1. Original Grammar

W -> PROGRAM B; D BEGIN I END.

B -> UC

C -> OC | UC | λ

D -> H : G;

G -> B, G | B

H -> INTEGER

I -> J | JI

J -> K | A

K -> PRINT(B);

A -> B = E;

E -> E + T | E – T | T

T -> T * F | T / F | F

F -> B | N | (E)

N -> LOM

L -> +|-|λ

M -> OM| λ

O -> 0|1|2|3|4|5|6|7|8|9

U -> P | Q | R | S

Non-Terminal	
<prog>	W
<identifier>	B
<more-id-digit>	C
<dec-list>	D
<dec>	G
<type>	H
<stat-list>	I
<stat>	J
<print>	K
<assign>	A
<expr>	E
<term>	T
<factor>	F
<number>	N
<sign>	L
<more-digit>	M
<digit>	O
<id>	U

2. BNF Grammar

W -> PROGRAM B; D BEGIN I END.
B -> UC
C -> OC
C -> UC
C -> λ
D -> H : G;
G -> B, G
G -> B
H -> INTEGER
I -> J
I -> JI
J -> K
J -> A
K -> PRINT(B);
A -> B = E;
E -> E + T
E -> E – T
E -> T
T -> T * F
T -> T / F
T -> F
F -> B
F -> N
F -> (E)
N -> LOM
L -> +
L -> -
L -> λ
M -> OM
M -> λ
O -> 0
O -> 1
O -> 2
O -> 3
O -> 4
O -> 5
O -> 6
O -> 7
O -> 8
O -> 9
U -> P
U -> Q
U -> R
U -> S

Non-Terminal	
<prog>	W
<identifier>	B
<more-id-digit>	C
<dec-list>	D
<dec>	G
<type>	H
<stat-list>	I
<stat>	J
<print>	K
<assign>	A
<expr>	E
<term>	T
<factor>	F
<number>	N
<sign>	L
<more-digit>	M
<digit>	O
<id>	U

3. Grammar for Table 1 (LL Table)

Grammar

W -> PROGRAM B; D BEGIN I END.
B -> UC
C -> OC
C -> UC
C -> lambda
D -> H : G;
G -> BZ
Z -> ,G
Z -> lambda
H -> INTEGER
I -> JV
V -> I
V -> lambda
J -> K
J -> A
K -> PRINT(B);
A -> B = E;
E -> TX
X -> +TX
X -> -TX
X -> lambda
T -> FY
Y -> /FY
Y -> *FY
Y -> lambda
F -> B
F -> N
F -> (E)
N -> LOM
L -> +
L -> -
L -> lambda
M -> OM
M -> lambda
O -> 0
O -> 1
O -> 2
O -> 3
O -> 4
O -> 5
O -> 6
O -> 7
O -> 8
O -> 9
U -> P
U -> Q
U -> R
U -> S

Non-Terminal	
<prog>	W
<identifier>	B
<more-id-digit>	C
<dec-list>	D
<dec>	G
<type>	H
<stat-list>	I
<stat>	J
<print>	K
<assign>	A
<expr>	E
<term>	T
<factor>	F
<number>	N
<sign>	L
<more-digit>	M
<digit>	O
<id>	U

4. Members of FIRST and FOLLOW sets

Non-terminals	FIRST	FOLLOW
<prog>	PROGRAM	$
<identifier>	P Q R S	) ; + - / * = ,
<more-id-digit>	0 1 2 3 4 5 6 7 8 9 P Q R S lambda	) ; + - / * = ,
<dec-list>	INTEGER	BEGIN
<dec>	P Q R S	;
Z	, lambda	;
<type>	INTEGER	:
<stat-list>	P Q R S PRINT	END.
V	P Q R S PRINT lambda	END.
<stat>	P Q R S PRINT	P Q R S PRINT END.
<print>	PRINT	P Q R S PRINT END.
<assign>	P Q R S	P Q R S PRINT END.
<expr>	P Q R S + - (0 1 2 3 4 5 6 7 8 9	) ;
X	+ - lambda	) ;
<term>	P Q R S + - (0 1 2 3 4 5 6 7 8 9	) ; + -
Y	/ * lambda	) ; + -
<factor>	P Q R S + - (0 1 2 3 4 5 6 7 8 9	) ; + - / *
<number>	0 1 2 3 4 5 6 7 8 9 + -	) ; + - / *
<sign>	+ - lambda	0 1 2 3 4 5 6 7 8 9
<more-digit>	0 1 2 3 4 5 6 7 8 9	) ; + - / *
<digit>	0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9) ; + - / * P Q R S = ,
<id>	P Q R S	0 1 2 3 4 5 6 7 8 9 P Q R S) ; + - / * = ,

E	A	K	J	V	I	H	Z	G	D	C	B	W	
TX	B = E ;		A	I	JV			BZ		UC	UC		**P**
TX	B = E ;		A	I	JV			BZ		UC	UC		**Q**
TX	B = E ;		A	I	JV			BZ		UC	UC		**R**
TX	B = E ;		A	I	JV			BZ		UC	UC		**S**
TX										OC			**0**
TX										OC			**1**
TX										OC			**2**
TX										OC			**3**
TX										OC			**4**
TX										OC			**5**
TX										OC			**6**
TX										OC			**7**
TX										OC			**8**
TX										OC			**9**
												PROGRAM B ; D BEGIN I END.	**PROGRAM**
													BEGIN
				lambda									**END.**
						INTEGER			H : G ;				**INTEGER**
		PRINT(B);	K	I	JV								**PRINT**
TX										lambda			**+**
TX										lambda			**-**
										lambda			**/**
										lambda			*****
TX													**(**
										lambda			**)**
							, G			lambda			**,**
							lambda			lambda			**;**
										lambda			**=**
													:
													$

Non-Terminal	
<prog>	W
<identifier>	B
<more-id-digit>	C
<dec-list>	D
<dec>	G
<type>	H
<stat-list>	I
<stat>	J
<print>	K
<assign>	A
<expr>	E
<term>	T
<factor>	F
<number>	N
<sign>	L
<more-digit>	M
<digit>	O
<id>	U

U	O	M	L	N	F	Y	T	X
P					B		FY	
Q					B		FY	
R					B		FY	
S					B		FY	
	0	OM	lambda	LOM	N		FY	
	1	OM	lambda	LOM	N		FY	
	2	OM	lambda	LOM	N		FY	
	3	OM	lambda	LOM	N		FY	
	4	OM	lambda	LOM	N		FY	
	5	OM	lambda	LOM	N		FY	
	6	OM	lambda	LOM	N		FY	
	7	OM	lambda	LOM	N		FY	
	8	OM	lambda	LOM	N		FY	
	9	OM	lambda	LOM	N		FY	
		lambda	+	LOM	N	lambda	FY	"+TX"
		lambda	-	LOM	N	lambda	FY	"-TX"
		lambda				/FY		
		lambda				*FY		
					(E)		FY	
		lambda				lambda		lambda
		lambda				lambda		lambda

```
#!usr/bin/env python3
# FILE: compiler.py
# FINAL PROJECT
# Professor Ahmadnia
# Group: Kevin Vuong, Anika Corpus, Christopher Grant
# Description: This program reads source code from a file, outputs it w/o comments and
properly formatted spaces.
#              It then tokenizes and then parses the source code and checks if it has
correct grammar.

from text_clean import *

from tokenizer import *

from LL_parser import *

from code_generator import *


def main():
    # First, remove all the comments
    with open("finalv1.txt") as source_file:
        new_file = open('finalv2.txt', mode='w+', encoding='utf-8')
        comment_remover(source_file, new_file)
        new_file.close()

    # Clean the spaces
    content = clean_text('finalv2.txt')

    token_list = tokenizer(content)

    if token_list == -1:
        exit(1)

    terminal_list = ['P', 'Q', 'R', 'S', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
'PROGRAM', 'BEGIN', 'END.',
                     'INTEGER', 'PRINT', '+', '-', '/', '*', '(', ')', ',', ';', '=', ':',
 '$']

    predict_table = {
        'W': {'P': 'aaa', 'Q': 'aaa', 'R': 'aaa', 'S': 'aaa', '0': 'aaa', '1': 'aaa', '2':
'aaa', '3': 'aaa', '4': 'aaa', '5': 'aaa', '6': 'aaa', '7': 'aaa', '8': 'aaa', '9': 'aaa',
'PROGRAM': 'PROGRAM B ; D BEGIN I END.', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', '
PRINT': 'aaa', '+': 'aaa', '-': 'aaa', '/': 'aaa', '*': 'aaa', '(': 'aaa', ')': 'aaa', ',':
 'aaa', '.': 'aaa', ';': 'aaa', '=': 'aaa', ':': 'aaa', '$': 'aaa'},
        'B': {'P': 'UC', 'Q': 'UC', 'R': 'UC', 'S': 'UC', '0': 'aaa', '1': 'aaa', '2': 'aaa
', '3': 'aaa', '4': 'aaa', '5': 'aaa', '6': 'aaa', '7': 'aaa', '8': 'aaa', '9': 'aaa', '
PROGRAM': 'aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'aaa', '+': 'aaa
', '-': 'aaa', '/': 'aaa', '*': 'aaa', '(': 'aaa', ')': 'aaa', ',': 'aaa', '.': 'aaa', ';':
 'aaa', '=': 'aaa', ':': 'aaa', '$': 'aaa'},
        'C': {'P': 'UC', 'Q': 'UC', 'R': 'UC', 'S': 'UC', '0': 'OC', '1': 'OC', '2': 'OC',
'3': 'OC', '4': 'OC', '5': 'OC', '6': 'OC', '7': 'OC', '8': 'OC', '9': 'OC', 'PROGRAM': '
aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'aaa', '+': 'lambda', '-':
'lambda', '/': 'lambda', '*': 'lambda', '(': 'aaa', ')': 'lambda', ',': 'lambda', '.': 'aaa
', ';': 'lambda', '=': 'lambda', ':': 'aaa', '$': 'aaa'},
        'D': {'P': 'aaa', 'Q': 'aaa', 'R': 'aaa', 'S': 'aaa', '0': 'aaa', '1': 'aaa', '2':
'aaa', '3': 'aaa', '4': 'aaa', '5': 'aaa', '6': 'aaa', '7': 'aaa', '8': 'aaa', '9': 'aaa',
'PROGRAM': 'aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'H : G ;', 'PRINT': 'aaa', '+':
 'aaa', '-': 'aaa', '/': 'aaa', '*': 'aaa', '(': 'aaa', ')': 'aaa', ',': 'aaa', '.': 'aaa',
 ';': 'aaa', '=': 'aaa', ':': 'aaa', '$': 'aaa'},
        'G': {'P': 'BZ', 'Q': 'BZ', 'R': 'BZ', 'S': 'BZ', '0': 'aaa', '1': 'aaa', '2': 'aaa
', '3': 'aaa', '4': 'aaa', '5': 'aaa', '6': 'aaa', '7': 'aaa', '8': 'aaa', '9': 'aaa', '
PROGRAM': 'aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'aaa', '+': 'aaa
', '-': 'aaa', '/': 'aaa', '*': 'aaa', '(': 'aaa', ')': 'aaa', ',': 'aaa', '.': 'aaa', ';':
 'aaa', '=': 'aaa', ':': 'aaa', '$': 'aaa'},
        'Z': {'P': 'aaa', 'Q': 'aaa', 'R': 'aaa', 'S': 'aaa', '0': 'aaa', '1': 'aaa', '2':
'aaa', '3': 'aaa', '4': 'aaa', '5': 'aaa', '6': 'aaa', '7': 'aaa', '8': 'aaa', '9': 'aaa',
'PROGRAM': 'aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'aaa', '+': '
aaa', '-': 'aaa', '/': 'aaa', '*': 'aaa', '(': 'aaa', ')': 'aaa', ',': ', G', '.': 'aaa',
';': 'lambda', '=': 'aaa', ':': 'aaa', '$': 'aaa'},
```

```
        'H': {'P': 'aaa', 'Q': 'aaa', 'R': 'aaa', 'S': 'aaa', '0': 'aaa', '1': 'aaa', '2':
'aaa', '3': 'aaa', '4': 'aaa', '5': 'aaa', '6': 'aaa', '7': 'aaa', '8': 'aaa', '9': 'aaa',
'PROGRAM': 'aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'INTEGER', 'PRINT': 'aaa', '+':
 'aaa', '-': 'aaa', '/': 'aaa', '*': 'aaa', '(': 'aaa', ')': 'aaa', ',': 'aaa', '.': 'aaa',
 ';': 'aaa', '=': 'aaa', ':': 'aaa', '$': 'aaa'},
        'I': {'P': 'JV', 'Q': 'JV', 'R': 'JV', 'S': 'JV', '0': 'aaa', '1': 'aaa', '2': 'aaa
', '3': 'aaa', '4': 'aaa', '5': 'aaa', '6': 'aaa', '7': 'aaa', '8': 'aaa', '9': 'aaa', '
PROGRAM': 'aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'JV', '+': 'aaa'
, '-': 'aaa', '/': 'aaa', '*': 'aaa', '(': 'aaa', ')': 'aaa', ',': 'aaa', '.': 'aaa', ';':
'aaa', '=': 'aaa', ':': 'aaa', '$': 'aaa'},
        'V': {'P': 'I', 'Q': 'I', 'R': 'I', 'S': 'I', '0': 'aaa', '1': 'aaa', '2': 'aaa', '
3': 'aaa', '4': 'aaa', '5': 'aaa', '6': 'aaa', '7': 'aaa', '8': 'aaa', '9': 'aaa', 'PROGRAM
': 'aaa', 'BEGIN': 'aaa', 'END.': 'lambda', 'INTEGER': 'aaa', 'PRINT': 'I', '+': 'aaa', '-'
: 'aaa', '/': 'aaa', '*': 'aaa', '(': 'aaa', ')': 'aaa', ',': 'aaa', '.': 'aaa', ';': 'aaa'
, '=': 'aaa', ':': 'aaa', '$': 'aaa'},
        'J': {'P': 'A', 'Q': 'A', 'R': 'A', 'S': 'A', '0': 'aaa', '1': 'aaa', '2': 'aaa', '
3': 'aaa', '4': 'aaa', '5': 'aaa', '6': 'aaa', '7': 'aaa', '8': 'aaa', '9': 'aaa', 'PROGRAM
': 'aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'K', '+': 'aaa', '-':
'aaa', '/': 'aaa', '*': 'aaa', '(': 'aaa', ')': 'aaa', ',': 'aaa', '.': 'aaa', ';': 'aaa',
'=': 'aaa', ':': 'aaa', '$': 'aaa'},
        'K': {'P': 'aaa', 'Q': 'aaa', 'R': 'aaa', 'S': 'aaa', '0': 'aaa', '1': 'aaa', '2':
'aaa', '3': 'aaa', '4': 'aaa', '5': 'aaa', '6': 'aaa', '7': 'aaa', '8': 'aaa', '9': 'aaa',
'PROGRAM': 'aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'PRINT ( B ) ;'
, '+': 'aaa', '-': 'aaa', '/': 'aaa', '*': 'aaa', '(': 'aaa', ')': 'aaa', ',': 'aaa', '.':
'aaa', ';': 'aaa', '=': 'aaa', ':': 'aaa', '$': 'aaa'},
        'A': {'P': 'B = E ;', 'Q': 'B = E ;', 'R': 'B = E ;', 'S': 'B = E ;', '0': 'aaa', '
1': 'aaa', '2': 'aaa', '3': 'aaa', '4': 'aaa', '5': 'aaa', '6': 'aaa', '7': 'aaa', '8': '
aaa', '9': 'aaa', 'PROGRAM': 'aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT
': 'aaa', '+': 'aaa', '-': 'aaa', '/': 'aaa', '*': 'aaa', '(': 'aaa', ')': 'aaa', ',': 'aaa
', '.': 'aaa', ';': 'aaa', '=': 'aaa', ':': 'aaa', '$': 'aaa'},
        'E': {'P': 'TX', 'Q': 'TX', 'R': 'TX', 'S': 'TX', '0': 'TX', '1': 'TX', '2': 'TX',
'3': 'TX', '4': 'TX', '5': 'TX', '6': 'TX', '7': 'TX', '8': 'TX', '9': 'TX', 'PROGRAM': '
aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'aaa', '+': 'TX', '-': 'TX'
, '/': 'aaa', '*': 'aaa', '(': 'TX', ')': 'aaa', ',': 'aaa', '.': 'aaa', ';': 'aaa', '=': '
aaa', ':': 'aaa', '$': 'aaa'},
        'X': {'P': 'aaa', 'Q': 'aaa', 'R': 'aaa', 'S': 'aaa', '0': 'aaa', '1': 'aaa', '2':
'aaa', '3': 'aaa', '4': 'aaa', '5': 'aaa', '6': 'aaa', '7': 'aaa', '8': 'aaa', '9': 'aaa',
'PROGRAM': 'aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'aaa', '+': '+
TX', '-': '-TX', '/': 'aaa', '*': 'aaa', '(': 'aaa', ')': 'lambda', ',': 'aaa', '.': 'aaa',
 ';': 'lambda', '=': 'aaa', ':': 'aaa', '$': 'aaa'},
        'T': {'P': 'FY', 'Q': 'FY', 'R': 'FY', 'S': 'FY', '0': 'FY', '1': 'FY', '2': 'FY',
'3': 'FY', '4': 'FY', '5': 'FY', '6': 'FY', '7': 'FY', '8': 'FY', '9': 'FY', 'PROGRAM': '
aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'aaa', '+': 'FY', '-': 'FY'
, '/': 'aaa', '*': 'aaa', '(': 'FY', ')': 'aaa', ',': 'aaa', '.': 'aaa', ';': 'aaa', '=': '
aaa', ':': 'aaa', '$': 'aaa'},
        'Y': {'P': 'aaa', 'Q': 'aaa', 'R': 'aaa', 'S': 'aaa', '0': 'aaa', '1': 'aaa', '2':
'aaa', '3': 'aaa', '4': 'aaa', '5': 'aaa', '6': 'aaa', '7': 'aaa', '8': 'aaa', '9': 'aaa',
'PROGRAM': 'aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'aaa', '+': '
lambda', '-': 'lambda', '/': '/FY', '*': '*FY', '(': 'aaa', ')': 'lambda', ',': 'aaa', '.':
 'aaa', ';': 'lambda', '=': 'aaa', ':': 'aaa', '$': 'aaa'},
        'F': {'P': 'B', 'Q': 'B', 'R': 'B', 'S': 'B', '0': 'N', '1': 'N', '2': 'N', '3': 'N
', '4': 'N', '5': 'N', '6': 'N', '7': 'N', '8': 'N', '9': 'N', 'PROGRAM': 'aaa', 'BEGIN': '
aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'aaa', '+': 'N', '-': 'N', '/': 'aaa', '*':
 'aaa', '(': '( E )', ')': 'aaa', ',': 'aaa', '.': 'aaa', ';': 'aaa', '=': 'aaa', ':': 'aaa
', '$': 'aaa'},
        'N': {'P': 'aaa', 'Q': 'aaa', 'R': 'aaa', 'S': 'aaa', '0': 'LOM', '1': 'LOM', '2':
'LOM', '3': 'LOM', '4': 'LOM', '5': 'LOM', '6': 'LOM', '7': 'LOM', '8': 'LOM', '9': 'LOM',
'PROGRAM': 'aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'aaa', '+': '
LOM', '-': 'LOM', '/': 'aaa', '*': 'aaa', '(': 'aaa', ')': 'aaa', ',': 'aaa', '.': 'aaa',
';': 'aaa', '=': 'aaa', ':': 'aaa', '$': 'aaa'},
        'L': {'P': 'aaa', 'Q': 'aaa', 'R': 'aaa', 'S': 'aaa', '0': 'lambda', '1': 'lambda',
 '2': 'lambda', '3': 'lambda', '4': 'lambda', '5': 'lambda', '6': 'lambda', '7': 'lambda',
'8': 'lambda', '9': 'lambda', 'PROGRAM': 'aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': '
aaa', 'PRINT': 'aaa', '+': '+', '-': '-', '/': 'aaa', '*': 'aaa', '(': 'aaa', ')': 'aaa',
',': 'aaa', '.': 'aaa', ';': 'aaa', '=': 'aaa', ':': 'aaa', '$': 'aaa'},
        'M': {'P': 'aaa', 'Q': 'aaa', 'R': 'aaa', 'S': 'aaa', '0': 'OM', '1': 'OM', '2': '
OM', '3': 'OM', '4': 'OM', '5': 'OM', '6': 'OM', '7': 'OM', '8': 'OM', '9': 'OM', 'PROGRAM'
: 'aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'aaa', '+': 'lambda',
'-': 'lambda', '/': 'lambda', '*': 'lambda', '(': 'aaa', ')': 'lambda', ',': 'aaa', '.': '
aaa', ';': 'lambda', '=': 'aaa', ':': 'aaa', '$': 'aaa'},
```

```
        'O': {'P': 'aaa', 'Q': 'aaa', 'R': 'aaa', 'S': 'aaa', '0': '0', '1': '1', '2': '2',
 '3': '3', '4': '4', '5': '5', '6': '6', '7': '7', '8': '8', '9': '9', 'PROGRAM': 'aaa', '
BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'aaa', '+': 'aaa', '-': 'aaa', '/'
: 'aaa', '*': 'aaa', '(': 'aaa', ')': 'aaa', ',': 'aaa', '.': 'aaa', ';': 'aaa', '=': 'aaa'
, ':': 'aaa', '$': 'aaa'},
        'U': {'P': 'P', 'Q': 'Q', 'R': 'R', 'S': 'S', '0': 'aaa', '1': 'aaa', '2': 'aaa', '
3': 'aaa', '4': 'aaa', '5': 'aaa', '6': 'aaa', '7': 'aaa', '8': 'aaa', '9': 'aaa', 'PROGRAM
': 'aaa', 'BEGIN': 'aaa', 'END.': 'aaa', 'INTEGER': 'aaa', 'PRINT': 'aaa', '+': 'aaa', '-'
: 'aaa', '/': 'aaa', '*': 'aaa', '(': 'aaa', ')': 'aaa', ',': 'aaa', '.': 'aaa', ';': 'aaa'
, '=': 'aaa', ':': 'aaa', '$': 'aaa'},
    }

    parse_status = predictive_parser(token_list, predict_table, terminal_list,
starting_symbol='W')

    if parse_status is True:
        code_generator(content.split('\n'), 'main.cpp')

if __name__ == "__main__":
    main()
```

```
# FILE: text_clean.py
# FINAL PROJECT
# Professor Ahmadnia
# Group: Kevin Vuong, Anika Corpus, Christopher Grant
# Description: This program provides functionality for removing comments and formatting the
spaces.

import re


def comment_remover(file_read, file_write):
    """
    Removes the comments from the text file (file_read) and writes it to another file (
file_write).

    :param file_read: The text file being read from
    :param file_write: The text file being written into
    :return: None
    """
    content = file_read.readlines()
    content = ''.join(content)

    # The pattern to remove multi-line comments
    mult_line_comment_pattern = '//.*\n.*//'
    content = re.sub(mult_line_comment_pattern, '', content, 0)

    # The pattern to remove single-line comments
    single_line_comment_pattern = '//.*//'
    content = re.sub(single_line_comment_pattern, '', content, 0)

    file_write.writelines(content)


def space_formatter(expr):
    """
    Formats the spaces of a line of text

    Arguments:
        expr: a line of text

    Returns:
        line: the line of processed text

    """
    token_list = expr.split(" ")
    content = ''

    # This loop removes elements with empty string contents
    for token in token_list:
        if re.match(r'^\s*\n*\s*$', token):
            continue
        content = content + token.strip()

    # This section adds the appropriate spaces for the reserved words
    reserved_pattern = r'(\s*PROGRAM\s*|\s*INTEGER\s*|\s*PRINT\s*|\s*BEGIN\s*|\s*END\.\s*)'
    matched = re.match(reserved_pattern, content)
    word = ''
    if matched is not None:
        word = matched.group()

    if word == 'PROGRAM':
        content = re.sub(r'(\s*PROGRAM\s*)', 'PROGRAM ', content, 0)

    if word == 'INTEGER':
        content = re.sub(r'(\s*INTEGER\s*)', 'INTEGER ', content, 0)

    if word == 'PRINT':
        content = re.sub(r'(\s*PRINT\s*)', 'PRINT ', content, 0)

    if word == 'BEGIN':
```

```
            content = re.sub(r'\s*BEGIN\s*', 'BEGIN', content, 0)

        if word == 'END.':
            content = re.sub(r'\s*END.\s*', 'END.', content, 0)

        # This section adds the appropriate spaces for the symbols
        symbolic_pattern = r'(\=|\*|\-|\,|\:|\(|\)|\<\=|\+|\;)'
        matched = re.findall(symbolic_pattern, content)

        for word in matched:
            if word == '=':
                content = re.sub(r'\s*=\s*', ' = ', content, 0)

            if word == ',':
                content = re.sub(r'\s*,\s*', ' , ', content, 0)

            if word == ';':
                content = re.sub(r'\s*;\s*', ' ;', content, 0)

            if word == '(':
                content = re.sub(r'\s*\(\s*', ' ( ', content, 0)

            if word == ')':
                content = re.sub(r'\s*\)\s*', ' ) ', content, 0)

            if word == '+':
                content = re.sub(r'\s*\+\s*', ' + ', content, 0)

            if word == '-':
                content = re.sub(r'\s*-\s*', ' - ', content, 0)

            if word == '*':
                content = re.sub(r'\s*\*\s*', ' * ', content, 0)

            if word == ':':
                content = re.sub(r'\s*:\s*', ' : ', content, 0)

        return content+'\n'


    def clean_text(filename):
        """
        Cleans up the spaces in the text file.

        :rtype: lines_content: The entire content of the string cleaned up
        :param filename: The name of the file you want to clean
        """
        file = open('finalv2.txt', mode='r+', encoding='utf-8')
        lines_read = file.readlines()
        lines_content = ''

        for line in lines_read:

            # Ignore lines that only contain the newline character
            if re.match(pattern=r'\s*\n\s*', string=line):
                continue

            line = space_formatter(line)
            lines_content = lines_content + line

        # Writes the cleaned up text to the text file
        with open(filename, mode='w+') as new_file:
            new_file.writelines(lines_content.strip())

        return lines_content.strip()


    def main():

        # First, remove all the comments
```

```
    with open("finalv1.txt") as source_file:
        new_file = open('finalv2.txt', mode='w+', encoding='utf-8')
        comment_remover(source_file, new_file)
        new_file.close()

    # Clean the spaces
    content = clean_text('finalv2.txt')
    print(content)


if __name__ == "__main__":
    main()
```

```
# FILE: token.py
# FINAL PROJECT
# Professor Ahmadnia
# Group: Kevin Vuong, Anika Corpus, Christopher Grant
# Description: The token class enables parsing of the string input.

class Token:
    """
    A token class that categorizes a string. The 'token' is the category and the 'value' is
the specific string of the
    category.
    """
    def __init__(self, token, value, line_number):
        self.token = token
        self.value = value
        self.line_number = line_number

    def get_type(self):
        return self.token

    def get_value(self):
        return self.value

    def get_line_num(self):
        return self.line_number
```

```
# FILE: tokenizer.py
# FINAL PROJECT
# Professor Ahmadnia
# Group: Kevin Vuong, Anika Corpus, Christopher Grant
# Description: Provides a function to parse the source code for tokens.

import re

from token import *


def tokenizer(string):
    """
    Creates a list of tokens from the input string

    :param string: The source code
    :return: A list of tokens. Returns -1 if a token is not recognizable.
    """
    content_lines = string.split('\n')

    token_list = []
    reserved_pattern = r'^(PROGRAM|INTEGER|PRINT|BEGIN|END\.)$'
    symbol_pattern = r'^(=|\/|\*|\+|\-|\;|\:|\,|\(|\))$'
    number_pattern = r'^(\+|\-)?[0-9]+$'
    identifier_pattern = r'^(P|Q|R|S)+(P|Q|R|S|[0-9])*$'

    line_number = 1  # used to keep track of line number
    for line in content_lines:
        words_list = line.split()
        for word in words_list:
            if re.match(reserved_pattern, word):  # Check for reserved word token
                reserved = Token('RESERVED', word, line_number)
                token_list.append(reserved)

            elif re.match(number_pattern, word):  # Check for number token
                for symbol in word:
                    if re.match('(\+|-)', symbol):
                        sign = Token('SIGN', symbol, line_number)
                        token_list.append(sign)
                    elif re.match('[0-9]', symbol):
                        digit = Token('DIGIT', symbol, line_number)
                        token_list.append(digit)

            elif re.match(symbol_pattern, word):  # Check for symbol token
                symbol = Token('SYMBOL', word, line_number)
                token_list.append(symbol)

            elif re.match(identifier_pattern, word):  # Check for identifier token
                for symbol in word:
                    if re.match(r'(P|Q|R|S)', symbol):
                        _id = Token('ID', symbol, line_number)
                        token_list.append(_id)
                    elif re.match(r'[0-9]', symbol):
                        more_id_digit = Token('MORE_ID_DIGIT', symbol, line_number)
                        token_list.append(more_id_digit)
            else:
                print('Unknown word: ' + word, 'on line', line_number)
                return -1

        line_number += 1
    # For debug purposes
    # for token in token_list:
    #     print(token.get_type(), ':', token.get_value(), ':', token.get_line_num())

    return token_list
```

```
#!usr/bin/env python3
# FILE: LL_parser.py
# FINAL PROJECT
# Professor Ahmadnia
# Group: Kevin Vuong, Anika Corpus, Christopher Grant
# Description: This program provides a function to parse a tokenized list of input string to
 check
#              whether the input string is a valid string based on the Predictive Parsing
table.

import re

from token import *


def syntax_error_handler_1(error_value, line_number):
    """
    Handles the error condition in the terminal if-block

    :param error_value: The symbol under question
    :param line_number: The line number where the error has occurred
    :return:
    """
    if re.match(r':', error_value):
        print('Line ' + str(line_number) + ':', 'Missing a colon (:)')
    elif re.match(r';', error_value):
        print('Line ' + str(line_number - 1) + ':', 'Expected a semicolon')
    elif re.match(r'\)', error_value):
        print('Line ' + str(line_number) + ':', 'Expected a )')


def syntax_error_handler_2(error_value, line_number):
    """
    Handles the error condition in the non-terminal if-block

    :param error_value: The symbol under question
    :param line_number: The line number where the error has occurred
    :return: None
    """
    error_line = 'Line ' + str(line_number - 1) + ':'

    if re.match(r'(P|Q|R|S|BEGIN)', error_value):
        print(error_line, 'Missing a semicolon (;)')
    elif re.match(r';', error_value):
        print('Line ' + str(line_number) + ':', 'Missing an expression')
    elif re.match(r'\)', error_value):
        print('Line ' + str(line_number) + ':', 'Invalid expression')


def predictive_parser(token_list, predict_table, terminal_list, starting_symbol):
    """
    Determines whether the input string is accepted or rejected based on the prediction
table.

    :param token_list: a list of tokens to parse (basically, the input string in tokenized
form)
    :param predict_table: the prediction table being used
    :param terminal_list: a list of terminals for the grammar
    :param starting_symbol: the symbol to which the grammar starts with
    :return: Returns -1 if input string is rejected, otherwise returns 0 if accepted
    """

    stack = ['$', starting_symbol]  # Push the end-of-input symbol and the starting symbol
    i = 0  # Keeps track which token is currently being read

    # Add ending symbol to the end of the input
    ending_symbol = Token('$', '$', 0)
    token_list.append(ending_symbol)

    while stack:  # loop until stack is empty
```

```
        top_of_stack = stack[len(stack) - 1]
        token_read = token_list[i]
        char_read = token_read.get_value()  # Gets the actual terminal of the token

        if top_of_stack in terminal_list:  # Terminal
            if top_of_stack == char_read:
                stack.pop()
                i = i + 1
            else:
                print('\n1: The grammar has rejected the input string')
                print(top_of_stack, char_read)
                syntax_error_handler_1(top_of_stack, token_read.get_line_num())
                return -1
        else:   # Non-terminal
            if predict_table[top_of_stack][char_read] is not 'aaa':  # If table entry is
not an empty entry
                entry = stack.pop()
                if predict_table[entry][char_read] is not 'lambda':
                    # Push the entry into the stack in reverse order
                    for symbol in reversed(predict_table[entry][char_read].split()):
                        if re.match(r'PROGRAM|BEGIN|END\.|INTEGER|PRINT', symbol):
                            stack.append(symbol)
                        else:
                            for non_terminal in reversed(symbol):
                                stack.append(non_terminal)
            else:
                print('2: The grammar has rejected the input string')
                print(top_of_stack, char_read)
                syntax_error_handler_2(char_read, token_read.get_line_num())
                return -1
        print(stack)

    print('The grammar has accepted the input string\n')
    return True
```

```
# FILE: code_generator.py
# FINAL PROJECT
# Professor Ahmadnia
# Group: Kevin Vuong, Anika Corpus, Christopher Grant
# Description: Provides function to generate C++ code with the given source code.

import re


def code_generator(source, filename):
    """
    Generates source code from input source code

    :param source: A list. Contains the source code (where each line is an element of the
list)
    :param filename: The name of the file you want to output the generated source code to
    :return: Returns True upon successful completion
    """
    # Stores the string to be written to the file.
    content = ''

    # Go through each line, converting it to C++
    for line in source:
        if re.match(r'PRINT', line):
            line = re.sub(r'PRINT\s*\(', '\tcout <<', line, 0)
            line = re.sub(r'\)\s*;', '<< endl ;\n', line, 0)
            content += line
            continue

        if re.match(r'^PROGRAM', line):
            content += '#include <iostream>\nusing namespace std ;\n'
        elif re.match(r'^BEGIN', line):
            content += 'int main()\n{\n'
        elif re.match(r'^INTEGER', line):
            line = re.sub(r'INTEGER\s*:', 'int', line)
            content += line + '\n'
        elif re.match(r'^(P|Q|R|S)+(P|Q|R|S|[0-9])*', line):
            content += '\t' + line + '\n'
        elif re.match(r'END\.', line):
            content += '\treturn 0 ;\n}'

    # Write the code generated to the file
    file = open(filename, mode='w')
    file.writelines(content)
    file.close()

    return True
```