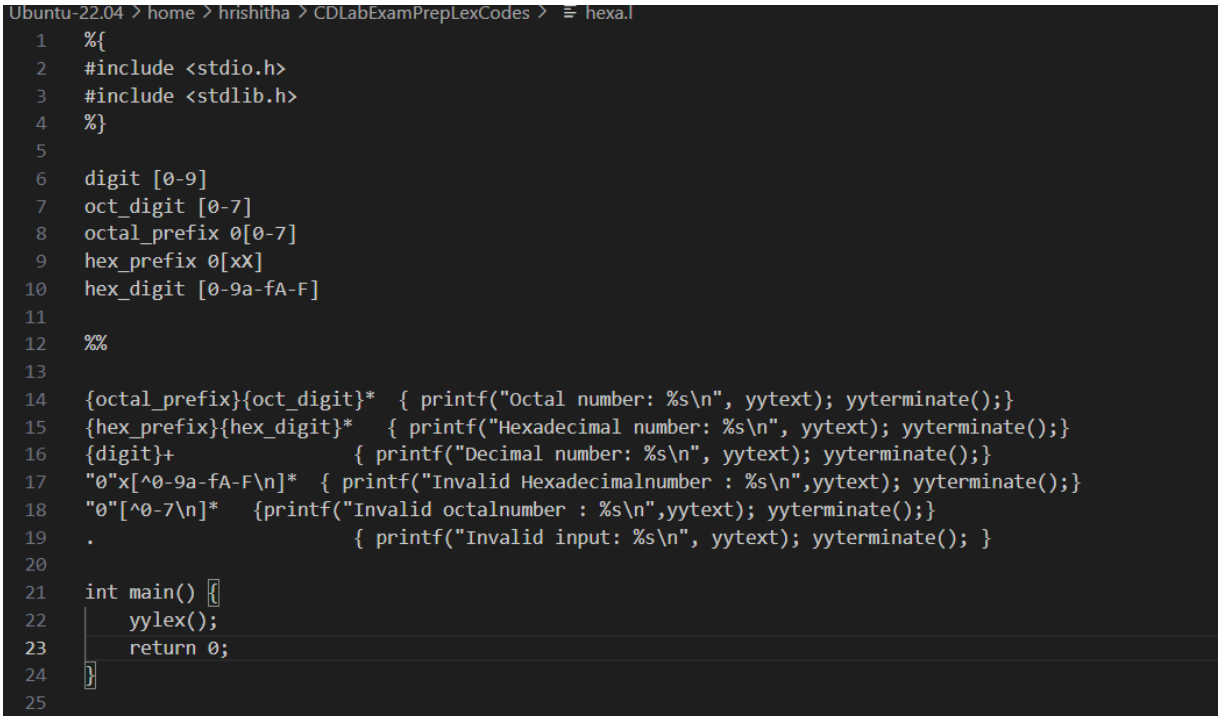
Week 2 codes:

1. Octal and hexadecimal

Code:

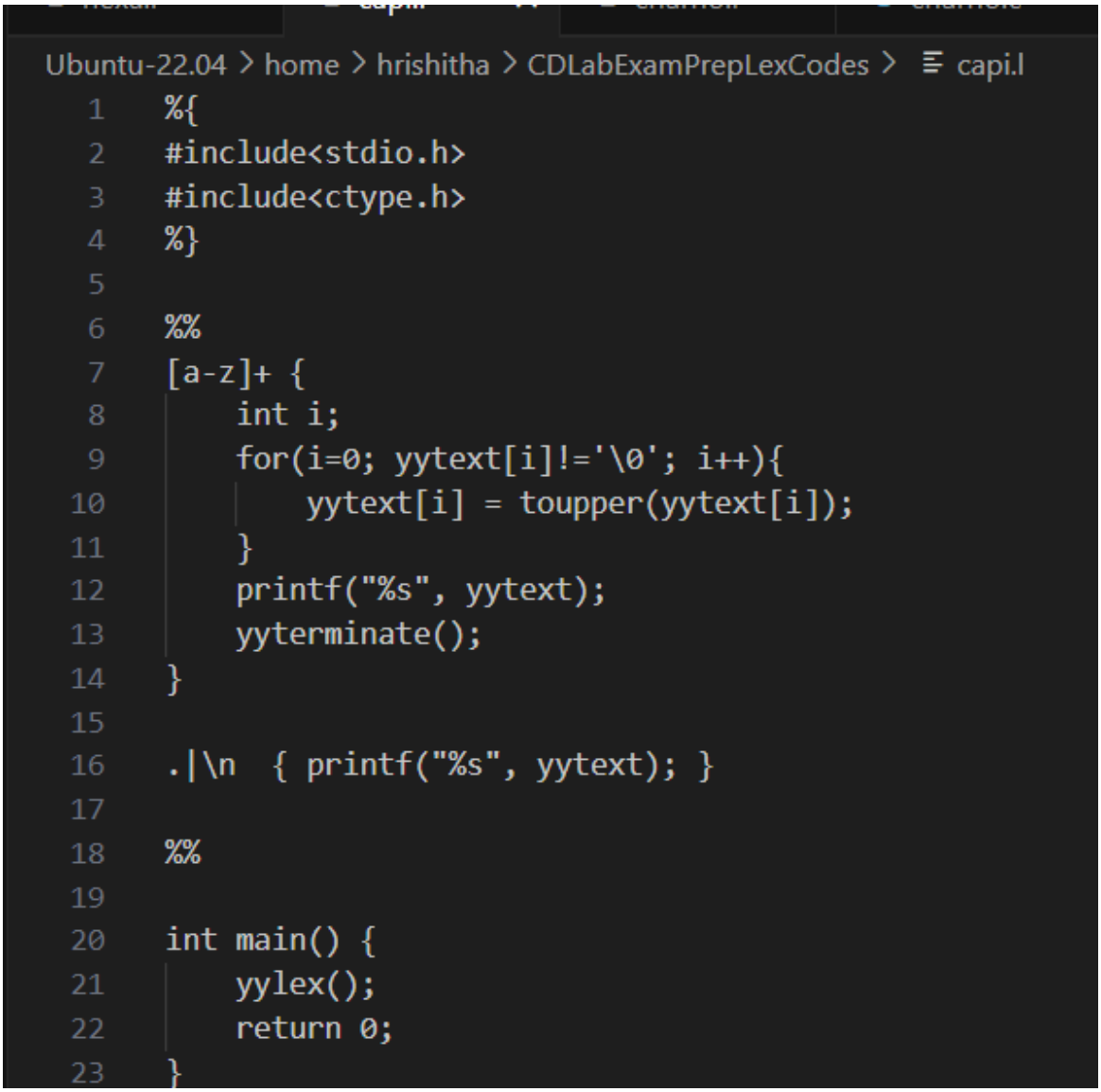



```
%{
#include <stdio.h>
#include <stdlib.h>
%}

digit [0-9]
oct_digit [0-7]
octal_prefix 0[0-7]
hex_prefix 0[xX]
hex_digit [0-9a-fA-F]

%%

{octal_prefix}{oct_digit}*        { printf("Octal number: %s\n", yytext); yyterminate();}
{hex_prefix}{hex_digit}*   { printf("Hexadecimal number: %s\n", yytext); yyterminate();}
{digit}+                  { printf("Decimal number: %s\n", yytext); yyterminate();}
"0"x[^0-9a-fA-F\n]*  { printf("Invalid Hexadecimalnumber : %s\n",yytext); yyterminate();}
"0"[^0-7\n]*   {printf("Invalid octalnumber : %s\n",yytext); yyterminate();}
.                   { printf("Invalid input: %s\n", yytext); yyterminate(); }
```

```
int main() {
    yylex();
    return 0;
}
```

2. Capitalize input

Code:



Ubuntu-22.04 > home > hrishitha > CDLabExamPrepLexCodes > capi.l

```
%{
#include<stdio.h>
#include<ctype.h>
%}

%%
[a-z]+ {
    int i;
    for(i=0; yytext[i]!='\0'; i++){
        yytext[i] = toupper(yytext[i]);
    }
    printf("%s", yytext);
    yyterminate();
}

.|\n  { printf("%s", yytext); }

%%

int main() {
    yylex();
    return 0;
}
```

%{
#include<stdio.h>
#include<ctype.h>

```
%}

%%
[a-z]+ {
    int i;
    for(i=0; yytext[i]!='\0'; i++){
        yytext[i] = toupper(yytext[i]);
    }
    printf("%s", yytext);
    yyterminate();
}

.|\n  { printf("%s", yytext); }

%%

int main() {
    yylex();
    return 0;
}
```

3. Scanner without lex

Code:

ScanOutLexEasy.py

```python
import re

TOKEN_TYPES = {
    'KEYWORD': r'(int|float|char|if|else|for|while|return)',
    'FUNCTION': r'(printf|scanf)',
    'STRING':r'\"(.)*\"' ,
    'COMMENT': r'//.*',
    'PREPROCESSOR_DIRECTIVE': r'#\s*\w+',
    'IDENTIFIER': r'[a-zA-Z_][a-zA-Z0-9_]*',
    'FLOAT_CONSTANT': r'\d+\.\d+',
    'INTEGER_CONSTANT': r'\d+',
    'OPERATOR': r'[\+\-\*/<>()]',
    'UNRECOGNISED TOKEN': r'[{};]',
    'WHITESPACE': r'\s+'
}

def tokenize(code):
    tokens = []
    for line in code.split('\n'):
        p = 0
        while p < len(line):
            match = None
            for token_type, pattern in TOKEN_TYPES.items():
                regex = re.compile(pattern)
                token = regex.match(line, p)
                if token:
                    value = token.group(0)
                    if token_type != 'WHITESPACE' and (token_type, value) not in tokens:
                        tokens.append((token_type, value))
                    p = token.end()
                    break
            if not token:
                # If no match is found, increment p by 1 to avoid infinite loop
                p += 1
    return tokens

code = """
```

```
                # If no match is found, inc
                p += 1
        return tokens

code = """
#include<stdio.h>
int main() {
  int num1 = 10;
  float num2 = 3.14;
  char letter = 'a';

  for(int i = 0; i < 5; i++) {
    printf("Hello world");
  }

  return 0;
}
"""
tokens = tokenize(code)
for token in tokens:
    print(token)
```

```
import re

TOKEN_TYPES = {
    'KEYWORD': r'(int|float|char|if|else|for|while|return)',
    'FUNCTION': r'(printf|scanf)',
    'STRING':r'\"(.)*\"' ,
    'COMMENT': r'//.*',
    'PREPROCESSOR_DIRECTIVE': r'#\s*\w+',
    'IDENTIFIER': r'[a-zA-Z_][a-zA-Z0-9_]*',
    'FLOAT_CONSTANT': r'\d+\.\d+',
    'INTEGER_CONSTANT': r'\d+',
    'OPERATOR': r'[\+\-\*/<>()]',
    'UNRECOGNISED TOKEN': r'[{};]',
    'WHITESPACE': r'\s+'
```

```
}

def tokenize(code):
    tokens = []
    for line in code.split('\n'):
        p = 0
        while p < len(line):
            match = None
            for token_type, pattern in TOKEN_TYPES.items():
                regex = re.compile(pattern)
                token = regex.match(line, p)
                if token:
                    value = token.group(0)
                    if token_type != 'WHITESPACE' and (token_type, value)
not in tokens:
                        tokens.append((token_type, value))
                    p = token.end()
                    break
            if not token:
                # If no match is found, increment p by 1 to avoid infinite
loop
                p += 1
    return tokens

code = """
#include<stdio.h>
int main() {
  int num1 = 10;
  float num2 = 3.14;
  char letter = 'a';

  for(int i = 0; i < 5; i++) {
    printf("Hello world");
  }

  return 0;
}
"""
tokens = tokenize(code)
for token in tokens:
```

```
print(token)
```

Week 3 codes:

1. **Character count**

ntu-22.04 > home > hrishitha > CDLabExamPrepLexCodes > charno.l

```
%{
#include <stdio.h>
int char_count = 0;
int space_count = 0;
int line_count = 0;
int tab_count = 0;

%}

%%

[a-zA-Z0-9] {char_count++;}
[\t] {tab_count++;}
[ ] {space_count++;}
\n {line_count++;}

%%

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return 1;
    }
    FILE *file = fopen(argv[1], "r");
    if (file == NULL) {
        fprintf(stderr, "Error: Could not open file %s\n", argv[1]);
        return 1;
    }
    yyin = file;
    yylex();
    printf("Characters: %d\n", char_count - line_count);
    printf("Spaces: %d\n", space_count);
    printf("Lines: %d\n", line_count);
    printf("Tabs: %d\n", tab_count);
    fclose(file);
    return 0;
}
```

```
%{
#include <stdio.h>
int char_count = 0;
int space_count = 0;
int line_count = 0;
int tab_count = 0;

%}

%%

[a-zA-Z0-9] {char_count++;}
[\t] {tab_count++;}
[ ] {space_count++;}
\n {line_count++;}

%%

int main(int argc, char *argv[]) {
   if (argc != 2) {
      fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
      return 1;
   }
   FILE *file = fopen(argv[1], "r");
   if (file == NULL) {
      fprintf(stderr, "Error: Could not open file %s\n", argv[1]);
      return 1;
   }
   yyin = file;
   yylex();
   printf("Characters: %d\n", char_count - line_count);
   printf("Spaces: %d\n", space_count);
   printf("Lines: %d\n", line_count);
   printf("Tabs: %d\n", tab_count);
   fclose(file);
   return 0;
}
```

2. **Character count c program**

ntu-22.04 > home > hrishitha > CDLabExamPrepLexCodes > C charno.c

```c
#include <stdio.h>
#include<ctype.h>
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return 1;
    }
    FILE *file = fopen(argv[1], "r");
    if (file == NULL) {
        fprintf(stderr, "Error: Could not open file %s\n", argv[1]);
        return 1;
    }
    int char_count = 0;
    int space_count = 0;
    int line_count = 0;
    int tab_count = 0;
    int ch;
    char special_characters[] = "!@#$%^&*()-_=+[]{}|;:',.<>?";

    while ((ch = fgetc(file)) != EOF) {
        if (ch == ' ') space_count++;
        else if (ch == '\n') line_count++;
        else if (ch == '\t') tab_count++;
        else {
            if (isalnum(ch)) char_count++;
            if (strchr(special_characters, c) != NULL) count++;
    }

    fclose(file);

    printf("Characters: %d\n", char_count);
    printf("Spaces: %d\n", space_count);
    printf("Lines: %d\n", line_count);
    printf("Tabs: %d\n", tab_count);

    return 0;
}
```

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

```c
#include<ctype.h>
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return 1;
    }
    FILE *file = fopen(argv[1], "r");
    if (file == NULL) {
        fprintf(stderr, "Error: Could not open file %s\n", argv[1]);
        return 1;
    }
    int char_count = 0;
    int space_count = 0;
    int line_count = 0;
    int tab_count = 0;
    int ch;
    char special_characters[] = "!@#$%^&*()-_=+[]{}|;:',.<>?";

    while ((ch = fgetc(file)) != EOF) {
        if (ch == ' ') space_count++;
        else if (ch == '\n') line_count++;
        else if (ch == '\t') tab_count++;
        else {
            if (isalnum(ch)) char_count++;
            if (strchr(special_characters, c) != NULL) count++;
    }

    fclose(file);

    printf("Characters: %d\n", char_count);
    printf("Spaces: %d\n", space_count);
    printf("Lines: %d\n", line_count);
    printf("Tabs: %d\n", tab_count);

    return 0;
}
}
```

3. **Tokenization by dfa**

```
class DFA:
    def __init__(self):
        self.states = {'q0', 'q1'}
        self.accept_states = {'q1'}
        self.transition = {
            'q0': {'letter': 'q1'},
            'q1': {'letter': 'q1', 'digit': 'q1'}
        }
        self.current_state = 'q0'

    def transition_function(self, symbol):
        if symbol.isalpha():
            return 'letter'
        elif symbol.isdigit():
            return 'digit'
        else:
            return None

    def is_accept_state(self):
        return self.current_state in self.accept_states

    def reset(self):
        self.current_state = 'q0'

    def tokenize(self, input_string):
        tokens = []
        current_token = ''
        for symbol in input_string:
            transition_key = self.transition_function(symbol)
            if transition_key is None:
                if current_token:
                    tokens.append(current_token)
                    current_token = ''
            else:
                next_state =
self.transition[self.current_state][transition_key]
                self.current_state = next_state
                current_token += symbol
```

```
        if current_token:
            tokens.append(current_token)
        return tokens


def main():
    dfa = DFA()
    input_string = "var1 = 10 + var2"
    tokens = dfa.tokenize(input_string)
    print("Input String:", input_string)
    print("Tokens:", tokens)


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

4. **Scanner using lex**

```
%{
#include <stdio.h>
%}

%option noyywrap
/* Regular expressions for tokens */
DIGIT [0-9]
LETTER [a-zA-Z]
ID {LETTER}({LETTER}|{DIGIT})*
INT_CONST {DIGIT}+
FLOAT_CONST {DIGIT}+"."{DIGIT}+
WS [ \t\n\r]+
COMMENT ("//"(.)*)
%%
\".*\" { printf("%s is a string\n", yytext); }
\'.*\' { printf("%s is a string\n", yytext); }
"for"|"if"|"else"|"while"|"int"|"char"|"float"|"return" { printf("%s is a keyword\n", yytext); }
"printf"|"scanf" {printf("%s is a function\n",yytext);}
{ID} { printf("Identifier: %s\n", yytext); }
{INT_CONST} { printf("Integer Constant: %s\n", yytext); }
{FLOAT_CONST} { printf("Float Constant: %s\n", yytext); }
"+"|"-"|"*"|"/"|"="|"("|")"|"<"|">" { printf("Operator: %s\n",yytext); }
{WS} ; /* Ignore whitespace */
{COMMENT} ; /* Ignore comments */
"#"([^\n])*\n { printf("Preprocessor Directive: %s\n", yytext); }
. { printf("Unrecognized token: %s\n", yytext); }
```

```
%%
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s input_file\n", argv[0]);
        return 1;
    }
    FILE *file = fopen(argv[1], "r");
    if (!file) {
        perror("Error opening file");
        return 1;
    }
    yyin = file;
    yylex();
    fclose(file);
    return 0;
}
```

Week 4 codes:

1. **Calculator**

```
%{
#include <stdio.h>
#include <stdlib.h>

int op = 0;
float a, b;

void digi();
%}

dig [0-9]+|([0-9]*)"."([0-9]+)
add "+"
sub "-"
mul "*"
div "/"
pow "^"
ln \n

%%

{dig} { digi(); }
{add} { op = 1; }
{sub} { op = 2; }
{mul} { op = 3; }
{div} { op = 4; }
{pow} { op = 5; }
```

```
{ln} { printf("\n The Answer : %f\n\n", a); }

%%

void digi()
{
        if (op == 0)
        a = atof(yytext);
        else
        {
        b = atof(yytext);
        switch (op)
        {
        case 1:
        a = a + b;
        break;
        case 2:
        a = a - b;
        break;
        case 3:
        a = a * b;
        break;
        case 4:
        a = a / b;
        break;
        case 5:
        for (int i = 1; i < b; i++)
        a *= a;
        break;
        }
        op = 0;
        }
}

int main()
{
        yylex();
        return 0;
}

int yywrap()
{
        return 1;
}
```

2. **Count of printf scanf statements**

```
%{
```

```
int printf_count = 0;
int scanf_count = 0;
%}

%%
"printf"{ printf_count++; }
"scanf"           { scanf_count++; }
.         ;
%%

int yywrap() {
        return 1;
}

int main(int argc, char *argv[]) {
        if (argc != 2) {
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
        }

        FILE *fp = fopen(argv[1], "r");
        if (fp == NULL) {
        printf("Error opening file.\n");
        return 1;
        }

        yyin = fp;
        yylex();

        printf("Number of printf statements: %d\n", printf_count);
        printf("Number of scanf statements: %d\n", scanf_count);

        fclose(fp);
        return 0;
}
```

3. No of identifiers in file

```
%{
#include<stdio.h>
#include<string.h>
#define MAX_IDENTIFIERS 100
int count = 0;
char identifiers[MAX_IDENTIFIERS][100];
%}
```

```
letter [a-zA-Z]
digit [0-9]
id ({letter}({letter}|{digit})*)

%%
\".*\" { printf("%s is a string\n", yytext); }
\'.*\' { printf("%s is a string\n", yytext); }
^#.* { printf("%s is a preprocessor directive\n", yytext); }
"for"|"if"|"else"|"while"|"int"|"char"|"float"|"return" { printf("%s is a keyword\n", yytext); }
"printf"|"scanf" {printf("%s is a function\n",yytext);}
{id} {
   int is_repeat = 0;
   for (int j = 0; j < count; j++) {
      if (strcmp(identifiers[j], yytext) == 0) {
         is_repeat = 1;
         break;
      }
   }
   if (!is_repeat) {
      printf("%s is an identifier\n", yytext);
      strcpy(identifiers[count], yytext);
      count++;
   }
}
. ;

%%
int yywrap(){
   return 1;
}
int main()
{
   FILE *fp;
   char file[100];
   printf("\nEnter the filename: ");
   scanf("%s", file);
   fp = fopen(file, "r");
   if (fp == NULL) {
      printf("File not found\n");
      return 1;
   }
   yyin = fp;
   yylex();
   printf("Total unique identifiers are: %d\n", count);
   fclose(fp);
   return 0;
}
```

4. **First and follow**

```
class First_Follow():
    def __init__(self, grammar):
        self.grammar = grammar
        self.non_terminals = grammar.keys()
        print(self.non_terminals)
        self.start = list(self.non_terminals)[0]
        self.rules = [(head, body) for head, bodies in grammar.items() for
body in bodies]
    def compute_first(self, variable):
        first = set()
        productions = [rule[1] for rule in self.rules if rule[0] ==
variable]
        for production in productions:
            if not production[0].isupper():
                first.add(production[0])
            else:
                for x in production:
                    first |= self.compute_first(x)
                    if "@" not in first:
                        break
        return first
    def compute_follow(self, variable):
        follow = set()
        if variable == self.start:
            follow.add('$')
        for rule in self.rules:
            for j, char in enumerate(rule[1]):
                if char == variable:
                    while j < len(rule[1]) - 1:
                        if not rule[1][j + 1].isupper():
                            follow.add(rule[1][j + 1])
                            break
                        else:
                            follow |= self.compute_first(rule[1][j + 1])
                            if '@' not in self.compute_first(rule[1][j +
1]):
                                break
                        j += 1
```

```
                    else:
                        if rule[0] != variable:
                            follow |= self.compute_follow(rule[0])
        follow.discard('@')
        return follow
    def print_sets(self):
        print("First Sets:")
        for non_terminal in self.non_terminals:
            print(f"{non_terminal}: {self.compute_first(non_terminal)}")
        print("\nFollow Sets:")
        for non_terminal in self.non_terminals:
            print(f"{non_terminal}: {self.compute_follow(non_terminal)}")
# ε
def main():
    # example_grammar = {
    # 'E': ['TZ'],
    # 'Z': ['+TZ', '@'],
    # 'T': ['FY'],
    # 'Y': ['*FY', '@'],
    # 'F': ['(E)', 'i'],
    # }
    example_grammar = {
        'S' : ['CC'],
        'C' : ['cC' , 'd']
    }
    ff = First_Follow(example_grammar)
    print("epsilon is printed as @")
    ff.print_sets()
if __name__ == '__main__':
    main()
```

Week 5 codes:

1. **LI parsing table**

```
class First_Follow():
    def __init__(self, grammar):
        self.grammar = grammar
        self.non_terminals = grammar.keys()
        print(self.non_terminals)
        self.start = list(self.non_terminals)[0]
        self.rules = [(head, body) for head, bodies in grammar.items() for
body in bodies]

    def compute_first(self, variable):
        first = set()

        productions = [rule[1] for rule in self.rules if rule[0] ==
variable]

        for production in productions:
            if not production[0].isupper():
                first.add(production[0])
            else:
                first |= self.compute_first(production[0])


        return first

    def compute_follow(self, variable):

        follow = set()

        if variable == self.start:
            follow.add('$')

        for rule in self.rules:
            for j, char in enumerate(rule[1]):
                if char == variable:
                    while j < len(rule[1]) - 1:
                        if not rule[1][j + 1].isupper():
```

```
                            follow.add(rule[1][j + 1])
                            break
                        else:
                            follow |= self.compute_first(rule[1][j + 1])
                            if '@' not in self.compute_first(rule[1][j +
1]):
                                break
                        j += 1
                    else:
                        if rule[0] != variable:
                            follow |= self.compute_follow(rule[0])
        follow.discard('@')
        return follow

    def print_sets(self):
        print("First Sets:")
        for non_terminal in self.non_terminals:
            print(f"{non_terminal}: {self.compute_first(non_terminal)}")

        print("\nFollow Sets:")
        for non_terminal in self.non_terminals:
            print(f"{non_terminal}: {self.compute_follow(non_terminal)}")

    def computeFirstOneRHS(self, variable):
        first = set()

        if not variable[0].isupper():
            first.add(variable[0])
        else:
            for x in variable:
                first |= self.compute_first(x)
                if '@' not in first:
                    break

        return first

    def compute_parsing_table(self):
        print('\nParsing Table')
        table = {}
```

```
        for rule in self.rules:
            rule1, rule2 = rule
            first = list(self.computeFirstOneRHS(rule2))
            if '@' in first:
                first.extend(self.compute_follow(rule1))
                while '@' in first:
                    first.remove('@')

            for terminal in first:
                key = (rule1, terminal)
                if key in table:
                    table[key].append(rule2)
                else:
                    table[key] = [rule2]

        for key, value in table.items():
            print(f'{key} : {value}')

        return table


# @
def main():
    example_grammar = {
    'E': ['TA'],
    'A': ['+TA', '@'],
    'T': ['FB'],
    'B': ['*FB', '@'],
    'F': ['(E)', 'i']
    }

    ff = First_Follow(example_grammar)

    ff.print_sets()
    ans = ff.compute_parsing_table()
    print(ans)

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