

# Project Title: Parsing C-Language programs

By: **Rahul Aditya** (18CS30032)

Guided by: **Prof. Sudeshna Sarkar**

Mentor: **Prithwish Jana**

## Problem Statement

Given a C-Language Program, verify whether the program is syntactically correct or not using a reduced and optimized grammar and output the production rules used to parse the given program. Following are the elaborated points to be done in the term project:

1. Design a grammar that can parse a C program and **display the production rules** applied at each reduction step.
2. Now given a C program, the code associated with the parser should output whether the given C program is **syntactically correct or not**.
3. While preparing the grammar, we must take care to avoid **reduce-conflicts**, which means that there is precisely one way to create a parse tree out of a given syntactically correct C program.
4. Production rules should be such that the designed grammar can handle as many complex features in the C language as possible. Some notable complex features that our grammar can handle:
  - a. Multi-dimensional arrays
  - b. Looping constructs like for-loop and while-loop
  - c. If-else constructs like if-else and switch-case
  - d. Basic pointer-arithmetic

## Fundamental Ideas

This section describes the fundamental ideas used in developing our grammar and explains how our grammar works.

Following are some noteworthy points:

1. We start with the start symbol '**translation\_unit.**' It is this **translation\_unit** that breaks into various kinds of '**declaration**' like global variables, assistive functions, and main function.
2. '**declaration**' further reduces to the type of variable or functions, the name of the variable or functions, and parameters and definitions in the case of functions.
3. In the case of functions, '**declaration**' also contains rules to '**statement.**' Quite naturally, '**statement**' includes production rules which account for the iteration statements, jump statements, etc.
4. '**expression**' is derived from '**statement**' wherever applicable. '**expression**' is the non-terminal symbol containing all forms of expressions in C-language, like assignment expression, logical expression, and many others.
5. So, the flow of our grammar is something like: '**translation\_unit**' -> '**declaration**' -> '**statement**' -> '**expression**'. Hence, one of us took the responsibility of one for '**declaration,**' '**statement**' and '**expression.**'

We have described the general structure of our grammar in terms of '**declaration,**' '**statement,**' and '**expression.**' Now, let's see each of these starting non-terminals in detail. Also, although we have stated these three major starting non-terminals as though they are entirely independent. It is true in general and for a basic understanding of our grammar. However, there is some interdependence between the non-terminals produced by these fundamental non-terminals. It will be

evident in the next section when we look at the production rules.

Following are the essential points about '**declaration**':

1. A **declaration** is a non-terminal symbol that reduces to **declaration\_specifiers** and **declarator\_list**.
2. **declaration\_specifiers** is the non-terminal symbol that is supposed to specify the type of variable or function. Hence, it ultimately reduces to **storage\_class\_specifier** and **type\_specifier**. **storage\_class\_specifier** reduces to **extern** or **static** while **type\_specifier** reduces to **void**, **int**, **float**, etc.
3. **declarator\_list** is the non-terminal symbol that is supposed to specify the name of the variable or function. In the case of functions, parameters and function definition follow **declarator\_list**. Hence, **declarator\_list** ultimately reduces to **identifier**, **parameter\_list**, and **statement**.
4. Hence, the **identifier** is ultimately reduced to some given name of the variable or function.
5. **parameter\_list** takes care of the parameter list. Hence, it reduces to **declaration\_specifiers** described above with additional production rules to account for multiple parameters.

Following are the essential points about '**statement**':

1. A **statement** is a non-terminal symbol that reduces to different statements that can exist in a C program. Hence, it reduces to different statements like **labelled\_statement**, **selection\_statement**, **iteration\_statement**, **jump\_statement**, etc.
2. **labelled\_statement** reduces to **identifier : statement**, or **case constant\_expression : statement**, or **default : statement**. (**case** and **default** are terminal symbols)
3. **selection\_statement** reduces to **if (expression) statement**, or

**if (expression) statement else statement, or switch (expression) statement.** (**if**, **else** and **switch** are terminal symbols)

4. **iteration\_statement** reduces to **while( expression) statement**, or **do statement while(expression);**, or **for (expression; expression; expression) statement**, or **for(declaration expression; expression) statement**. (**do**, **while**, and **for** are terminal symbols)
5. **jump\_statement** reduces to **goto identifier ;**, or **continue ;**, or **break;**, or **return expression;**

Following are the essential points about ‘**expression**’:

1. **expression** is the non-terminal symbol which reduces to different kinds of expressions in C like **assignment\_expression**, **conditional\_expression**, **cast\_expression**, **arithmetic\_expression**, **postfix\_expression**, **unary\_expression**, etc.
2. **assignment\_expression** reduces to **declaration\_specifiers identifier assignment\_operator expression**. **assignment\_operator** could be one of **=**, **\*=**, **/=**, **%=**, **+=**, **-=**, **<<=**, **>>=**, **&=**, **^=**, **|=**, etc.
3. **conditional\_expression** reduces to one or some combination of **logical\_OR\_expression**, **logical\_AND\_expression**, **relational\_expression**, **equality-expression**, etc.
4. **arithmetic\_expression** reduces to one or some combination of **multiplicative\_expression**, **additive\_expression**, **shift\_expression**, etc.
5. **postfix\_expression** reduces to one of **postfix\_expression.identifier**, **postfix\_expression -> identifier**, **postfix\_expression ++**, etc.
6. **cast\_expression** reduces to **unary\_expression** or **( type\_name ) cast\_expression**.

7. **unary\_expression** reduces to **++ unary\_expression**,  
**unary\_operator cast\_expression, sizeof unary\_expression**,  
etc. **unary\_operator** can be one of **&, \*, +, -, ~, !**.

## **The Grammar for Parsing C-Language**

Grammar is defined by a four tuple **<S, T, NT, P>**, where S = start symbol, T = set of terminal symbols, NT = set of non-terminal symbols, and P = set of production rules. The terminal symbols are in uppercase, whereas the non-terminal symbols are in lowercase.

**S** = translation\_unit

**T** = { VOID, CHAR, SHORT, INT, LONG, FLOAT, DOUBLE, SIGNED, UNSIGNED, CONST, AUTO, STATIC, EXTERN, INLINE, RETURN, CASE, IF, ELSE, DO, WHILE, FOR, SWITCH, DEFAULT, CONTINUE, BREAK, GOTO, RESTRICT, VOLATILE, SIZEOF, REGISTER, IDENTIFIER, INTEGER\_CONSTANT, FLOATING\_CONSTANT, CHARACTER\_CONSTANT, STRING\_LITERAL, Deref, INC, DEC, DOT, DOTS, ADD, SUB, MUL, DIV, NEG, NOT, MODULO, EQ, NEQ, AND, OR, LEFT\_PAREN, RIGHT\_PAREN, LEFT\_SQUARE\_BRACKET, RIGHT\_SQUARE\_BRACKET, LEFT\_BRACKET, RIGHT\_BRACKET, BITWISE\_AND, BITWISE\_XOR, BITWISE\_OR, BINARY\_AND\_EQUAL, BINARY\_XOR\_EQUAL, BINARY\_OR\_EQUAL, MULTIPLY\_EQUAL, DIVIDE\_EQUAL, MOD\_EQUAL, SHIFT\_LEFT\_EQUAL, SHIFT\_RIGHT\_EQUAL, PLUS\_EQUAL, MINUS\_EQUAL, SHIFT\_LEFT, SHIFT\_RIGHT, LESS, MORE, LESS\_EQUAL, MORE\_EQUAL, QUESTION, COLON, SEMICOLON, ASSIGN, COMMA }

**NT** = { translation\_unit, external\_declaration, declaration\_list\_opt, function\_definition, declaration\_list, declaration,

init\_declarator\_list\_opt, declaration\_specifiers\_opt,  
declaration\_specifiers, init\_declarator\_list, init\_declarator,  
storage\_class\_specifier, type\_specifier, specifier\_qualifier\_list\_opt,  
specifier\_qualifier\_list, type\_qualifier, function\_specifier,  
primary\_expression, const, argument\_expression\_list\_opt,  
postfix\_expression, argument\_expression\_list, unary\_expression,  
unary\_operator, cast\_expression, multiplicative\_expression,  
additive\_expression, shift\_expression, relational\_expression,  
equality\_expression, AND\_expression, exclusive\_OR\_expression,  
inclusive\_OR\_expression, logical\_AND\_expression,  
logical\_OR\_expression, conditional\_expression,  
assignment\_expression, assignment\_operator, expression,  
const\_expression, pointer\_opt, declarator, type\_qualifier\_list\_opt,  
assignment\_expression\_opt, direct\_declarator, pointer,  
type\_qualifier\_list, parameter\_type\_list, parameter\_list,  
parameter\_declaration, IDENTIFIER\_list, type\_name, initializer,  
designation\_opt, initializer\_list, designation, designator\_list,  
designator, statement, labelled\_statement, block\_item\_list\_opt,  
compound\_statement, block\_item\_list, block\_item, exp\_opt,  
exp\_statement, selection\_statement\_base, selection\_statement,  
iteration\_statement, jump\_statement }

Next, we state the production rules in separate lines for the sake of clarity. The following are the production rules, i.e., each line belongs to the set of production rules **P**:

1. translation\_unit -> external\_declaration
2. translation\_unit -> translation\_unit external\_declaration
3. external\_declaration -> function\_definition
4. external\_declaration -> declaration
5. declaration\_list\_opt -> declaration\_list
6. declaration\_list\_opt -> **epsilon**
7. function\_definition -> declaration\_specifiers declarator

declaration\_list\_opt compound\_statement

8. declaration\_list -> declaration

9. declaration\_list -> declaration\_list declaration

10. declaration -> declaration\_specifiers init\_declarator\_list\_opt

SEMICOLON

11. init\_declarator\_list\_opt -> init\_declarator\_list

12. init\_declarator\_list\_opt -> **epsilon**

13. declaration\_specifiers\_opt -> declaration\_specifiers

14. declaration\_specifiers\_opt -> **epsilon**

15. declaration\_specifiers -> storage\_class\_specifier

declaration\_specifiers\_opt

16. declaration\_specifiers -> type\_specifier declaration\_specifiers\_opt

17. declaration\_specifiers -> type\_qualifier declaration\_specifiers\_opt

18. declaration\_specifiers -> function\_specifier

declaration\_specifiers\_opt

19. init\_declarator\_list -> init\_declarator

20. init\_declarator\_list -> init\_declarator\_list COMMA init\_declarator

21. init\_declarator -> declarator

22. init\_declarator -> declarator ASSIGN initializer

23. storage\_class\_specifier -> EXTERN | STATIC | AUTO |

REGISTER

24. type\_specifier -> VOID | CHAR | SHORT | INT | LONG | FLOAT |

DOUBLE | SIGNED | UNSIGNED

25. specifier\_qualifier\_list\_opt -> specifier\_qualifier\_list | **epsilon**

26. specifier\_qualifier\_list -> type\_specifier specifier\_qualifier\_list\_opt

27. specifier\_qualifier\_list -> type\_qualifier specifier\_qualifier\_list\_opt

28. type\_qualifier -> CONST | RESTRICT | VOLATILE

29. function\_specifier -> INLINE

30. primary\_expression -> IDENTIFIER | const | STRING\_LITERAL |

LEFT\_PAREN expression RIGHT\_PAREN

31. const -> INTEGER\_CONSTANT | FLOATING\_CONSTANT |

CHARACTER\_CONSTANT

32. argument\_expression\_list\_opt -> argument\_expression\_list | **epsilon**

33. postfix\_expression -> primary\_expression | postfix\_expression  
DOT IDENTIFIER | postfix\_expression DEREf IDENTIFIER |  
postfix\_expression INC | postfix\_expression DEC

34. postfix\_expression -> postfix\_expression  
LEFT\_SQUARE\_BRACKET expression RIGHT\_SQUARE\_BRACKET

35. postfix\_expression -> postfix\_expression LEFT\_PAREN  
argument\_expression\_list\_opt RIGHT\_PAREN

36. postfix\_expression -> LEFT\_PAREN type\_name RIGHT\_PAREN  
LEFT\_BRACKET initializer\_list RIGHT\_BRACKET

37. postfix\_expression -> LEFT\_PAREN type\_name RIGHT\_PAREN  
LEFT\_BRACKET initializer\_list COMMA RIGHT\_BRACKET

38. argument\_expression\_list -> assignment\_expression

39. argument\_expression\_list -> argument\_expression\_list COMMA  
assignment\_expression

40. unary\_expression -> postfix\_expression | INC unary\_expression |  
DEC unary\_expression | unary\_operator cast\_expression | SIZEOF  
unary\_expression | SIZEOF LEFT\_PAREN type\_name  
RIGHT\_PAREN

41. unary\_operator -> BITWISE\_AND | MUL | ADD | SUB | NEG |  
NOT

42. cast\_expression -> unary\_expression | LEFT\_PAREN type\_name  
RIGHT\_PAREN cast\_expression

43. multiplicative\_expression -> cast\_expression |  
multiplicative\_expression MUL cast\_expression |  
multiplicative\_expression DIV cast\_expression |  
multiplicative\_expression MODULO cast\_expression

44. additive\_expression -> multiplicative\_expression |  
additive\_expression ADD multiplicative\_expression |  
additive\_expression SUB multiplicative\_expression

45. shift\_expression -> additive\_expression | shift\_expression



SHIFT\_LEFT additive\_expression | shift\_expression SHIFT\_RIGHT  
additive\_expression

46. relational\_expression -> shift\_expression | relational\_expression  
LESS shift\_expression | relational\_expression MORE shift\_expression  
| relational\_expression LESS\_EQUAL shift\_expression |  
relational\_expression MORE\_EQUAL shift\_expression

47. equality\_expression -> relational\_expression

48. equality\_expression -> equality\_expression EQ  
relational\_expression

49. equality\_expression -> equality\_expression NEQ  
relational\_expression

50. AND\_expression -> equality\_expression | AND\_expression  
BITWISE\_AND equality\_expression

51. exclusive\_OR\_expression -> AND\_expression |  
exclusive\_OR\_expression BITWISE\_XOR AND\_expression

52. inclusive\_OR\_expression -> exclusive\_OR\_expression |  
inclusive\_OR\_expression BITWISE\_OR exclusive\_OR\_expression

53. logical\_AND\_expression -> inclusive\_OR\_expression |  
logical\_AND\_expression AND inclusive\_OR\_expression

54. logical\_OR\_expression -> logical\_AND\_expression |  
logical\_OR\_expression OR logical\_AND\_expression

55. conditional\_expression -> logical\_OR\_expression |  
logical\_OR\_expression QUESTION expression COLON  
conditional\_expression

56. assignment\_expression -> conditional\_expression |  
unary\_expression assignment\_operator assignment\_expression

57. assignment\_operator -> ASSIGN | MULTIPLY\_EQUAL |  
DIVIDE\_EQUAL | MOD\_EQUAL | PLUS\_EQUAL | MINUS\_EQUAL |  
SHIFT\_LEFT\_EQUAL | SHIFT\_RIGHT\_EQUAL |  
BINARY\_AND\_EQUAL | BINARY\_XOR\_EQUAL |  
BINARY\_OR\_EQUAL

58. expression -> assignment\_expression | expression COMMA

assignment\_expression

59. const\_expression -> conditional\_expression

60. pointer\_opt -> pointer | **epsilon**

61. declarator -> pointer\_opt direct\_declarator

62. type\_qualifier\_list\_opt -> type\_qualifier\_list | **epsilon**

63. assignment\_expression\_opt -> assignment\_expression | **epsilon**

64. direct\_declarator -> IDENTIFIER | LEFT\_PAREN declarator  
RIGHT\_PAREN

65. direct\_declarator -> direct\_declarator LEFT\_SQUARE\_BRACKET  
type\_qualifier\_list\_opt assignment\_expression\_opt  
RIGHT\_SQUARE\_BRACKET

66. direct\_declarator -> direct\_declarator LEFT\_SQUARE\_BRACKET  
STATIC type\_qualifier\_list\_opt assignment\_expression  
RIGHT\_SQUARE\_BRACKET

67. direct\_declarator -> direct\_declarator LEFT\_SQUARE\_BRACKET  
type\_qualifier\_list\_opt MUL RIGHT\_SQUARE\_BRACKET

68. direct\_declarator -> direct\_declarator LEFT\_PAREN  
parameter\_type\_list RIGHT\_PAREN

69. direct\_declarator -> direct\_declarator LEFT\_PAREN  
IDENTIFIER\_list RIGHT\_PAREN | direct\_declarator LEFT\_PAREN  
RIGHT\_PAREN

70. pointer -> MUL type\_qualifier\_list\_opt | MUL type\_qualifier\_list\_opt  
pointer

71. type\_qualifier\_list -> type\_qualifier | type\_qualifier\_list  
type\_qualifier

72. parameter\_type\_list -> parameter\_list | parameter\_list COMMA  
DOTS

73. parameter\_list -> parameter\_declaration | parameter\_list COMMA  
parameter\_declaration

74. parameter\_declaration -> declaration\_specifiers declarator |  
declaration\_specifiers

75. IDENTIFIER\_list -> IDENTIFIER

76. IDENTIFIER\_list -> IDENTIFIER\_list COMMA IDENTIFIER  
77. type\_name -> specifier\_qualifier\_list  
78. initializer -> assignment\_expression | LEFT\_BRACKET  
initializer\_list RIGHT\_BRACKET | LEFT\_BRACKET initializer\_list  
COMMA RIGHT\_BRACKET  
79. designation\_opt -> designation | **epsilon**  
80. initializer\_list -> designation\_opt initializer | initializer\_list COMMA  
designation\_opt initializer  
81. designation -> designator\_list ASSIGN  
82. designator\_list -> designator | designator\_list designator  
83. designator -> LEFT\_SQUARE\_BRACKET const\_expression  
RIGHT\_SQUARE\_BRACKET | DOT IDENTIFIER  
84. statement -> labelled\_statement | compound\_statement |  
exp\_statement | selection\_statement | iteration\_statement |  
jump\_statement  
85. labelled\_statement -> IDENTIFIER COLON statement | CASE  
const\_expression COLON statement | DEFAULT COLON statement  
86. block\_item\_list\_opt -> block\_item\_list | **epsilon**  
87. compound\_statement -> LEFT\_BRACKET block\_item\_list\_opt  
RIGHT\_BRACKET  
88. block\_item\_list -> block\_item | block\_item\_list block\_item  
89. block\_item -> declaration | statement  
90. exp\_opt -> expression | **epsilon**  
91. selection\_statement\_base -> IF LEFT\_PAREN expression  
RIGHT\_PAREN statement  
92. selection\_statement -> selection\_statement\_base \_ THEN  
93. selection\_statement -> selection\_statement\_base ELSE statement  
| SWITCH LEFT\_PAREN expression RIGHT\_PAREN statement  
94. iteration\_statement -> WHILE LEFT\_PAREN expression  
RIGHT\_PAREN statement  
95. iteration\_statement -> DO statement WHILE LEFT\_PAREN  
expression RIGHT\_PAREN SEMICOLON

96. iteration\_statement -> FOR LEFT\_PAREN exp\_opt SEMICOLON  
exp\_opt SEMICOLON exp\_opt RIGHT\_PAREN statement  
97. iteration\_statement -> FOR LEFT\_PAREN declaration exp\_opt  
SEMICOLON exp\_opt RIGHT\_PAREN statement  
98. jump\_statement -> GOTO IDENTIFIER SEMICOLON |  
CONTINUE SEMICOLON | BREAK SEMICOLON | RETURN exp\_opt  
SEMICOLON

## **README**

Steps to check the syntactic correctness of a C-program, say  
“check.c”, are as follows:

1. Open the terminal in the NLP folder of the zipped file, and paste the “check.c” file in it.
2. Type “make” in the terminal.
3. There are two alternate ways from here on:
  - a. Using Makefile: Open the “Makefile” in the folder and edit the file in two places. The first place is the “output.txt: a.out” label. Add “./a.out < check.c > out.txt” if you want to see the output in a file named out.txt. Also, add “out.txt” in the “clean” label to remove the file “out.txt” once “make clean” is called.
  - b. Without using Makefile: Type the command “./a.out < check.c” to view the output in the terminal.

## **List of syntactically correct C-programs and outputs**

The codes and corresponding output screenshots for **syntactically correct** C-programs have been given. The screenshots of C-programs provided here are for test1.c, test2.c, ..., test10.c in the **NLP** folder. To view, the entire corresponding outputs, type “make” in the terminal and refer to output1.txt, output2.txt, ..., output10.txt. These output files

contain the production rules applied at each line of the file.

## 1. The sample input as given by Mentor

Code:

```
1  #include <stdio.h>
2  #define myGlobalInt 5
3
4  int main()
5  {
6      int myLocalInt = 7;
7      int myProd = myLocalInt * myGlobalInt;
8      printf("My Calculator => %d * %d = %04d \n", myGlobalInt, myLocalInt, myProd);
9      return 0;
10 }
```

Output:

```
200 ##### LINE NO : 10 #####
201 block_item_list_opt -> block_item_list
202 compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
203 function_definition -> declaration_specifiers declarator declaration_list_opt
   compound_statement
204 external_declaration -> function_definition
205 translation_unit -> external_declaration
206
207 Parsing Successful
208
```

## 2. Most basic C program

Code:

```

1  | #include <stdio.h>
2
3  int main() {
4      int input1 = 10;
5      int input2;
6      scanf("%d", &input2);
7
8      printf("%d %d\n", input1, input2);
9
10     return 0;
11 }

```

Output:

```

197 ##### LINE NO : 9 #####
198
199 ##### LINE NO : 10 #####
200 const -> INTEGER_CONSTANT
201 primary_expression -> const
202 postfix_expression -> primary_expression
203 unary_expression -> postfix_expression
204 cast_expression -> unary_expression
205 multiplicative_expression -> cast_expression
206 additive_expression -> multiplicative_expression
207 shift_expression -> additive_expression
208 relational_expression -> shift_expression
209 equality_expression -> relational_expression
210 AND_expression -> equality_expression
211 exclusive_OR_expression -> AND_expression
212 inclusive_OR_expression -> exclusive_OR_expression
213 logical_AND_expression -> inclusive_OR_expression
214 logical_OR_expression -> logical_AND_expression
215 conditional_expression -> logical_OR_expression
216 assignment_expression -> conditional_expression
217 expression -> assignment_expression
218 exp_opt -> expression
219 jump_statement -> RETURN exp_opt SEMICOLON
220 statement -> jump_statement
221 block_item -> statement
222 block_item_list -> block_item_list block_item
223
224 ##### LINE NO : 11 #####
225 block_item_list_opt -> block_item_list
226 compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
227 function_definition -> declaration_specifiers declarator declaration_list_opt
   compound_statement
228 external_declaration -> function_definition
229 translation_unit -> external_declaration
230
231 Parsing Successful
232

```

### 3. Testing various operations

Code:

```

1  #include <stdio.h>
2
3  int main() {
4      int num = 0;
5      num++;
6      num--;
7      num = num * num;
8      num = num - 12;
9      num = num % 13;
10     num = ((num >> 8) << 5) + 19) - 1100;
11     num = 81 ^ 50;
12     num <=<= 5;
13     num >=>= 2;
14     num &= 21;
15     num ^= 122;
16
17     return 0;
18 }

```

Output:

```

470 ##### LINE NO : 16 #####
471
472 ##### LINE NO : 17 #####
473 const -> INTEGER_CONSTANT
474 primary_expression -> const
475 postfix_expression -> primary_expression
476 unary_expression -> postfix_expression
477 cast_expression -> unary_expression
478 multiplicative_expression -> cast_expression
479 additive_expression -> multiplicative_expression
480 shift_expression -> additive_expression
481 relational_expression -> shift_expression
482 equality_expression -> relational_expression
483 AND_expression -> equality_expression
484 exclusive_OR_expression -> AND_expression
485 inclusive_OR_expression -> exclusive_OR_expression
486 logical_AND_expression -> inclusive_OR_expression
487 logical_OR_expression -> logical_AND_expression
488 conditional_expression -> logical_OR_expression
489 assignment_expression -> conditional_expression
490 expression -> assignment_expression
491 exp_opt -> expression
492 jump_statement -> RETURN exp_opt SEMICOLON
493 statement -> jump_statement
494 block_item -> statement
495 block_item_list -> block_item_list block_item
496
497 ##### LINE NO : 18 #####
498 block_item_list_opt -> block_item_list
499 compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
500 function_definition -> declaration_specifiers declarator declaration_list_opt
    compound_statement
501 external_declaration -> function_definition
502 translation_unit -> external_declaration
503
504 Parsing Successful
505

```

## 4. Testing data types' declarations and initializations

Code:

```

1  #include <stdio.h>
2
3  int main() {
4      int input;
5      input = 20;
6
7      auto float check1 = -45.56;
8      volatile int check2 = 89;
9
10     float f = .87e-03;
11     double d = 25.;
12     float t = 43.46;
13     const char ch = 'a';
14
15     char buff[30] = "Here is a good one!\n\t";
16     char arr[4] = "";
17     int int_array[] = {10, 7, 8, 9, 1, 5};
18
19     short int A = (5 ^ 7) | 9;
20     signed int B = -A;
21     unsigned long C = A + B;
22
23     return 0;
24 }

```

Output:

```

581 ##### LINE NO : 22 #####
582 const -> INTEGER CONSTANT
583 primary_expression -> const
584 postfix_expression -> primary_expression
585 unary_expression -> postfix_expression
586 cast_expression -> unary_expression
587 multiplicative_expression -> cast_expression
588 additive_expression -> multiplicative_expression
589 shift_expression -> additive_expression
590 relational_expression -> shift_expression
591 equality_expression -> relational_expression
592 AND_expression -> equality_expression
593 exclusive_OR_expression -> AND_expression
594 inclusive_OR_expression -> exclusive_OR_expression
595 logical_AND_expression -> inclusive_OR_expression
596 logical_OR_expression -> logical_AND_expression
597 conditional_expression -> logical_OR_expression
598 assignment_expression -> conditional_expression
599 expression -> assignment_expression
600 exp_opt -> expression
601 jump_statement -> RETURN exp_opt SEMICOLON
602 statement -> jump_statement
603 block_item -> statement
604 block_item_list -> block_item_list block_item
605
606 ##### LINE NO : 23 #####
607 block_item_list_opt -> block_item_list
608 compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
609 function_definition -> declaration_specifiers declarator declaration_list_opt
610 compound_statement
611 external_declaration -> function_definition
612 translation_unit -> external_declaration
613 Parsing Successful
614

```

## 5. Checking basic for-loop and function call

Code:



```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define MXN 1000
5
6  int dp[MXN];
7
8  void Fibo(int n) {
9      dp[0] = 0;
10     dp[1] = 1;
11     for(int i = 2; i <= n; i++) {
12         dp[i] = dp[i - 1] + dp[i - 2];
13     }
14 }
15
16 int main() {
17     int n = 50;
18
19     Fibo(n);
20
21     return 0;
22 }

```

Output:

```

434 ##### LINE NO : 21 #####
435 const -> INTEGER_CONSTANT
436 primary_expression -> const
437 postfix_expression -> primary_expression
438 unary_expression -> postfix_expression
439 cast_expression -> unary_expression
440 multiplicative_expression -> cast_expression
441 additive_expression -> multiplicative_expression
442 shift_expression -> additive_expression
443 relational_expression -> shift_expression
444 equality_expression -> relational_expression
445 AND_expression -> equality_expression
446 exclusive_OR_expression -> AND_expression
447 inclusive_OR_expression -> exclusive_OR_expression
448 logical_AND_expression -> inclusive_OR_expression
449 logical_OR_expression -> logical_AND_expression
450 conditional_expression -> logical_OR_expression
451 assignment_expression -> conditional_expression
452 expression -> assignment_expression
453 exp_opt -> expression
454 jump_statement -> RETURN exp_opt SEMICOLON
455 statement -> jump_statement
456 block_item -> statement
457 block_item_list -> block_item_list block_item
458
459 ##### LINE NO : 22 #####
460 block_item_list_opt -> block_item_list
461 compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
462 function_definition -> declaration_specifiers declarator declaration_list_opt
463 compound_statement
464 external_declaration -> function_definition
465 translation_unit -> translation_unit external_declaration
466 Parsing Successful
467

```

## 6. Checking the following:

- a. declaration of 'extern' and 'static' functions

- b. nested if-else statements
- c. return statements
- d. some other auxiliary operations

Code:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  extern double func();
5
6  const int INVALID = -1;
7
8  static int demo(int x) {
9      if(x == 0) {
10         return 0;
11     }
12     else if(x % 4 == 0) {
13         if(x % 2 == 0) {
14             return x / 2;
15         }
16         else return x + 4;
17     }
18     else if(x % 3 == 0) {
19         return (x - 1) * (x - 1);
20     }
21     else {
22         return (x + 1) * (x + 1);
23     }
24     return INVALID;
25 }
26
27
28 int main() {
29     int N = 100;
30     // This is the comment to check parsing
31     int ans = demo(N);
32     ans += N;
33
34     /* This is another type of comment to check parsing */
35     printf("%d\n", ans);
36
37     return 0;
38 }
```

Output:

```
736 ##### LINE NO : 38 #####
737 block_item_list_opt -> block_item_list
738 compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
739 function_definition -> declaration_specifiers declarator declaration_list_opt compound_statement
740 external_declaration -> function_definition
741 translation_unit -> translation_unit external_declaration
742
743 Parsing Successful
744
```

7. Checking the following:
- a. Pointer arithmetic via swap function

- b. Reference passing to functions
- c. Passing arrays and 32-bit integers together as parameters
- d. Casting a variable

Code:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void swap(int *a, int *b) {
5      int temp = *a;
6      *a = *b;
7      *b = temp;
8  }
9
10 void print_the_array(int A[], int n) {
11     for(int i = 0; i < n; i++) {
12         A[i] += 10;
13         printf("%d ", A[i]);
14     }
15     printf("\n");
16 }
17
18 int main() {
19     int A[] = {10, 7, 8, 9, 1, 5};
20     int n = (int) sizeof(A) / (int) sizeof(A[0]);
21     printf("Size of the array = %d\n", n);
22
23     swap(&A[0], &A[n - 1]);
24     print_the_array(A, n);
25
26     return 0;
27 }
```

Output:

```
912 ##### LINE NO : 27 #####
913 block_item_list_opt -> block_item_list
914 compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
915 function_definition -> declaration_specifiers declarator declaration_list_opt compound_statement
916 external_declaration -> function_definition
917 translation_unit -> translation_unit external_declaration
918
919 Parsing Successful
```

- 8. Checking the following:
  - a. switch-case statements

- b. keywords like continue, break from the for-loop, and default
- c. nested if-else with some operations
- d. multiple conditions using logical operators in if-else clauses

Code:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define NONE -1
6
7  int main() {
8      int i, ans = 0;
9      for(int i = 0; i < 100; i++) {
10         switch(i) {
11             case(50) : continue;
12             case(60) : break;
13             default : ans++;
14         }
15     }
16
17     if(ans >= 50 && ((ans <= 75) || (ans % 10 == 0))) {
18         ans--;
19     }
20     else {
21         ans++;
22     }
23
24     if(ans < -100 || ans > 100) {
25         ans = NONE;
26     }
27     else if(ans % 2 == 0) {
28         ans /= 2;
29     }
30     else {
31         if(ans % 3 == 0 && ans <= 69) {
32             ans /= 3;
33         }
34         else {
35             ans = (ans << 2);
36         }
37     }
38
39     printf("%d\n", ans);
40
41
42     return 0;
43 }

```

Output:

```

901 ##### LINE NO : 43 #####
902 block_item list opt -> block item list
903 compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
904 function_definition -> declaration_specifiers declarator declaration_list_opt compound_statement
905 external_declaration -> function_definition
906 translation_unit -> external_declaration
907
908 Parsing Successful

```

## 9. Checking the following:

- a. while and do-while loops
- b. if-else-if statements without brackets (1 line)
- c. break keyword use in while loop

Code:

```
1  #include <stdio.h>
2
3  int main() {
4      int n = 0;
5
6      do {
7          if(n % 2 == 0)
8              n++;
9          else if(n % 3 == 0)
10             n += 2;
11         else
12             n += 4;
13     } while(n < 100);
14
15     while(1) {
16         if(n == 0) {
17             break;
18         }
19
20         if(n % 10 == 0)
21             n /= 10;
22         else if(n % 3 == 0)
23             n /= 3;
24         else
25             n /= 2;
26     }
27
28     printf("%d\n", n);
29
30     return 0;
31 }
32 }
```

Output:

```
574 ##### LINE NO : 32 #####
575 block_item_list_opt -> block_item_list
576 compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
577 function_definition -> declaration_specifiers declarator declaration_list_opt compound_statement
578 external_declaration -> function_definition
579 translation_unit -> external_declaration
580
581 Parsing Successful
```

## 10. Solving the famous knapsack problem using Dynamic Programming, which checks multiple rules at once like:

- a. Multi-dimensional arrays (Used 2-d arrays, but grammar can handle any number of dimensions)
- b. Nested for loops
- c. Function calls in any arbitrary position
- d. If else statements with logical operators
- e. Casting operation
- f. Using '?' and ':' to state a conditional statement

Code:

```
1  #include <stdio.h>
2
3  const int INF = (int) 1e9;
4
5  int max(int a, int b) {
6      int ans;
7      if(a > b) {
8          ans = a;
9      }
10     else {
11         ans = b;
12     }
13     return ans;
14 }
15
16 int main() {
17     int value[100];
18     int weight[100];
19     int n;
20     int W;
21     printf("Enter the number of items and max. weight of Knapsack: ");
22     scanf("%d %d", &n, &W);
23
24     printf("Enter the value of items\n");
25     for(int i = 0; i < n; i++) {
26         scanf("%d", &value[i]);
27     }
28     printf("Enter the weight of items\n");
29     for(int i = 0; i < n; i++) {
30         scanf("%d", &weight[i]);
31     }
32     int dp[100][100];
33     for(int i = 0; i <= n; i++) {
34         for(int j = 0; j <= W; j++) {
35             if(i == 0 || j == 0) {
36                 dp[i][j] = 0;
37             }
38             else {
39                 int choice1 = dp[i - 1][j];
40                 int choice2 = (j >= weight[i] ? (dp[i - 1][j - weight[i]] + value[i]) : -INF);
41                 dp[i][j] = max(choice1, choice2);
42             }
43         }
44     }
45
46     printf("%d", dp[n][W]);
47
48     return 0;
49 }
```

Output:

```
1668 exclusive_OR_expression -> AND_expression
1669 inclusive_OR_expression -> exclusive_OR_expression
1670 logical_AND_expression -> inclusive_OR_expression
1671 logical_OR_expression -> logical_AND_expression
1672 conditional_expression -> logical_OR_expression
1673 assignment_expression -> conditional_expression
1674 expression -> assignment_expression
1675 exp_opt -> expression
1676 exp_statement -> exp_opt SEMICOLON
1677 statement -> exp_statement
1678 block_item -> statement
1679 block_item_list -> block_item_list block_item
1680
1681 ##### LINE NO : 47 #####
1682
1683 ##### LINE NO : 48 #####
1684 const -> INTEGER_CONSTANT
1685 primary_expression -> const
1686 postfix_expression -> primary_expression
1687 unary_expression -> postfix_expression
1688 cast_expression -> unary_expression
1689 multiplicative_expression -> cast_expression
1690 additive_expression -> multiplicative_expression
1691 shift_expression -> additive_expression
1692 relational_expression -> shift_expression
1693 equality_expression -> relational_expression
1694 AND_expression -> equality_expression
1695 exclusive_OR_expression -> AND_expression
1696 inclusive_OR_expression -> exclusive_OR_expression
1697 logical_AND_expression -> inclusive_OR_expression
1698 logical_OR_expression -> logical_AND_expression
1699 conditional_expression -> logical_OR_expression
1700 assignment_expression -> conditional_expression
1701 expression -> assignment_expression
1702 exp_opt -> expression
1703 jump_statement -> RETURN exp_opt SEMICOLON
1704 statement -> jump_statement
1705 block_item -> statement
1706 block_item_list -> block_item_list block_item
1707
1708 ##### LINE NO : 49 #####
1709 block_item_list_opt -> block_item_list
1710 compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
1711 function_definition -> declaration_specifiers declarator declaration_list_opt compound_statement
1712 external_declaration -> function_definition
1713 translation_unit -> translation_unit external_declaration
1714
1715 Parsing Successful
```

## List of syntactically incorrect C-programs and outputs

The codes and corresponding output screenshots for **syntactically incorrect** C-programs have been given. The screenshots of C-programs provided here are for incorrect1.c, incorrect2.c, ..., incorrect10.c in the **NLP** folder. To view entire corresponding outputs, type “make” in the terminal and refer to incorrect\_out1.txt, incorrect\_out2.txt, ..., incorrect\_out10.txt. These output files contain the production rules applied at each line of the file.

## 1. Code:

```
1  #include <stdio.h>
2  #define myGlobalInt 5
3
4  int main()
5  {
6      int myLocalInt = 7;
7      int myProd = myLocalInt myGlobalInt;
8      printf("My Calculator => %d * %d = %04d \n", myGlobalInt, myLocalInt, myProd);
9      return 0;
10 }
```

## Output:

```
47 ##### LINE NO : 7 #####
48 type_specifier -> INT
49 declaration_specifiers -> type_specifier declaration_specifiers_opt
50 direct_declarator -> IDENTIFIER
51 declarator -> pointer_opt direct_declarator
52 primary_expression -> IDENTIFIER
53 postfix_expression -> primary_expression
54 unary_expression -> postfix_expression
55 cast_expression -> unary_expression
56 multiplicative_expression -> cast_expression
57 additive_expression -> multiplicative_expression
58 shift_expression -> additive_expression
59 relational_expression -> shift_expression
60 equality_expression -> relational_expression
61 AND_expression -> equality_expression
62 exclusive_OR_expression -> AND_expression
63 inclusive_OR_expression -> exclusive_OR_expression
64 logical_AND_expression -> inclusive_OR_expression
65 logical_OR_expression -> logical_AND_expression
66 conditional_expression -> logical_OR_expression
67 assignment_expression -> conditional_expression
68 initializer -> assignment_expression
69 init_declarator -> declarator ASSIGN initializer
70 init_declarator_list -> init_declarator
71 init_declarator_list_opt -> init_declarator_list
72 syntax error
73
74 Parsing Unsuccessful
75
```



## 2. Code:

```
1  #include<stdio.h>
2
3  int main()
4  {
5      int a = 1, b = 2;
6      if(a > b)
7          a += 2;
8
9      b += 2;
10
11     else
12         b += 2;
13
14     return 0;
15 }
```

## Output:

```
123 ##### LINE NO : 8 #####
124
125 ##### LINE NO : 9 #####
126 selection_statement -> selection_statement_base _ THEN
127 statement -> selection_statement
128 block_item -> statement
129 block_item_list -> block_item_list block_item
130 primary_expression -> IDENTIFIER
131 postfix_expression -> primary_expression
132 unary_expression -> postfix_expression
133 assignment_operator -> PLUS_EQUAL
134 const -> INTEGER CONSTANT
135 primary_expression -> const
136 postfix_expression -> primary_expression
137 unary_expression -> postfix_expression
138 cast_expression -> unary_expression
139 multiplicative_expression -> cast_expression
140 additive_expression -> multiplicative_expression
141 shift_expression -> additive_expression
142 relational_expression -> shift_expression
143 equality_expression -> relational_expression
144 AND_expression -> equality_expression
145 exclusive_OR_expression -> AND_expression
146 inclusive_OR_expression -> exclusive_OR_expression
147 logical_AND_expression -> inclusive_OR_expression
148 logical_OR_expression -> logical_AND_expression
149 conditional_expression -> logical_OR_expression
150 assignment_expression -> conditional_expression
151 assignment_expression -> unary_expression assignment_operator assignment_expression
152 expression -> assignment_expression
153 exp_opt -> expression
154 exp_statement -> exp_opt SEMICOLON
155 statement -> exp_statement
156 block_item -> statement
157 block_item_list -> block_item_list block_item
158
159 ##### LINE NO : 10 #####
160
161 ##### LINE NO : 11 #####
162 block_item_list_opt -> block_item_list
163 syntax error
164
165 Parsing Unsuccessful
166
```

### 3. Code:

```
1  #include<stdio.h>
2
3  int main()
4  {
5      char str[] = "Hello World!";
6      str[5] = 't';
7      return 0;
8  }
```

### Output:

```
45 ##### LINE NO : 6 #####
46 primary_expression -> IDENTIFIER
47 postfix_expression -> primary_expression
48 const -> INTEGER_CONSTANT
49 primary_expression -> const
50 postfix_expression -> primary_expression
51 unary_expression -> postfix_expression
52 cast_expression -> unary_expression
53 multiplicative_expression -> cast_expression
54 additive_expression -> multiplicative_expression
55 shift_expression -> additive_expression
56 relational_expression -> shift_expression
57 equality_expression -> relational_expression
58 AND_expression -> equality_expression
59 exclusive_OR_expression -> AND_expression
60 inclusive_OR_expression -> exclusive_OR_expression
61 logical_AND_expression -> inclusive_OR_expression
62 logical_OR_expression -> logical_AND_expression
63 conditional_expression -> logical_OR_expression
64 assignment_expression -> conditional_expression
65 expression -> assignment_expression
66 syntax error
67
68 Parsing Unsuccessful
69
```

#### 4. Code:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a[5] = {1, 2, 3, 4, 5};
6      int i = 0;
7      for(i = 0; i < 5; i++; i++) {
8          a[i] += 1;
9      }
10
11     return 0;
12 }
```

#### Output:

```
207 postfix_expression -> primary_expression
208 unary_expression -> postfix_expression
209 cast_expression -> unary_expression
210 multiplicative_expression -> cast_expression
211 additive_expression -> multiplicative_expression
212 shift_expression -> additive_expression
213 relational_expression -> relational_expression LESS shift_expression
214 equality_expression -> relational_expression
215 AND_expression -> equality_expression
216 exclusive_OR_expression -> AND_expression
217 inclusive_OR_expression -> exclusive_OR_expression
218 logical_AND_expression -> inclusive_OR_expression
219 logical_OR_expression -> logical_AND_expression
220 conditional_expression -> logical_OR_expression
221 assignment_expression -> conditional_expression
222 expression -> assignment_expression
223 exp_opt -> expression
224 primary_expression -> IDENTIFIER
225 postfix_expression -> primary_expression
226 postfix_expression -> postfix_expression INC
227 unary_expression -> postfix_expression
228 cast_expression -> unary_expression
229 multiplicative_expression -> cast_expression
230 additive_expression -> multiplicative_expression
231 shift_expression -> additive_expression
232 relational_expression -> shift_expression
233 equality_expression -> relational_expression
234 AND_expression -> equality_expression
235 exclusive_OR_expression -> AND_expression
236 inclusive_OR_expression -> exclusive_OR_expression
237 logical_AND_expression -> inclusive_OR_expression
238 logical_OR_expression -> logical_AND_expression
239 conditional_expression -> logical_OR_expression
240 assignment_expression -> conditional_expression
241 expression -> assignment_expression
242 exp_opt -> expression
243 syntax error
244
245 Parsing Unsuccessful
```

## 5. Code:

```
1  #include <stdio.h>
2
3  int fib(int a, int b)
4  {
5      if(a <= 1)
6          return a;
7      return fib(a - 1, b) + fib(a - 2, b - 1);
8  }
9
10 int main()
11 {
12     int n = 9;
13     int sum = 0;
14     for(int i = 1; i <= n; i++) {
15         sum += fib(i, 0);
16     }
17     printf("%d\n", sum);
18     return 0;
19 }
20 }
```

## Output:

```
484
485 ##### LINE NO : 18 #####
486
487 ##### LINE NO : 19 #####
488 const -> INTEGER CONSTANT
489 primary_expression -> const
490 postfix_expression -> primary_expression
491 unary_expression -> postfix_expression
492 cast_expression -> unary_expression
493 multiplicative_expression -> cast_expression
494 additive_expression -> multiplicative_expression
495 shift_expression -> additive_expression
496 relational_expression -> shift_expression
497 equality_expression -> relational_expression
498 AND_expression -> equality_expression
499 exclusive_OR_expression -> AND_expression
500 inclusive_OR_expression -> exclusive_OR_expression
501 logical_AND_expression -> inclusive_OR_expression
502 logical_OR_expression -> logical_AND_expression
503 conditional_expression -> logical_OR_expression
504 assignment_expression -> conditional_expression
505 expression -> assignment_expression
506 exp_opt -> expression
507 jump_statement -> RETURN exp_opt SEMICOLON
508 statement -> jump_statement
509 block_item -> statement
510 block_item_list -> block_item_list block_item
511
512 ##### LINE NO : 20 #####
513 block_item_list_opt -> block_item_list
514 compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
515 statement -> compound_statement
516 iteration_statement -> FOR LEFT_PAREN declaration exp_opt SEMICOLON exp_opt RIGHT_PAREN statement
517 statement -> iteration_statement
518 block_item -> statement
519 block_item_list -> block_item_list block_item
520 block_item_list_opt -> block_item_list
521 syntax error
522
523 Parsing Unsuccessful
```

## Some C-programs which our grammar cannot handle

1. C-programs having struct/union. It should be syntactically correct but our grammar output is syntactically incorrect.

Code:

```
1  #include <stdio.h>
2
3  struct coord
4  {
5      int x, y;
6  };
7
8  int main() {
9      struct coord A;
10     A.x = 10;
11     A.y = 20;
12     printf("%d %d\n", A.x, A.y);
13
14     return 0;
15 }
```

Output:

```
1  Parsing begins
2
3
4  ##### LINE NO : 2 #####
5
6  ##### LINE NO : 3 #####
7  syntax error
8
9  Parsing Unsuccessful
10
```

2. This special edge scenario is in a switch-case. Here, the correct output is syntactically incorrect but our code outputs are syntactically correct. The **case** label should exist within the **switch** statement only, but our grammar doesn't account for that. However, our grammar can handle other complicated **switch-case** statements.

Code:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6
7      int choice = 2;
8      int temp = 1;
9      switch(choice);
10     {
11         case 1:
12             temp = 2;
13             break;
14         case 2:
15             temp = 3;
16             break;
17         case 3:
18             temp = 4;
19             break;
20         case 4:
21             temp = 5;
22             break;
23         default:
24             temp = 6;
25     }
26
27     return 0;
28 }
```

Output:

```
402 ##### LINE NO : 28 #####
403 block_item_list_opt -> block_item_list
404 compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
405 function_definition -> declaration_specifiers declarator declaration_list_opt compound_statement
406 external_declaration -> function_definition
407 translation_unit -> external_declaration
408
409 Parsing Successful
410
```

## **Bibliographic resources:**

**Compilers: Principles, Techniques, and Tools** by Alfred V. Aho,  
Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman