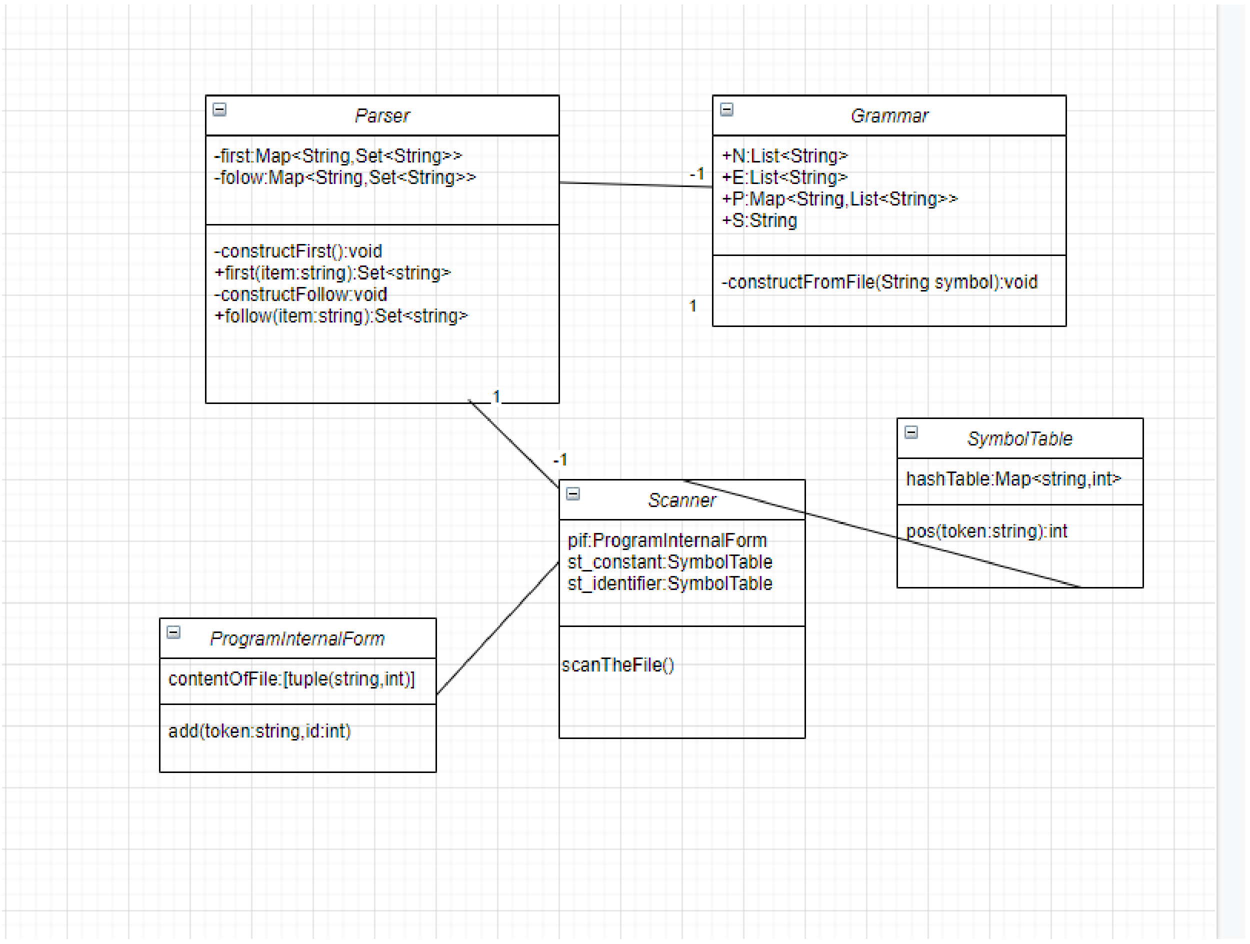
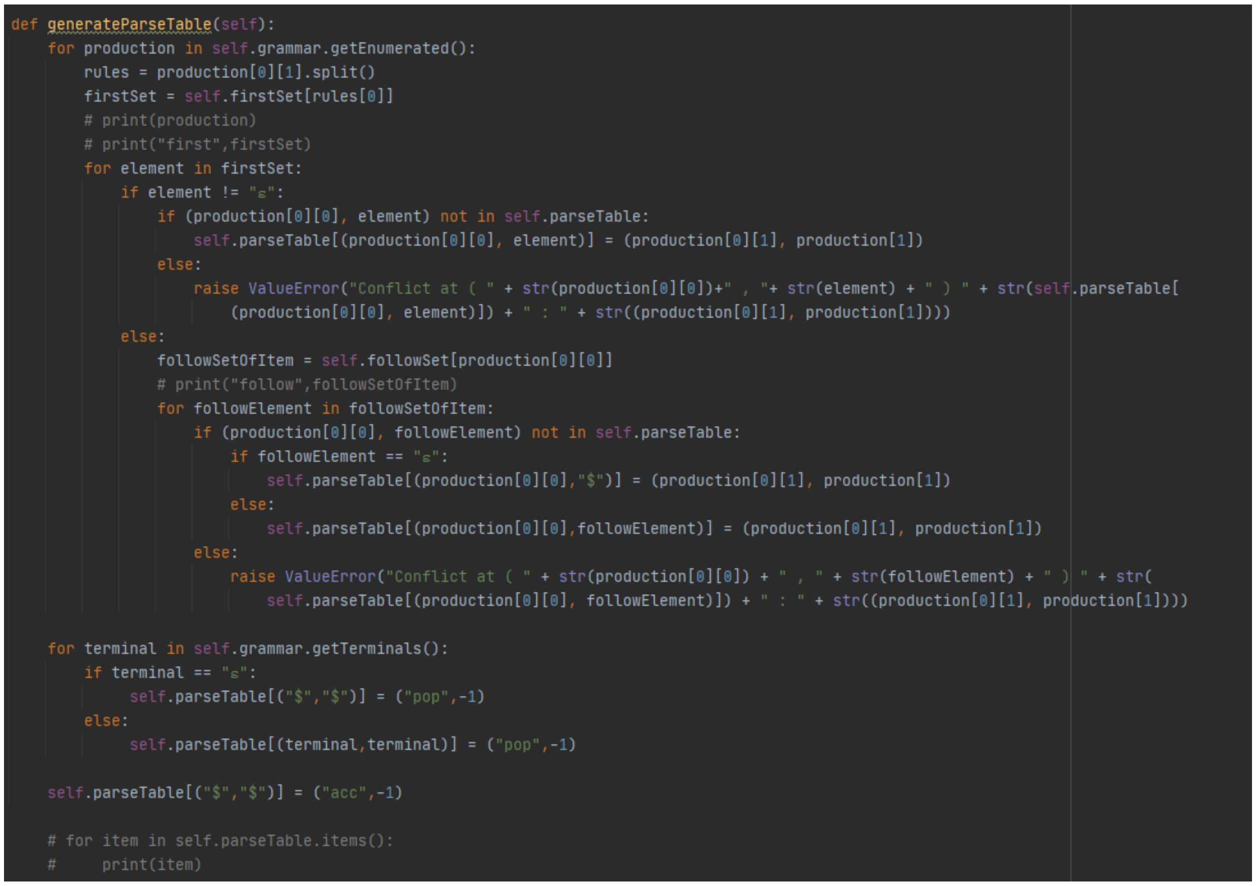
https://github.com/RavaszTamas/FormalLanguagesLaboratories/tree/main/lab5


Parser
-first:Map<String,Set<String>>
-folow:Map<String,Set<String>>
-constructFirst():void
+first(item:string):Set<string>
-constructFollow:void
+follow(item:string):Set<string>
Grammar
+N:List<String>
+E:List<String>
+P:Map<String,List<String>>
+S:String
-constructFromFile(String symbol):void
-1
1
1
-1
SymbolTable
hashTable:Map<string,int>
pos(token:string):int
Scanner
pif:ProgramInternalForm
st_constant:SymbolTable
st_identifier:SymbolTable
scanTheFile()
ProgramInternalForm
contentOfFile:[tuple(string,int)]
add(token:string,id:int)

Class diagram for the application



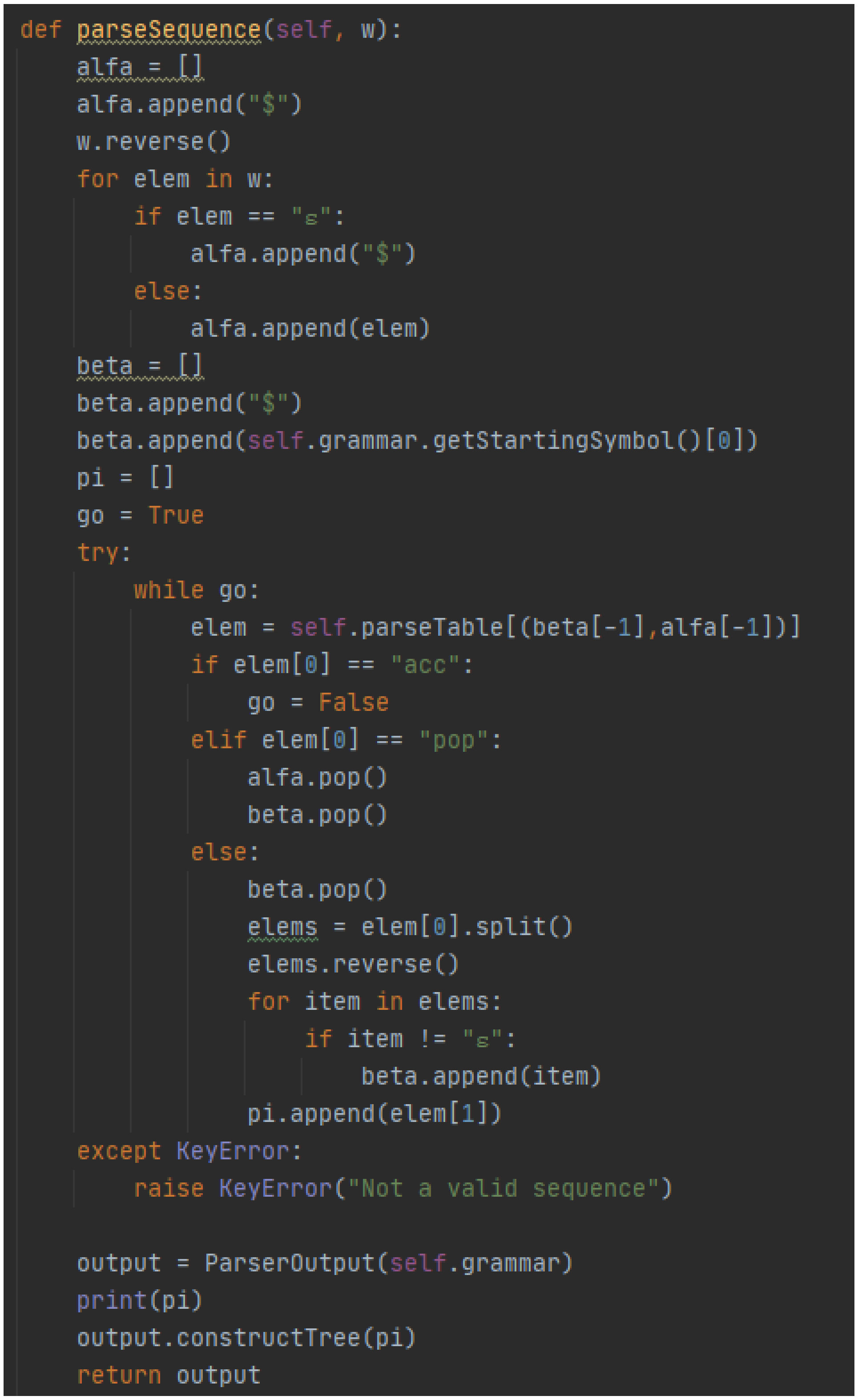
```python
def generateParseTable(self):
    for production in self.grammar.getEnumerated():
        rules = production[0][1].split()
        firstSet = self.firstSet[rules[0]]
        # print(production)
        # print("first",firstSet)
        for element in firstSet:
            if element != "ε":
                if (production[0][0], element) not in self.parseTable:
                    self.parseTable[(production[0][0], element)] = (production[0][1], production[1])
                else:
                    raise ValueError("Conflict at ( " + str(production[0][0])+" , "+ str(element) + " ) " + str(self.parseTable[
                        (production[0][0], element)]) + " : " + str((production[0][1], production[1])))
            else:
                followSetOfItem = self.followSet[production[0][0]]
                # print("follow",followSetOfItem)
                for followElement in followSetOfItem:
                    if (production[0][0], followElement) not in self.parseTable:
                        if followElement == "ε":
                            self.parseTable[(production[0][0],"$")] = (production[0][1], production[1])
                        else:
                            self.parseTable[(production[0][0],followElement)] = (production[0][1], production[1])
                    else:
                        raise ValueError("Conflict at ( " + str(production[0][0]) + " , " + str(followElement) + " ) " + str(
                            self.parseTable[(production[0][0], followElement)]) + " : " + str((production[0][1], production[1])))

    for terminal in self.grammar.getTerminals():
        if terminal == "ε":
             self.parseTable[("$","$")] = ("pop",-1)
        else:
             self.parseTable[(terminal,terminal)] = ("pop",-1)

    self.parseTable[("$","$")] = ("acc",-1)

    # for item in self.parseTable.items():
    #     print(item)
```

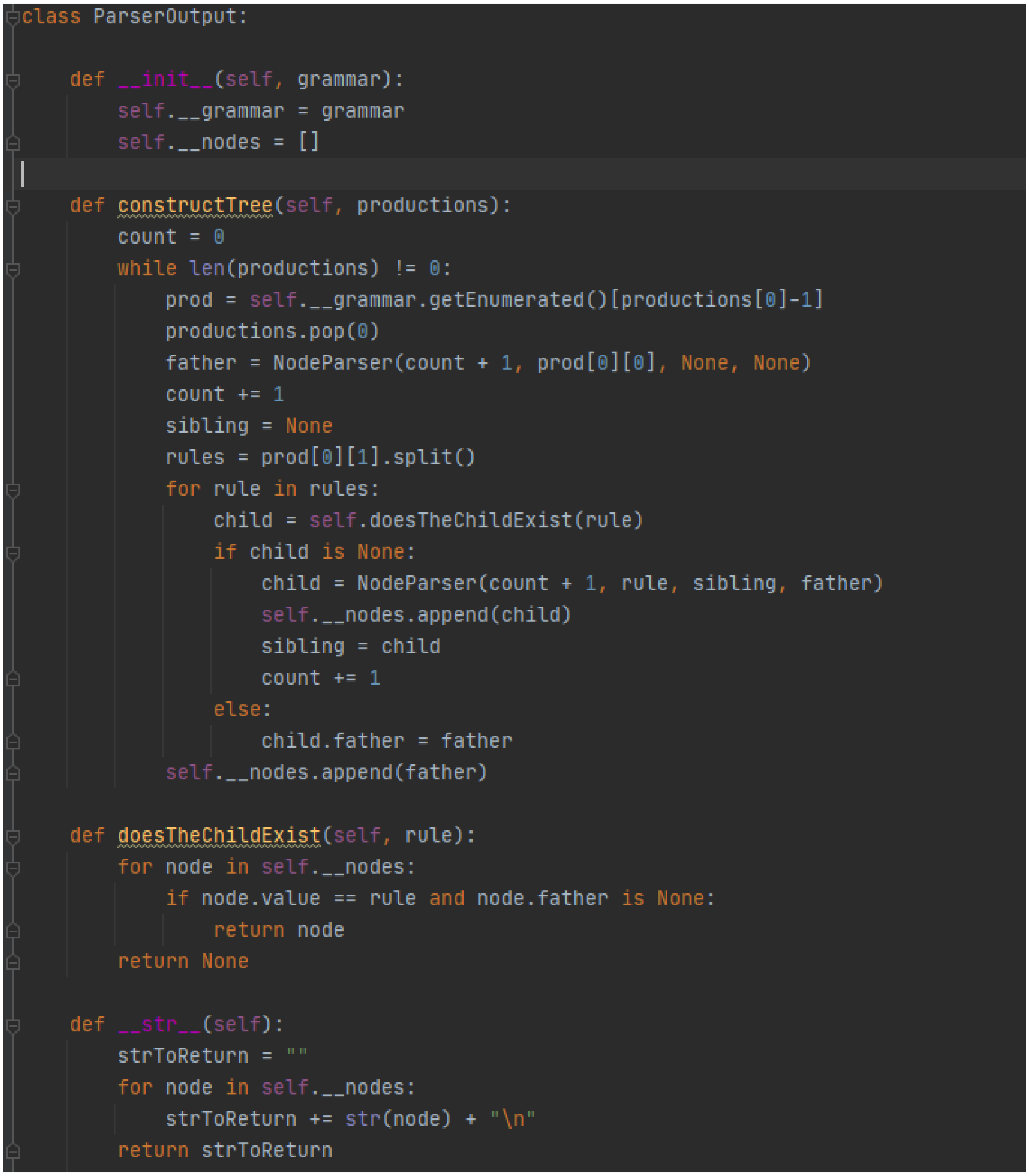
The algorithm which generates the parsing table. The table is generated when the Parser object is instantiated and it uses the follow and first sets to generate the table. It goes through all of the productions and for each production’s left hand side’s first elements are assigned to in the parsing table the right hand side of the production, with the element in the first set as another key, if epsilon appears in the first set the elements that are in the follow set similarly to elements in the first set. Conflicts appear of more than one item is located in a cell of the table in that case the grammar is not ll(1).
For the terminals and the empty token we assign pop and acc respectively



```
def parseSequence(self, w):
    alfa = []
    alfa.append("$")
    w.reverse()
    for elem in w:
        if elem == "ε":
            alfa.append("$")
        else:
            alfa.append(elem)
    beta = []
    beta.append("$")
    beta.append(self.grammar.getStartingSymbol()[0])
    pi = []
    go = True
    try:
        while go:
            elem = self.parseTable[(beta[-1],alfa[-1])]
            if elem[0] == "acc":
                go = False
            elif elem[0] == "pop":
                alfa.pop()
                beta.pop()
            else:
                beta.pop()
                elems = elem[0].split()
                elems.reverse()
                for item in elems:
                    if item != "ε":
                        beta.append(item)
                pi.append(elem[1])
    except KeyError:
        raise KeyError("Not a valid sequence")

    output = ParserOutput(self.grammar)
    print(pi)
    output.constructTree(pi)
    return output
```

A sequence is parsed using this algorithm which is the parsing algorithm for ll(1), after initializing the input and working stack for each pair of elements at the end of the working stack we get their production in the parsing table, if it is pop we pop from both of the stacks if acc the sequence is accepted, if we get a production then we pop the working stack and push the elements from the obtained production's right hand side into it. If we geta KeyError we have an invalid input sequence and an exception is raised.



```
class ParserOutput:

    def __init__(self, grammar):
        self.__grammar = grammar
        self.__nodes = []

    def constructTree(self, productions):
        count = 0
        while len(productions) != 0:
            prod = self.__grammar.getEnumerated()[productions[0]-1]
            productions.pop(0)
            father = NodeParser(count + 1, prod[0][0], None, None)
            count += 1
            sibling = None
            rules = prod[0][1].split()
            for rule in rules:
                child = self.doesTheChildExist(rule)
                if child is None:
                    child = NodeParser(count + 1, rule, sibling, father)
                    self.__nodes.append(child)
                    sibling = child
                    count += 1
                else:
                    child.father = father
            self.__nodes.append(father)

    def doesTheChildExist(self, rule):
        for node in self.__nodes:
            if node.value == rule and node.father is None:
                return node
        return None

    def __str__(self):
        strToReturn = ""
        for node in self.__nodes:
            strToReturn += str(node) + "\n"
        return strToReturn
```

The parser output is used by the parser to represent the output. In this case we have a tree with the father sibling relation.