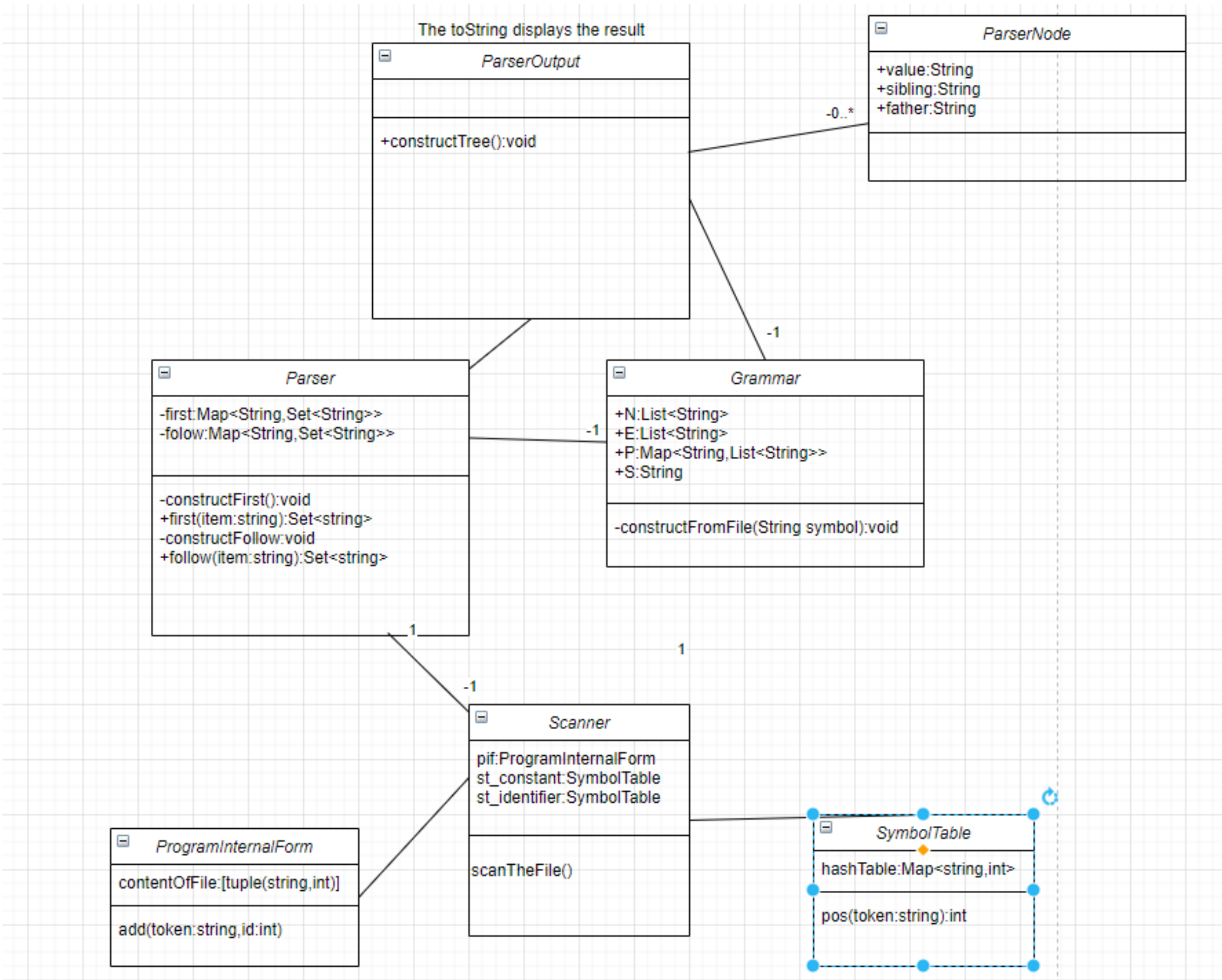
Ravasz Tamás, Andrei-Daniel Sălăgean

Parser documentation

https://github.com/RavaszTamas/FormalLanguagesLaboratories/tree/main/lab5-7

Class diagram



Class diagram for the application

```python
class Grammar:

    @staticmethod
    def processTheLine(line):
        return [value.strip().strip('"') for value in line.strip().split("=",1)[1].strip().split("|")]

    @staticmethod
    def processTheLineRule(line):
        return [value.strip().strip('"') for value in line.strip().split("=",1)[1].strip().split("\n")]

    @staticmethod
    def processRules(linesOfRules):
        result = {}
        enumerated = []
        index = 1
        for rule in linesOfRules:
            rule.strip()
            if rule == "":
                continue
            lhs, rhs = rule.split('->')
            lhs = lhs.strip()
            rhs = [value.strip() for value in rhs.split('|')]
            for value in rhs:
                if lhs not in result:
                    result[lhs] = []
                result[lhs].append(value)
                enumerated.append(((lhs,value),index))
                index +=1
        return result,enumerated


    @staticmethod
    def fromFile(fileName):
        with codecs.open(fileName,'r',encoding='utf8') as file:
            N = Grammar.processTheLine(file.readline())
            E = Grammar.processTheLine(file.readline())
            S = Grammar.processTheLine(file.readline())
            P,enumerated = Grammar.processRules(Grammar.processTheLineRule(''.join([line for line in file])))
            return Grammar(N,E,P,S,enumerated)
```

The main representation of a grammar used by our application.

```
N = "S" | "A" | "B" | "C" | "D"
E = "+" | "ε" | "*" | ")" | "(" | "a"
S = "S"
P =
S -> B A
A -> + B A | ε
B -> D C
C -> * D C | ε
D -> ( S ) | a
```

An example like this is used here where the nonterminals and terminals are in sperarate rows and the starting symbol is defined in a separate line then each production in a separate line.

```python
def constructFirst(self):
    for terminal in self.grammar.getTerminals():
        self.firstSet[terminal] = {terminal}

    for nonTerminal in self.grammar.getNonTerminals():
        initial = set()
        for production in self.grammar.getProductsForNonTerminal(nonTerminal):
            productionElements = production.split()
            if productionElements[0] in self.grammar.getTerminals():
                initial.add(productionElements[0])
        self.firstSet[nonTerminal] = initial

    modified = True
    while modified:
        modified = False
        for nonTerminal in self.grammar.getNonTerminals():
            for production in self.grammar.getProductsForNonTerminal(nonTerminal):
                productionElements = production.split()
                go = True
                for item in productionElements:
                    if len(self.firstSet[item]) == 0:
                        go = False
                        break
                if go:
                    concatResult = copy.deepcopy(self.firstSet[productionElements[0]])
                    if 'ε' in concatResult:
                        concatResult.remove('ε')

                    for i in range(len(productionElements)):
                            concatResult = self.generateConcatenationOfLengthOne(concatResult,productionElements[i])

                    if not concatResult.issubset(self.firstSet[nonTerminal]):
                        self.firstSet[nonTerminal] = self.firstSet[nonTerminal].union(self.firstSet[productionElements[0]])
                        modified = True
```

FIRST algorithm. It follows what was described in the professor’s description only that not all sets are checked but the a boolean value is observed to check wheter any changes occur.
At the initalization the all terminals get the first set as themselves then all non-terminals will get the the first terminal that appears as the first item in each production where the non-terminal is at the left hand side. After that for each non terminal if the for all items on the right hand side we have a non-empty first set then we will generate in the concat result is the result of the concatenation of length one.

```python
def generateConcatenationOfLengthOne(self,firstSet,secondSet):
    resultSet = set()
    for itemOne in firstSet:
        for itemTwo in secondSet:

            concatItem = (itemOne,itemTwo)
            if concatItem[0] != 'ε':
                resultSet.add(concatItem[0])
            else:
                resultSet.add(concatItem[1])
    return resultSet
```

The operation looks like this. Where if we have double epsilon we preserve it otherwise the first or second item will remain in the set depending on if the first item is epsilon or not.

```python
def constructFollow(self):
    for nonTerminal in self.grammar.getNonTerminals():
        if nonTerminal == self.grammar.getStartingSymbol()[0]:
            self.followSet[nonTerminal] = set("ε")
        else:
            self.followSet[nonTerminal] = set()

    modified = True
    while modified:
        modified = False
        for nonTerminal in self.grammar.getNonTerminals():
            for lhs, rhs in self.grammar.getProductions().items():
                for production in rhs:
                    elements = production.split()
                    if nonTerminal in elements:
                        index = elements.index(nonTerminal)
                        temp = elements[index + 1:]
                        if len(temp) == 0:
                            if not self.followSet[lhs].issubset(self.followSet[nonTerminal]):
                                self.followSet[nonTerminal] = self.followSet[nonTerminal].union(self.followSet[lhs])
                                modified = True
                        else:
                            if "ε" in self.firstSet[temp[0]]:
                                temporarySet = copy.deepcopy(self.firstSet[temp[0]])
                                temporarySet.remove('ε')
                                temporarySet = temporarySet.union(self.followSet[lhs])

                                if not temporarySet.issubset(self.followSet[nonTerminal]):
                                    self.followSet[nonTerminal] = self.followSet[nonTerminal].union(temporarySet)
                                    modified = True
                            else:
                                if not self.firstSet[temp[0]].issubset(self.followSet[nonTerminal]):
                                    self.followSet[nonTerminal] = self.followSet[nonTerminal].union(self.firstSet[temp[0]])
                                    modified = True
    print(self.followSet)
```

For the follow set creation it is initialized as depicted in the course with the starting symbol having epsilon then the loops represent the iteration and the modified behaves similarly as in the first set. When finding the item on the right hand side we get what is following it and depending on if the first contains epsilon (or it is an empty list which is the same) we will add the first of the that part or the follow of the left hand side

```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

```python
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.

```python
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.

```python
class NodeParser:
    """
    A structure representing a node inside our binary search tree
    the value is the token,
    the code is the position in the symbol table
    right and left are possible child nodes
    """

    def __init__(self, id, value, sibling, father):
        self.id = id
        self.value = value
        self.sibling = sibling
        self.father = father

    def __str__(self):
        strToReturn = "Node " + str(self.id) + ": " + str(self.value) + " Father:"
        if self.father is None:
            strToReturn += " None "
        else:
            strToReturn += str(self.father.id)

        strToReturn += " Sibling: "

        if self.sibling is None:
            strToReturn += " None "
        else:
            strToReturn += str(self.sibling.id)

        return strToReturn
```

The node for the parser output, with a father and a sibling and the current value, each of them is a string.

Documentation for the Scanner and the finite automata can be found in the previous lab documentations.