_expression -> conditional_expression
Rule 94 declaration -> declaration_specifiers SEMICOLON
Rule 95 declaration -> declaration_specifiers init_declarator_list SEMICOLON
Rule 96 declaration_specifiers -> storage_class_specifier
Rule 97 declaration_specifiers -> storage_class_specifier declaration_specifiers_opt
Rule 98 declaration_specifiers -> type_specifier
Rule 99 declaration_specifiers -> type_specifier declaration_specifiers_opt
Rule 100 declaration_specifiers -> type_qualifier

Rule 101 declaration_specifiers -> type_qualifier declaration_specifiers_opt
Rule 102 declaration_specifiers_opt -> declaration_specifiers
Rule 103 declaration_specifiers_opt -> end_token
Rule 104 end_token -> <empty>
Rule 105 init_declarator_list -> init_declarator
Rule 106 init_declarator_list -> init_declarator_list COMMA init_declarator
Rule 107 init_declarator -> declarator
Rule 108 init_declarator -> declarator ASSIGN initializer
Rule 109 storage_class_specifier -> TYPEDEF
Rule 110 storage_class_specifier -> EXTERN
Rule 111 storage_class_specifier -> STATIC
Rule 112 storage_class_specifier -> AUTO
Rule 113 storage_class_specifier -> REGISTER
Rule 114 type_specifier -> VOID
Rule 115 type_specifier -> CHAR
Rule 116 type_specifier -> SHORT
Rule 117 type_specifier -> INT
Rule 118 type_specifier -> LONG
Rule 119 type_specifier -> FLOAT
Rule 120 type_specifier -> DOUBLE
Rule 121 type_specifier -> SIGNED
Rule 122 type_specifier -> UNSIGNED
Rule 123 type_specifier -> struct_or_union_specifier
Rule 124 type_specifier -> TYPE_NAME
Rule 125 struct_or_union_specifier -> struct_or_union IDENTIFIER LPAREN struct_declaration_list RPAPREN
Rule 126 struct_or_union_specifier -> struct_or_union LPAREN struct_declaration_list RPAPREN
Rule 127 struct_or_union_specifier -> struct_or_union IDENTIFIER
Rule 128 struct_or_union -> STRUCT
Rule 129 struct_or_union -> UNION
Rule 130 struct_declaration_list -> struct_declaration
Rule 131 struct_declaration_list -> struct_declaration_list struct_declaration
Rule 132 struct_declaration -> specifier_qualifier_list struct_declarator_list SEMICOLON
Rule 133 specifier_qualifier_list -> type_specifier specifier_qualifier_list
Rule 134 specifier_qualifier_list -> type_specifier

Rule 135 specifier_qualifier_list -> type_qualifier specifier_qualifier_list
Rule 136 specifier_qualifier_list -> type_qualifier
Rule 137 struct_declarator_list -> struct_declarator
Rule 138 struct_declarator_list -> struct_declarator_list COMMA struct_declarator
Rule 139 struct_declarator -> declarator
Rule 140 struct_declarator -> SIGNIFY constant_expression
Rule 141 struct_declarator -> declarator SIGNIFY constant_expression
Rule 142 type_qualifier -> CONST
Rule 143 type_qualifier -> VOLATILE
Rule 144 pointer -> MULT
Rule 145 pointer -> MULT type_qualifier_list
Rule 146 pointer -> MULT pointer
Rule 147 pointer -> MULT type_qualifier_list pointer
Rule 148 type_qualifier_list -> type_qualifier
Rule 149 type_qualifier_list -> type_qualifier_list type_qualifier
Rule 150 parameter_type_list -> parameter_list
Rule 151 parameter_list -> parameter_declaration
Rule 152 parameter_list -> parameter_list COMMA parameter_declaration
Rule 153 parameter_declaration -> declaration_specifiers declarator
Rule 154 parameter_declaration -> declaration_specifiers abstract_declarator
Rule 155 parameter_declaration -> declaration_specifiers
Rule 156 identifier_list -> IDENTIFIER
Rule 157 identifier_list -> identifier_list COMMA IDENTIFIER
Rule 158 type_name -> specifier_qualifier_list
Rule 159 type_name -> specifier_qualifier_list abstract_declarator
Rule 160 abstract_declarator -> pointer
Rule 161 abstract_declarator -> direct_abstract_declarator
Rule 162 abstract_declarator -> pointer direct_abstract_declarator
Rule 163 direct_abstract_declarator -> LEFTBRACE abstract_declarator RIGHTBRACE
Rule 164 direct_abstract_declarator -> LSQPAREN RSQPAREN
Rule 165 direct_abstract_declarator -> LSQPAREN constant_expression RSQPAREN
Rule 166 direct_abstract_declarator -> direct_abstract_declarator LSQPAREN RSQPAREN
Rule 167 direct_abstract_declarator -> direct_abstract_declarator LSQPAREN constant_expression RSQPAREN
Rule 168 direct_abstract_declarator -> LEFTBRACE RIGHTBRACE
Rule 169 direct_abstract_declarator -> LEFTBRACE parameter_type_list RIGHTBRACE

Rule 170 direct_abstract_declarator -> direct_abstract_declarator LEFTBRACE RIGHTBRACE
Rule 171 direct_abstract_declarator -> direct_abstract_declarator LEFTBRACE parameter_type_list RIGHTBRACE
Rule 172 initializer -> assignment_expression
Rule 173 initializer -> LPAREN initializer_list LPAREN
Rule 174 initializer -> LPAREN initializer_list COMMA LPAREN
Rule 175 initializer_list -> initializer
Rule 176 initializer_list -> initializer_list COMMA initializer
Rule 177 statement -> labeled_statement
Rule 178 statement -> compound_statement
Rule 179 statement -> expression_statement
Rule 180 statement -> selection_statement
Rule 181 statement -> iteration_statement
Rule 182 statement -> jump_statement
Rule 183 labeled_statement -> IDENTIFIER SIGNIFY statement
Rule 184 labeled_statement -> CASE constant_expression SIGNIFY statement
Rule 185 labeled_statement -> DEFAULT SIGNIFY statement
Rule 186 compound_statement -> LPAREN RPAPREN
Rule 187 compound_statement -> LPAREN statement_list RPAPREN
Rule 188 compound_statement -> LPAREN declaration_list RPAPREN
Rule 189 compound_statement -> LPAREN declaration_list statement_list RPAPREN
Rule 190 statement_list -> statement
Rule 191 statement_list -> statement_list statement
Rule 192 expression_statement -> SEMICOLON
Rule 193 expression_statement -> expression SEMICOLON
Rule 194 selection_statement -> IF LEFTBRACE expression RIGHTBRACE statement
Rule 195 selection_statement -> IF LEFTBRACE expression RIGHTBRACE statement ELSE statement
Rule 196 selection_statement -> SWITCH LEFTBRACE expression RIGHTBRACE statement
Rule 197 iteration_statement -> WHILE LEFTBRACE expression RIGHTBRACE statement
Rule 198 iteration_statement -> DO statement WHILE LEFTBRACE expression RIGHTBRACE SEMICOLON
Rule 199 iteration_statement -> FOR LEFTBRACE expression_statement expression_statement RIGHTBRACE statement
Rule 200 iteration_statement -> FOR LEFTBRACE expression_statement expression_statement expression RIGHTBRACE statement

Rule 201 jump_statement -> GOTO IDENTIFIER SEMICOLON
Rule 202 jump_statement -> CONTINUE SEMICOLON
Rule 203 jump_statement -> BREAK SEMICOLON
Rule 204 jump_statement -> RETURN SEMICOLON
Rule 205 jump_statement -> RETURN expression SEMICOLON
Rule 206 jump_statement -> RETURN CONSTANT SEMICOLON
Rule 207 jump_statement -> RETURN IDENTIFIER SEMICOLON

Procedure to run the code:

Given that, all the required libraries are fulfilled (bison,
Step1 : Copy the code to “nlp_group10_test.c”
Step2 : Open terminal in that particular folder
Step3: Enter “make clean” command
Step4: Enter “make” command
Step5: Parsing rules followed will be shown in “nlp_group10.txt”

```
pranshul@LAPTOP-S2831VFS:/mnt/c/Users/Pranshul/Pictures/abcd$ ma
ke
bison -dty nlp_group10.y --report=all
flex nlp_group10.l
gcc -W -c lex.yy.c
gcc -W -c nlp_group10_2.c
gcc -W -c y.tab.c
gcc lex.yy.o nlp_group10_2.o y.tab.o -lfl
./a.out < nlp_group10_test.c > nlp_group10.txt
pranshul@LAPTOP-S2831VFS:/mnt/c/Users/Pranshul/Pictures/abcd$ ma
ke clean
rm a.out lex.yy.* y.* nlp_group10_2.o
pranshul@LAPTOP-S2831VFS:/mnt/c/Users/Pranshul/Pictures/abcd$
```

Output for a basic hello world program

Input Code :

```
/*
This is comment 1
This is comment 2
This is comment 3
*/

// test: some specific keywords
//#include <stdio.h>
int main(){
   printf("HELLO WORLD\n");
}
```

```
******************* Multi Line Comment Starts at Line = 1 *******************
-------------------- Comment at Line = 2 -----------------
-------------------- Comment at Line = 3 -----------------
-------------------- Comment at Line = 4 -----------------
-------------------- Comment at Line = 5 -----------------
******************* Multi Line Comment Ends at Line = 5 *******************
#################### LINE NO : 6 ####################
#################### LINE NO : 7 ####################
| Rule: type_specifier => int |
| Rule: declaration_specifiers => type_specifier declaration_specifiers_opt |
| Rule: direct_declarator => IDENTIFIER |
| Rule: direct_declarator => direct_declarator ( identifier_list_opt ) |
| Rule: declarator => pointer_opt direct_declarator |
#################### LINE NO : 10 ####################
| Rule: primary_expression => IDENTIFIER|
| Rule: postfix_expression => primary_expression |
| Rule: primary_expression => STRING LITERAL |
| Rule: postfix_expression => primary_expression |
| Rule: unary_expression => postfix_expression |
| Rule: cast_expression => unary_expression |
| Rule: multiplicative_expression => cast_expression |
| Rule: additive_expression => multiplicative_expression |
| Rule: shift_expression => additive_expression |
| Rule: relational_expression => shift_expression |
```

| Rule: equality_expression => relational_expression |
| Rule: AND_expression => equality_expression |
| Rule: exclusive_OR_expression => AND_expression |
| Rule: inclusive_OR_expression => exclusive_OR_expression |
| Rule: logical_AND_expression => inclusive_OR_expression |
| Rule: logical_OR_expression => logical_AND_expression |
| Rule: conditional_expression => logical_OR_expression |
| Rule: assignment_expression => conditional_expression |
| Rule: argument_expression_list => assignment_expression |
| Rule: postfix_expression => postfix_expression (OPTIONAL : argument expression list) |
| Rule: unary_expression => postfix_expression |
| Rule: cast_expression => unary_expression |
| Rule: multiplicative_expression => cast_expression |
| Rule: additive_expression => multiplicative_expression |
| Rule: shift_expression => additive_expression |
| Rule: relational_expression => shift_expression |
| Rule: equality_expression => relational_expression |
| Rule: AND_expression => equality_expression |
| Rule: exclusive_OR_expression => AND_expression |
| Rule: inclusive_OR_expression => exclusive_OR_expression |
| Rule: logical_AND_expression => inclusive_OR_expression |
| Rule: logical_OR_expression => logical_AND_expression |
| Rule: conditional_expression => logical_OR_expression |
| Rule: assignment_expression => conditional_expression |
| Rule: expression => assignment_expression |
| Rule: expression_statement => expression_opt ; |
| Rule: statement => expression_statement |
| Rule: block_item => statement |
| Rule: block_item_list => block_item |

################### LINE NO : 11 ###################
| Rule: compound_statement => { block_item_list_opt } |
| Rule: function_definition => declaration_specifiers declarator declaration_list_opt compound_statement |
| Rule: external_declaration => function_definition |
| Rule: translation_unit => external_declaration|

Syntactic Error Detection

Input Code :

```
/*
---> CS39003 : Assignment-4
---> Name: Pritkumar Godhani + Debanjan Saha
---> Roll: 19CS10048 + 19CS30014
*/

// test: some specific keywords

int main()[
   int a,b;
   printf("ERROR IN brackets after main()");
]
```

Output :

```
****************** Multi Line Comment Starts at Line = 1 ******************
-------------------- Comment at Line = 2 -----------------
-------------------- Comment at Line = 3 -----------------
-------------------- Comment at Line = 4 -----------------
-------------------- Comment at Line = 5 -----------------
****************** Multi Line Comment Ends at Line = 5 ******************
#################### LINE NO : 6 ####################
#################### LINE NO : 7 ####################
#################### LINE NO : 9 ####################
| Rule: type_specifier => int |
| Rule: declaration_specifiers => type_specifier declaration_specifiers_opt |
| Rule: direct_declarator => IDENTIFIER |
| Rule: direct_declarator => direct_declarator ( identifier_list_opt ) |
#################### LINE NO : 10 ####################
Detected Error: syntax error
```

APPENDIX

We have initially coded in python, but due to the complexness of the C-grammar and the ply library, for better optimisation and visualization, we have shifted to BISON

Implementation of Lex using ply.lex() is done in lexing.py and lexi

The code can be found in the attached zip file.

Due to a large number of screenshots of the examples C-code parser, we decided to keep the output of the parsing examples into .txt file of the corresponding C-code which would be available in the .zip file as well.

Contribution

Lexing - Vardhan, Pranshul
Parsing - Pranshul, Khushal, Vardhan
Mid-Evaluation Presentation - Khushal, Vardhan
Ideas and Implementation - Khushal, Vardhan, Pranshul
Report - Khushal, Pranshul, Vardhan

Group Members

Headed By-
Course Lecturer - Prof. Sudeshna Sarkar
TA - Prithwish Jana

Members -
S.S.S.Vardhan (19CH30018)
Pranshul Narang (19EC10051)
Katra Khushal Jagadish (19CH10018)