Parser - Documentation:

Language used: Python

=== Grammar ===
non_terminals: List<String> => Contains the set of nonterminal symbols
terminals: List<String> => Contains the set of terminal symbols
S: List<Pair<Pair<String, String>, String>> => Contains the set of states
productions: List<Production> => Contains the set of production rules
---
get_grammar_from_file(String): Void => Reads the data from the input file and stores it
get_productions_for_non_terminal(String): returns the set of productions for a give nonterminal string
CFG_check(): Boolean => Returns True if the grammar is Context-free, False otherwise

expand(): =>
advance(): =>
momentary_insuccess(): =>
back(): =>
another_try(): =>
success(): =>

=== Production ===
start: String => the "head" of the rule, contains the string of symbols with at least one nonterminal
rules: List<String> => the "body" of the rule, contains the strings of symbols that can result from the start

=== ParserConfing ===
grammar: Grammar => Contains the grammar used for parsing
s: String => Contains the current state of the parser
i: Integer => Contains the current position of the symbol in input sequence
alpha: List => Represents the working stack (contains the way the parse is built)
beta: List => Represents the input stack, part of the tree to be built
---
expand(): => For a nonterminal head of input stack it expands it according to the rule
advance(): => For a terminal head of input stack that equals the currenty symbol from input, i is incremented and alpha stack advances with the top of beta stack
momentary_insuccess(): => For a terminal head of input stack that is diferent from the current symbol from input, parser state is set to back state ('b')
back(): => For a terminal head of working stack, i is decremented and alpha stack advances
another_try(): => For a nonterminal head of working stack, either the program goes into error state ('e') or normal state ('q')
success(): => Sets the final state ('f') corresponding to success w ∈ L(G)

g1.txt:
S A B
a b epsilon

S
S -> epsilon | a B | b A
A -> a
A -> a A | b B
B -> b | b B
A B -> a

g2.txt:
program cmpdstmt decllist declaration type type1 arraydecl stmt stmtlist simplstmt assignstmt expression iostmt structstmt ifstmt forstmt condition RELATION IDENTIFIER "Boolean" "Integer" "String" "List" "<" type1 ">" "=" "+" "-" "*" "/" "(" ")" "[" "]" "," "read" "print" CONSTANT : { } "if" "else" "for" < <= == != >= >
program
program -> decllist cmpdstmt
decllist -> declaration | declaration decllist
declaration -> type IDENTIFIER
type -> type1 | arraydecl
type1 -> "Boolean" | "Integer" | "String"
arraydecl -> "List" "<" type1 ">"
cmpdstmt -> "BEGIN" stmtlist "END"
stmtlist -> stmt | stmt ";" stmtlist
stmt -> simplstmt | structstmt
simplstmt -> assignstmt | iostmt
assignstmt -> IDENTIFIER "=" expression
expression -> expression "+" | "-" term | term
iostmt -> "read" "(" IDENTIFIER ")" | "write" "(" IDENTIFIER | CONSTANT ")"
structstmt -> cmpdstmt | ifstmt | forstmt
ifstmt -> "if" condition ":" stmt ["else" ":" stmt]
forstmt -> "for" assignstmt condition assignstmt ":" stmt
condition -> expression RELATION expression
RELATION -> < | <= | == | != | >= | >