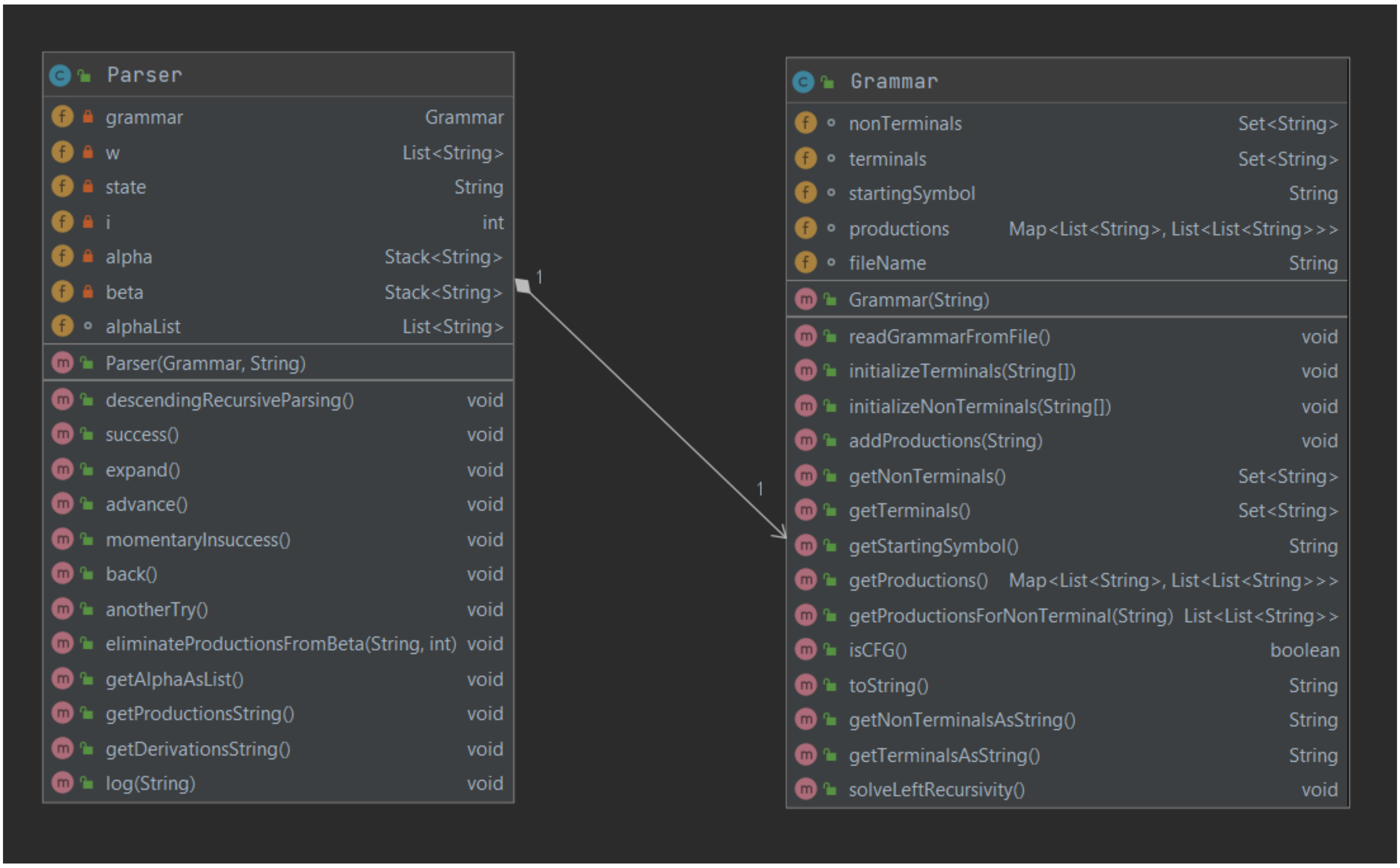
Requirement: Statement: Implement a parser algorithm (cont.) – Descendent Recursive Parser

PART 2: Deliverables

Functions corresponding to the assigned parsing strategy + appropriate tests, as detailed below:

Recursive Descendent - functions corresponding to moves (*expand*, *advance*, *momentary insuccess*, *back*, *another try*, *success*)

Analysis and Design:



Parser	
grammar	Grammar
w	List<String>
state	String
i	int
alpha	Stack<String>
beta	Stack<String>
alphaList	List<String>
Parser(Grammar, String)	
descendingRecursiveParsing()	void
success()	void
expand()	void
advance()	void
momentaryInsuccess()	void
back()	void
anotherTry()	void
eliminateProductionsFromBeta(String, int)	void
getAlphaAsList()	void
getProductionsString()	void
getDerivationsString()	void
log(String)	void

Grammar	
nonTerminals	Set<String>
terminals	Set<String>
startingSymbol	String
productions	Map<List<String>, List<List<String>>>
fileName	String
Grammar(String)	
readGrammarFromFile()	void
initializeTerminals(String[])	void
initializeNonTerminals(String[])	void
addProductions(String)	void
getNonTerminals()	Set<String>
getTerminals()	Set<String>
getStartingSymbol()	String
getProductions()	Map<List<String>, List<List<String>>>
getProductionsForNonTerminal(String)	List<List<String>>
isCFG()	boolean
toString()	String
getNonTerminalsAsString()	String
getTerminalsAsString()	String
solveLeftRecursivity()	void

The Parser is a class which uses Descendent Recursive Parsing Algorithm.

Its attributes are:

- grammar: Grammar – the grammar according to which the parsing is done
- w: List<String> – the word to be parsed, represented as a List of Strings (which represent terminal symbols)
- state: String – the current state of the parsing
- i: Int – the position of the current symbol in the input sequence w
- alpha: Stack<String> – the working stack
- beta: Stack<String> – the input stack
- alphaList: List<String> – a list containing all the symbols from the working stack, in the right order

Its methods are:

- descendingRecursiveParsing() – parses the input sequence using the descending recursive parsing; displays an appropriate message denoting if the sequence is accepted or not

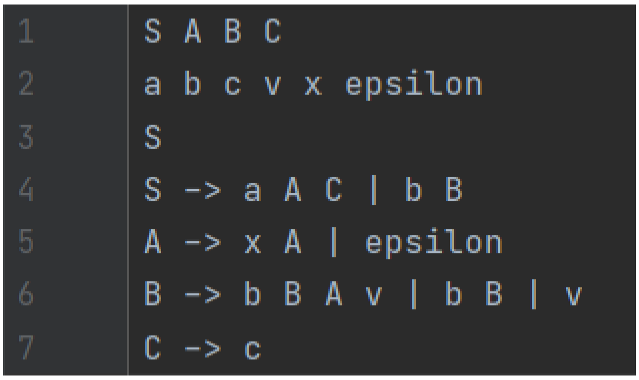
- expand()
  - Pre: head of beta is a nonterminal A; state = “q”
  - Post: A is pushed to alpha; all the symbols from the rhs of the first production of A are pushed to beta
- advance()
  - Pre: head of beta is a terminal a, equal to the current symbol from input; state = “q”
  - Post: i = i+1; a is pushed to alpha and removed from beta
- momentaryInsuccess()
  - Pre: head of beta is a terminal a, different from the current symbol from input
  - Post: state = “b”
- back()
  - Pre: head of alpha is a terminal a, state = “b”
  - Post: i = i-1, a is pushed to beta and removed from alpha
- anotherTry()
  - Pre: head of the working stack is a nonterminal A, state=”b”
  - Post: if A still has a production A -> Gamma that was not used => state = “q”; A is pushed to alpha; all the symbols from Gamma are pushed to beta // else if i=1 and A=S, state=”e”// else state = “b”; A is pushed to beta and removed from alpha
- eliminateProductionsFromBeta(String A, int i) – eliminates from beta all the symbols from the top, that were added using the production i of the nonterminal A

Implementation:

https://github.com/LaviniaGalan/FLCD/tree/master/Lab6

Testing:

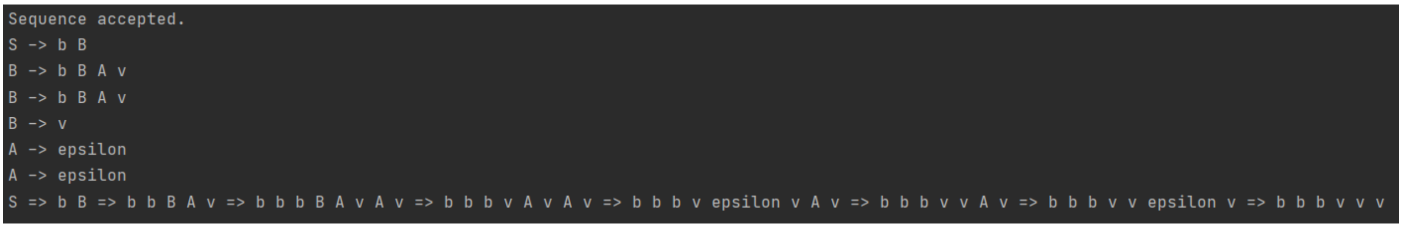
1. Grammar =



```
S A B C
a b c v x epsilon
S
S -> a A C | b B
A -> x A | epsilon
B -> b B A v | b B | v
C -> c
```

a) Input sequence = “bbbvvv”

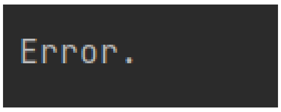
Result =



```
Sequence accepted.
S -> b B
B -> b B A v
B -> b B A v
B -> v
A -> epsilon
A -> epsilon
S => b B => b b B A v => b b b B A v A v => b b b v A v A v => b b b v epsilon v A v => b b b v v A v => b b b v v epsilon v => b b b v v v
```

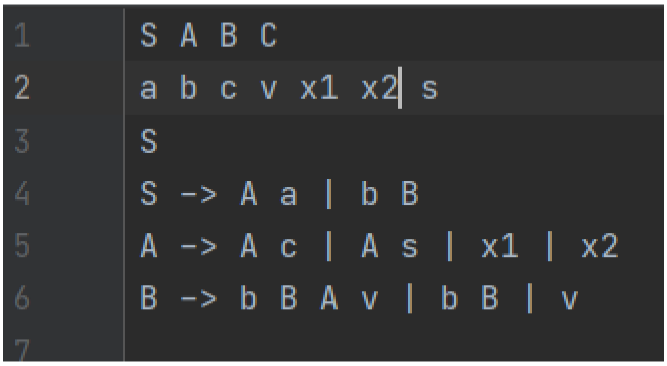
b) Input sequence = “b b b b v v v c v”

Result =



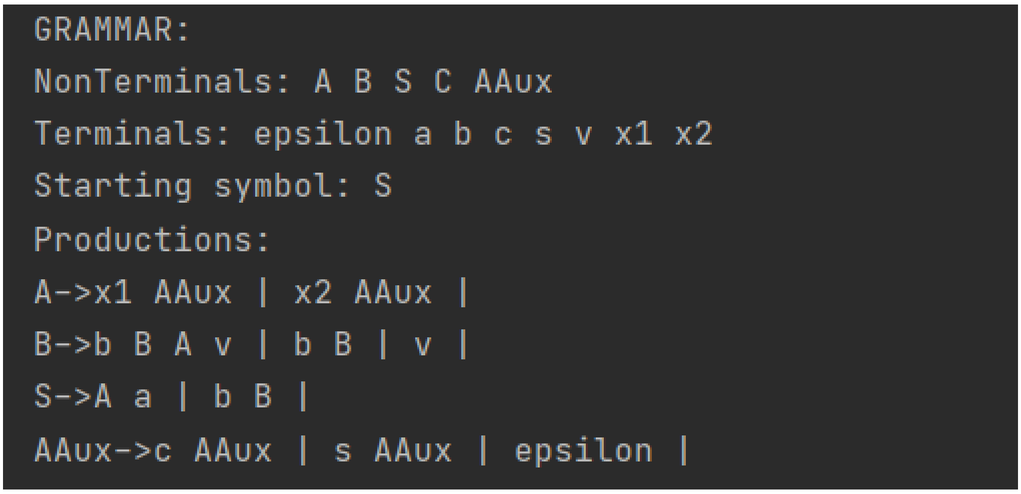
```
Error.
```

2. Grammar = (left recursive grammar)



```
S A B C
a b c v x1 x2 s
S
S -> A a | b B
A -> A c | A s | x1 | x2
B -> b B A v | b B | v
```

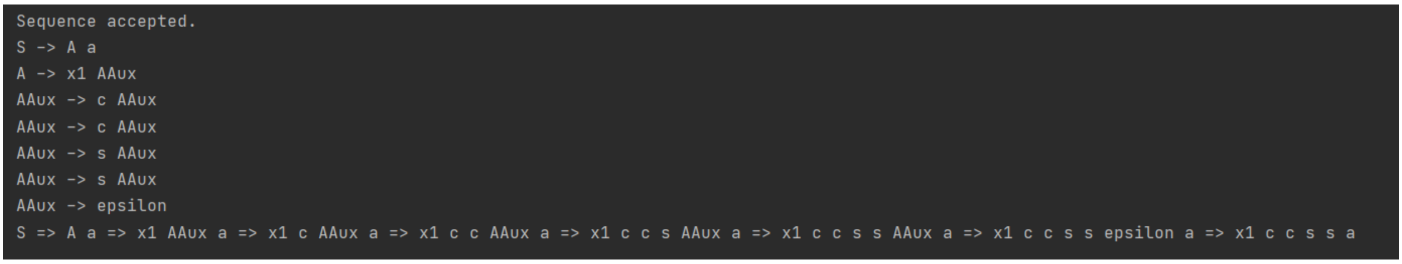
The grammar is transformed:



```
GRAMMAR:
NonTerminals: A B S C AAux
Terminals: epsilon a b c s v x1 x2
Starting symbol: S
Productions:
A->x1 AAux | x2 AAux |
B->b B A v | b B | v |
S->A a | b B |
AAux->c AAux | s AAux | epsilon |
```

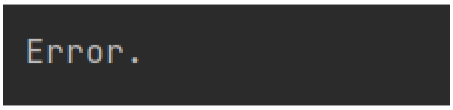
a) Input sequence = “x1 c c s s a”

Result =



```
Sequence accepted.
S -> A a
A -> x1 AAux
AAux -> c AAux
AAux -> c AAux
AAux -> s AAux
AAux -> s AAux
AAux -> epsilon
S => A a => x1 AAux a => x1 c AAux a => x1 c c AAux a => x1 c c s AAux a => x1 c c s s AAux a => x1 c c s s epsilon a => x1 c c s s a
```

b) Input sequence = “x1”



```
Error.
```