https://github.com/iuliaaai/LFTC

```
class Grammar:

    def __init__(self, N, E, P, S):
        self.N = N
        self.E = E
        self.P = P
        self.S = S

    @staticmethod
    def parseLine(line):
        symbols = []
        for value in line.strip().split('=', 1)[1].strip().split(' '):
            symbols.append(value.strip())
        return symbols

    @staticmethod
    def parseProduction(rules):
        prodDict = {}

        for rule in rules:
            lhs, rhs = rule.split('->')
            lhs = lhs.strip()
            rhs = [value.strip() for value in rhs.split('|')]

            for value in rhs:
                if lhs in prodDict.keys():
                    prodDict[lhs].append(value)
                else:
                    prodDict[lhs] = [value]

        return prodDict

    @staticmethod
    def readFile(fileName):
        with open(fileName, 'r', encoding='utf-8') as file:
            N = Grammar.parseLine(file.readline())
            E = Grammar.parseLine(file.readline())
            S = file.readline().split('=')[1].strip()
            file.readline()
            P = Grammar.parseProduction([line.strip() for line in file])

            if not Grammar.validate(N, E, P, S):
               raise Exception("Input file doesn't have a right format.")

            return Grammar(N, E, P, S)

    @staticmethod
    def validate(N, E, P, S):
        if S not in N:
            return False
        for key in P.keys():
            if key not in N and key not in E:
                return False
            for production in P[key]:
                for elem in production.strip().split():
                    if elem not in N and elem not in E and elem != 'E':
                        return False
```

```
        return True

    def isNonTerminal(self, value):
        return value in self.N

    def getNonTerminals(self):
        return ', '.join(self.N)

    def getTerminals(self):
        return ', '.join(self.E)

    def getProductionsFor(self, nonTerminal):
        if not self.isNonTerminal(nonTerminal):
            raise Exception('Can only show productions for non-terminals')
        for key in self.P.keys():
            if key == nonTerminal:
                return self.P[key]

    def getAllProductions(self):
        return ', '.join([' -> '.join([str(prod), str(self.P[prod])]) for
prod in self.P])

    def isCFG(self):
        for key in self.P.keys():
            if key not in self.N:
                    return False
        return True

    def __str__(self):
        return 'N = { ' + self.getNonTerminals() + ' }\n' \
               + 'E = { ' + self.getTerminals() + ' }\n' \
               + 'P = { ' + self.getAllProductions() + ' }\n' \
               + 'S = ' + str(self.S) + '\n'
```

g1.txt

```
N = S A B C
E = 0 1
S = S
P =
  S -> 0 B | 1 A
  A -> 0 | 0 S | 1 A A
  B -> 1 | 1 S | 0 B B
```

g2.txt

```
N = program tempDecl declList declaration variableDeclaration constDeclaration type1
arrayDecl type stmtList stmt simplStmt assignStmt expression term factor ioStmt stringExp
structStmt ifStmt tempElifStmt elseStmt elifStmt whileStmt condition relation
E = + - * / < <= = >= > ?: == % ! != @ [ ] { } # “ ” ‘ ’ ; ( ) func Int String Bool Char if elif else
let var ret True False read print loop GO STOP identifier constant integer string boolean E
S = program
P =
program -> GO tempDecl STOP
tempDecl -> E | tempDecl declList | tempDecl stmtList
declList -> declaration | declaration declList
```

declaration -> variableDeclaration | constDeclaration
variableDeclaration -> var identifier @ type = expression ; | var identifier @ type ;
constDeclaration -> let identifier @ type = expression ;
type1 -> Bool | Int | Char | String
arrayDecl -> [type1]
type -> type1 | arrayDecl
stmtList -> stmt | stmt stmtList
stmt -> simplStmt | structStmt
simplStmt -> assignStmt | ioStmt
assignStmt -> identifier = expression ;
expression -> expression + term | expression - term | term | boolean
term -> term * factor | term / factor | term % factor | factor
factor -> (expression) | identifier | integer
ioStmt -> read (identifier) ; | print (stringExp) ;
stringExp -> string | identifier
structStmt -> ifStmt | whileStmt
ifStmt -> if condition { stmtList } tempElifStmt | if condition { stmtList } tempElifStmt elseStmt
tempElifStmt -> E | tempElifStmt elifStmt
elseStmt -> else { stmtList }
elifStmt -> elif condition { stmtList }
whileStmt -> loop condition { stmtList }
condition -> expression relation expression
relation -> < | <= | = | >= | > | !=