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_specifer** 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:

```c
#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;
}
```

Output:

```
#################### LINE NO : 10 ####################
block_item_list_opt -> block_item_list
compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
function_definition -> declaration_specifiers declarator declaration_list_opt
compound_statement
external_declaration -> function_definition
translation_unit -> external_declaration

Parsing Successful

```

2. Most basic C program

Code:

```c
#include <stdio.h>

int main() {
    int input1 = 10;
    int input2;
    scanf("%d", &input2);

    printf("%d %d\n", input1, input2);

    return 0;
}
```

Output:

```
#################### LINE NO : 9 ####################

#################### LINE NO : 10 ####################
const -> INTEGER_CONSTANT
primary_expression -> const
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
AND_expression -> equality_expression
exclusive_OR_expression -> AND_expression
inclusive_OR_expression -> exclusive_OR_expression
logical_AND_expression -> inclusive_OR_expression
logical_OR_expression -> logical_AND_expression
conditional_expression -> logical_OR_expression
assignment_expression -> conditional_expression
expression -> assignment_expression
exp_opt -> expression
jump_statement -> RETURN exp_opt SEMICOLON
statement -> jump_statement
block_item -> statement
block_item_list -> block_item_list block_item

#################### LINE NO : 11 ####################
block_item_list_opt -> block_item_list
compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
function_definition -> declaration_specifiers declarator declaration_list_opt
compound_statement
external_declaration -> function_definition
translation_unit -> external_declaration

Parsing Successful

```

3. Testing various operations

Code:

```c
#include <stdio.h>

int main() {
    int num = 0;
    num++;
    num--;
    num = num * num;
    num = num - 12;
    num = num % 13;
    num = (((num >> 8) << 5) + 19) - 1100;
    num = 81 ^ 50;
    num <<= 5;
    num >>= 2;
    num &= 21;
    num ^= 122;

    return 0;
}
```

Output:

```
#################### LINE NO : 16 ####################

#################### LINE NO : 17 ####################
const -> INTEGER_CONSTANT
primary_expression -> const
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
AND_expression -> equality_expression
exclusive_OR_expression -> AND_expression
inclusive_OR_expression -> exclusive_OR_expression
logical_AND_expression -> inclusive_OR_expression
logical_OR_expression -> logical_AND_expression
conditional_expression -> logical_OR_expression
assignment_expression -> conditional_expression
expression -> assignment_expression
exp_opt -> expression
jump_statement -> RETURN exp_opt SEMICOLON
statement -> jump_statement
block_item -> statement
block_item_list -> block_item_list block_item

#################### LINE NO : 18 ####################
block_item_list_opt -> block_item_list
compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
function_definition -> declaration_specifiers declarator declaration_list_opt
compound_statement
external_declaration -> function_definition
translation_unit -> external_declaration

Parsing Successful

```

4. Testing data types' declarations and initializations

Code:

```c
#include <stdio.h>

int main() {
    int input;
    input = 20;

    auto float check1 = -45.56;
    volatile int check2 = 89;

    float f = .87e-03;
    double d = 25.;
    float t = 43.46;
    const char ch = 'a';

    char buff[30] = "Here is a good one!\n\t";
    char arr[4] = "";
    int int_array[] = {10, 7, 8, 9, 1, 5};

    short int A = (5 ^ 7) | 9;
    signed int B = -A;
    unsigned long C = A + B;

    return 0;
}
```

Output:

```
#################### LINE NO : 22 ####################
const -> INTEGER_CONSTANT
primary_expression -> const
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
AND_expression -> equality_expression
exclusive_OR_expression -> AND_expression
inclusive_OR_expression -> exclusive_OR_expression
logical_AND_expression -> inclusive_OR_expression
logical_OR_expression -> logical_AND_expression
conditional_expression -> logical_OR_expression
assignment_expression -> conditional_expression
expression -> assignment_expression
exp_opt -> expression
jump_statement -> RETURN exp_opt SEMICOLON
statement -> jump_statement
block_item -> statement
block_item_list -> block_item_list block_item

#################### LINE NO : 23 ####################
block_item_list_opt -> block_item_list
compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
function_definition -> declaration_specifiers declarator declaration_list_opt
compound_statement
external_declaration -> function_definition
translation_unit -> external_declaration

Parsing Successful

```

5. Checking basic for-loop and function call

Code:

```c
#include <stdio.h>
#include <stdlib.h>

#define MXN 1000

int dp[MXN];

void Fibo(int n) {
    dp[0] = 0;
    dp[1] = 1;
    for(int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
}

int main() {
    int n = 50;

    Fibo(n);

    return 0;
}
```

Output:

```
#################### LINE NO : 21 ####################
const -> INTEGER_CONSTANT
primary_expression -> const
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
AND_expression -> equality_expression
exclusive_OR_expression -> AND_expression
inclusive_OR_expression -> exclusive_OR_expression
logical_AND_expression -> inclusive_OR_expression
logical_OR_expression -> logical_AND_expression
conditional_expression -> logical_OR_expression
assignment_expression -> conditional_expression
expression -> assignment_expression
exp_opt -> expression
jump_statement -> RETURN exp_opt SEMICOLON
statement -> jump_statement
block_item -> statement
block_item_list -> block_item_list block_item

#################### LINE NO : 22 ####################
block_item_list_opt -> block_item_list
compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
function_definition -> declaration_specifiers declarator declaration_list_opt
compound_statement
external_declaration -> function_definition
translation_unit -> translation_unit external_declaration

Parsing Successful

```

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:

```c
#include <stdio.h>
#include <stdlib.h>

extern double func();

const int INVALID = -1;

static int demo(int x) {
    if(x == 0) {
        return 0;
    }
    else if(x % 4 == 0) {
        if(x % 2 == 0) {
            return x / 2;
        }
        else return x + 4;
    }
    else if(x % 3 == 0) {
        return (x - 1) * (x - 1);
    }
    else {
        return (x + 1) * (x + 1);
    }

    return INVALID;
}

int main() {
    int N = 100;
    // This is the comment to check parsing
    int ans = demo(N);
    ans += N;

    /* This is another type of comment to check parsing */
    printf("%d\n", ans);

    return 0;
}
```

Output:

```
################### LINE NO : 38 ###################
block_item_list_opt -> block_item_list
compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
function_definition -> declaration_specifiers declarator declaration_list_opt compound_statement
external_declaration -> function_definition
translation_unit -> translation_unit external_declaration

Parsing Successful

```

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:

```c
#include <stdio.h>
#include <stdlib.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void print_the_array(int A[], int n) {
    for(int i = 0; i < n; i++) {
        A[i] += 10;
        printf("%d ", A[i]);
    }
    printf("\n");
}

int main() {
    int A[] = {10, 7, 8, 9, 1, 5};
    int n = (int) sizeof(A) / (int) sizeof(A[0]);
    printf("Size of the array = %d\n", n);

    swap(&A[0], &A[n - 1]);
    print_the_array(A, n);

    return 0;
}
```

Output:

```
################## LINE NO : 27 ##################
block_item_list_opt -> block_item_list
compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
function_definition -> declaration_specifiers declarator declaration_list_opt compound_statement
external_declaration -> function_definition
translation_unit -> translation_unit external_declaration

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:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NONE -1

int main() {
    int i, ans = 0;
    for(int i = 0; i < 100; i++) {
        switch(i) {
            case(50) : continue;
            case(60) : break;
            default : ans++;
        }
    }

    if(ans >= 50 && ((ans <= 75) || (ans % 10 == 0))) {
        ans--;
    }
    else {
        ans++;
    }

    if(ans < -100 || ans > 100) {
        ans = NONE;
    }
    else if(ans % 2 == 0) {
        ans /= 2;
    }
    else {
        if(ans % 3 == 0 && ans <= 69) {
            ans /= 3;
        }
        else {
            ans = (ans << 2);
        }
    }

    printf("%d\n", ans);


    return 0;
}
```

Output:

```
################### LINE NO : 43 ###################
block_item_list_opt -> block_item_list
compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
function_definition -> declaration_specifiers declarator declaration_list_opt compound_statement
external_declaration -> function_definition
translation_unit -> external_declaration

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:

```c
#include <stdio.h>

int main() {
    int n = 0;

    do {
        if(n % 2 == 0)
            n++;
        else if(n % 3 == 0)
            n += 2;
        else
            n += 4;

    } while(n < 100);

    while(1) {
        if(n == 0) {
            break;
        }

        if(n % 10 == 0)
            n /= 10;
        else if(n % 3 == 0)
            n /= 3;
        else
            n /= 2;
    }

    printf("%d\n", n);

    return 0;
}
```

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:

```c
#include <stdio.h>

const int INF = (int) 1e9;

int max(int a, int b) {
    int ans;
    if(a > b) {
        ans = a;
    }
    else {
        ans = b;
    }
    return ans;
}

int main() {
    int value[100];
    int weight[100];
    int n;
    int W;
    printf("Enter the number of items and max. weight of Knapsack: ");
    scanf("%d %d", &n, &W);

    printf("Enter the value of items\n");
    for(int i = 0; i < n; i++) {
        scanf("%d", &value[i]);
    }
    printf("Enter the weight of items\n");
    for(int i = 0; i < n; i++) {
        scanf("%d", &weight[i]);
    }
    int dp[100][100];
    for(int i = 0; i <= n; i++) {
        for(int j = 0; j <= W; j++) {
            if(i == 0 || j == 0) {
                dp[i][j] = 0;
            }
            else {
                int choice1 = dp[i - 1][j];
                int choice2 = (j >= weight[i] ? (dp[i - 1][j - weight[i]] + value[i]) : -INF);
                dp[i][j] = max(choice1, choice2);
            }
        }
    }

    printf("%d", dp[n][W]);

    return 0;
}
```

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:

```c
#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;
}
```

Output:

```
################### LINE NO : 7 ###################
type_specifier -> INT
declaration_specifiers -> type_specifier declaration_specifiers_opt
direct_declarator -> IDENTIFIER
declarator -> pointer_opt direct_declarator
primary_expression -> IDENTIFIER
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
AND_expression -> equality_expression
exclusive_OR_expression -> AND_expression
inclusive_OR_expression -> exclusive_OR_expression
logical_AND_expression -> inclusive_OR_expression
logical_OR_expression -> logical_AND_expression
conditional_expression -> logical_OR_expression
assignment_expression -> conditional_expression
initializer -> assignment_expression
init_declarator -> declarator ASSIGN initializer
init_declarator_list -> init_declarator
init_declarator_list_opt -> init_declarator_list
syntax error

Parsing Unsuccessful

```

2. Code:

```c
#include<stdio.h>

int main()
{
    int a = 1, b = 2;
    if(a > b)
        a += 2;

    b += 2;

    else
        b += 2;

    return 0;
}
```

Output:

```
#################### LINE NO : 8 ####################

#################### LINE NO : 9 ####################
selection_statement -> selection_statement_base _ THEN
statement -> selection_statement
block_item -> statement
block_item_list -> block_item_list block_item
primary_expression -> IDENTIFIER
postfix_expression -> primary_expression
unary_expression -> postfix_expression
assignment_operator -> PLUS_EQUAL
const -> INTEGER_CONSTANT
primary_expression -> const
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
AND_expression -> equality_expression
exclusive_OR_expression -> AND_expression
inclusive_OR_expression -> exclusive_OR_expression
logical_AND_expression -> inclusive_OR_expression
logical_OR_expression -> logical_AND_expression
conditional_expression -> logical_OR_expression
assignment_expression -> conditional_expression
assignment_expression -> unary_expression assignment_operator assignment_expression
expression -> assignment_expression
exp_opt -> expression
exp_statement -> exp_opt SEMICOLON
statement -> exp_statement
block_item -> statement
block_item_list -> block_item_list block_item

#################### LINE NO : 10 ####################

#################### LINE NO : 11 ####################
block_item_list_opt -> block_item_list
syntax error

Parsing Unsuccessful

```

3. Code:

```c
#include<stdio.h>

int main()
{
    char str[] = "Hello World!";
    str[5;] = 't';
    return 0;
}
```

Output:

```
##################### LINE NO : 6 #####################
primary_expression -> IDENTIFIER
postfix_expression -> primary_expression
const -> INTEGER_CONSTANT
primary_expression -> const
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
AND_expression -> equality_expression
exclusive_OR_expression -> AND_expression
inclusive_OR_expression -> exclusive_OR_expression
logical_AND_expression -> inclusive_OR_expression
logical_OR_expression -> logical_AND_expression
conditional_expression -> logical_OR_expression
assignment_expression -> conditional_expression
expression -> assignment_expression
syntax error

Parsing Unsuccessful

```

4. Code:

```c
#include <stdio.h>

int main()
{
    int a[5] = {1, 2, 3, 4, 5};
    int i = 0;
    for(i = 0; i < 5; i++; i++) {
        a[i] += 1;
    }

    return 0;
}
```

Output:

```
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> relational_expression LESS shift_expression
equality_expression -> relational_expression
AND_expression -> equality_expression
exclusive_OR_expression -> AND_expression
inclusive_OR_expression -> exclusive_OR_expression
logical_AND_expression -> inclusive_OR_expression
logical_OR_expression -> logical_AND_expression
conditional_expression -> logical_OR_expression
assignment_expression -> conditional_expression
expression -> assignment_expression
exp_opt -> expression
primary_expression -> IDENTIFIER
postfix_expression -> primary_expression
postfix_expression -> postfix_expression INC
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
AND_expression -> equality_expression
exclusive_OR_expression -> AND_expression
inclusive_OR_expression -> exclusive_OR_expression
logical_AND_expression -> inclusive_OR_expression
logical_OR_expression -> logical_AND_expression
conditional_expression -> logical_OR_expression
assignment_expression -> conditional_expression
expression -> assignment_expression
exp_opt -> expression
syntax error

Parsing Unsuccessful
```

5. Code:

```c
#include <stdio.h>

int fib(int a, int b)
{
    if(a <= 1)
        return a;
    return fib(a - 1, b) + fib(a - 2, b - 1);
}

int main()
{
    int n = 9;
    int sum = 0;
    for(int i = 1; i <= n; i++) {
        sum += fib(i, 0);

    printf("%d\n", sum);

    return 0;
}
```

Output:

```
###################### LINE NO : 18 ######################

###################### LINE NO : 19 ######################
const -> INTEGER_CONSTANT
primary_expression -> const
postfix_expression -> primary_expression
unary_expression -> postfix_expression
cast_expression -> unary_expression
multiplicative_expression -> cast_expression
additive_expression -> multiplicative_expression
shift_expression -> additive_expression
relational_expression -> shift_expression
equality_expression -> relational_expression
AND_expression -> equality_expression
exclusive_OR_expression -> AND_expression
inclusive_OR_expression -> exclusive_OR_expression
logical_AND_expression -> inclusive_OR_expression
logical_OR_expression -> logical_AND_expression
conditional_expression -> logical_OR_expression
assignment_expression -> conditional_expression
expression -> assignment_expression
exp_opt -> expression
jump_statement -> RETURN exp_opt SEMICOLON
statement -> jump_statement
block_item -> statement
block_item_list -> block_item_list block_item

###################### LINE NO : 20 ######################
block_item_list_opt -> block_item_list
compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
statement -> compound_statement
iteration_statement -> FOR LEFT_PAREN declaration exp_opt SEMICOLON exp_opt RIGHT_PAREN statement
statement -> iteration_statement
block_item -> statement
block_item_list -> block_item_list block_item
block_item_list_opt -> block_item_list
syntax error

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:

```c
#include <stdio.h>

struct coord
{
    int x, y;
};

int main() {
    struct coord A;
    A.x = 10;
    A.y = 20;
    printf("%d %d\n", A.x, A.y);

    return 0;
}
```

   Output:

```
Parsing begins


#################### LINE NO : 2 ####################

#################### LINE NO : 3 ####################
syntax error

Parsing Unsuccessful

```

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:

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{

    int choice = 2;
    int temp = 1;
    switch(choice);
    {
        case 1:
            temp = 2;
            break;
        case 2:
            temp = 3;
            break;
        case 3:
            temp = 4;
            break;
        case 4:
            temp = 5;
            break;
        default:
            temp = 6;
    }

    return 0;
}
```

   Output:

```
################### LINE NO : 28 ###################
block_item_list_opt -> block_item_list
compound_statement -> LEFT_BRACKET block_item_list_opt RIGHT_BRACKET
function_definition -> declaration_specifiers declarator declaration_list_opt compound_statement
external_declaration -> function_definition
translation_unit -> external_declaration

Parsing Successful

```

Bibliographic resources:

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