Final lab parsing (integrate Lab 5, Lab 6, Lab 7)

Micle Sergiu

Daniel-Vladut Andreica

https://github.com/sergiu9999/Lab5-SergiuDaniel

1. One of the following parsing methods will be chosen (assigned by teaching staff):

    1.a. recursive descendent

    1.b. ll(1)

    1.c. lr(0)

2. The representation of the parsing tree (output) will be (decided by the team):

    2.a. productions string (max grade = 8.5)

    2.b. derivations string (max grade = 9)

    2.c. table (using father and sibling relation) (max grade = 10)

PART 1: Deliverables

Class Grammar (required operations: read a grammar from file, print set of nonterminals, set of terminals, set of productions, productions for a given nonterminal, CFG check)

Input files: g1.txt (simple grammar from course/seminar), g2.txt (grammar of the minilanguage - syntax rules from Lab 1b)

Grammer Class:

```
class Grammar:
    def __init__(self):
        self.starting_symbol = None
        self.non_terminals = []
        self.terminals = []
        self.productions = defaultdict(list)
        self.is_cfg = True

    def get_starting_symbol(self):
```

```
        return self.starting_symbol

    def get_non_terminals(self):
        return self.non_terminals

    def get_terminals(self):
        return self.terminals

    def get_productions(self):
        return self.productions

    def get_is_cfg(self):
        return self.is_cfg

    def read_grammar(self, file_name):
        f = open(file_name, "r")

        def current_line():
            return f.readline().strip()

        self.starting_symbol = current_line()
        self.terminals = current_line().split(',')
        self.non_terminals = current_line().split(',')
        if self.starting_symbol not in self.non_terminals:
            raise GrammarException("Starting symbol not in nonterminals")
        for raw_line in f.readlines():
            line = raw_line.strip()
            symbol_groups = line.split('->')
            if len(symbol_groups) != 2:
                raise GrammarException("Invalid production rule")
            left_side, right_side = symbol_groups
            left_symbols = left_side.strip().split(' ')
            if len(left_symbols) > 1:
                self.is_cfg = False
                production_left = tuple(left_symbols)
            else:
                production_left = left_symbols[0]
            right_symbols = right_side.strip().split(' ')
            self.productions[production_left].append(right_symbols)
```

Output File

```
S
a,b,c
S
S -> a S b S
```

```
S -> a S
S -> c
```

Lab 6

Statement: Implement a parser algorithm (cont.) - as assigned by the coordinating teacher, at the previous lab

Remark: Lab work evaluation is not done per project/team, but per team member (reflecting the individual contribution to what has been delivered). Please make sure that you split the tasks in a balanced way among team members! In case you decide to do pair programming, each team member is supposed to know all the details of what has been implemented!

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)

LL(1) - functions FIRST, FOLLOW

LR(0) - functions Closure, GoTo, CanonicalCollection

```
class State(Enum):
    NORMAL = "q"
    BACK = "b"
    FINAL = "f"
    ERROR = "e"


class RecursiveDescendent:
    def __init__(self, grammar: Grammar, sequence, logs=False) -> None:
        self.state = State.NORMAL
        self.grammar = grammar
        self.sequence = sequence
        self.position = 0
        self.working_stack = []
        self.input_stack = [grammar.get_starting_symbol()]
        self.logs = logs

    def expand(self):
        if self.logs:
```

```
            print("expand")
        non_terminal = self.input_stack[0]
        production = self.grammar.productions[non_terminal][0]
        self.working_stack.append([non_terminal, 0])
        self.input_stack = production + self.input_stack[1:]

    def advance(self):
        if self.logs:
            print("advance")
        self.position += 1
        self.working_stack.append(self.input_stack[0])
        self.input_stack = self.input_stack[1:]

    def momentary_insuccess(self):
        if self.logs:
            print("momentary insuccess")
        self.state = State.BACK

    def back(self):
        if self.logs:
            print("back")
        self.position -= 1
        self.input_stack = [self.working_stack[-1]] + self.input_stack
        self.working_stack.pop()

    def another_try(self):
        if self.logs:
            print("another_try")
        non_terminal, production_number = self.working_stack[-1]

        if production_number < len(self.grammar.productions[non_terminal]) - 1:
            pop_length =
len(self.grammar.productions[non_terminal][production_number])
            self.input_stack = (
                    self.grammar.productions[non_terminal][production_number +
1] +
                    self.input_stack[pop_length:]
            )
            self.working_stack[-1][1] += 1
            self.state = State.NORMAL

        else:
            if self.position == 0 and non_terminal ==
self.grammar.get_starting_symbol():
                self.state = State.ERROR
                return

            self.working_stack.pop()
            pop_length =
len(self.grammar.productions[non_terminal][production_number])
            self.input_stack = [non_terminal] + self.input_stack[pop_length:]
```

```
    def success(self):
        if self.logs:
            print("success")
        self.state = State.FINAL

    def get_production_string(self):
        production_string = ""
        for elem in self.working_stack:
            if elem in self.grammar.get_terminals():
                continue
            production_string += f"{elem[0]}{elem[1]} "
        return production_string

    def get_parsing_tree(self):
        production_tree = LabeledTree()
        for node_index, elem in enumerate(self.working_stack):
            if elem in self.grammar.get_terminals():
                production_tree.add_label(node_index, elem)
                continue
            production_tree.add_label(node_index, elem[0])
            children_labels = self.grammar.productions[elem[0]][elem[1]]
            for child_index, child in enumerate(children_labels):
                production_tree.add_son(node_index, node_index + child_index)
        return production_tree.get_table()

    def start(self):
        while self.state not in [State.FINAL, State.ERROR]:
            if self.logs:
                print(f"state {self.state}")
                print(f"position:{self.position}")
                print(f"working stack: {self.working_stack}")
                print(f"input stack: {self.input_stack}")
                print("")

            if self.state == State.NORMAL:
                if self.position == len(self.sequence) and not self.input_stack:
                    self.success()
                elif not self.input_stack:
                    self.state = State.BACK
                else:
                    if self.input_stack[0] in self.grammar.get_non_terminals():
                        self.expand()
                    else:
                        if self.position == len(self.sequence):
                            self.momentary_insuccess()
                        elif self.input_stack[0] ==
self.sequence[self.position]:
                            self.advance()
                        else:
                            self.momentary_insuccess()
```

```
            else:
                if self.state == State.BACK:
                    if self.working_stack[-1] in self.grammar.get_terminals():
                        self.back()
                    else:
                        self.another_try()

        if self.state == State.FINAL:
            return self.get_production_string(), "success"
        else:
            return None, "error"
```

Test Cases

```
grammar = Grammar()
grammar.read_grammar("g1.txt")
rd = RecursiveDescendent(grammar, "aacbc", True)
productions, result = rd.start()
print(result)
if result == "success":
    print(productions)
```

The output is:

success

S0 S1 S2 S2

Lab 7

Statement: Implement a parser algorithm (cont.)

PART 3: Deliverables

1. Algorithms corresponding to parsing table (if needed) and parsing strategy

2. Class ParserOutput - DS and operations corresponding to choice 2.a/2.b/2.c (Lab 5) (required operations: transform parsing tree into representation; print DS to screen and to file)

Remark: If the table contains conflicts, you will be helped to solve them. It is important to print a message containing row (symbol in LL(1), respectively state in LR(0)) and column (symbol) where the conflict appears. For LL(1), values (α,i) might also help.

```
class LabeledTree:
    def __init__(self):
        self._sons = defaultdict(list)
        self._label = {}
        self._right_sibling = {}
        self._father = {}

    def add_son(self, node, son):
        if self._sons[node]:
            self._right_sibling[self._sons[node][-1]] = son
        self._sons[node].append(son)
        self._father[son] = node

    def add_label(self, node, node_label):
        self._label[node] = node_label

    def get_table(self):
        table = []
        for node, label in self._label.items():
            table.append((node, label,
self._father.get(node),self._right_sibling.get(node)))
        return table
```

Test Cases

```
grammar = Grammar()
grammar.read_grammar("g1.txt")
rd = RecursiveDescendent(grammar, "aacbc", True)
productions, result = rd.start()
print(result)
if result == "success":
    print(productions)
for i in rd.get_parsing_tree():
```

```
    print(i)
```

Output:

state State.NORMAL

position:0

working stack: []

input stack: ['S']

expand

state State.NORMAL

position:0

working stack: [['S', 0]]

input stack: ['a', 'S', 'b', 'S']

advance

state State.NORMAL

position:1

working stack: [['S', 0], 'a']

input stack: ['S', 'b', 'S']

expand

state State.NORMAL

position:1

working stack: [['S', 0], 'a', ['S', 0]]

input stack: ['a', 'S', 'b', 'S', 'b', 'S'] advance

state State.NORMAL

position:2

working stack: [['S', 0], 'a', ['S', 0], 'a']

input stack: ['S', 'b', 'S', 'b', 'S']

expand

state State.NORMAL

position:2

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 0]]

input stack: ['a', 'S', 'b', 'S', 'b', 'S', 'b', 'S']

momentary insuccess

state State.BACK

position:2

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 0]]

input stack: ['a', 'S', 'b', 'S', 'b', 'S', 'b', 'S']

another_try

state State.NORMAL

position:2

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 1]]

input stack: ['a', 'S', 'b', 'S', 'b', 'S']

momentary insuccess

state State.BACK

position:2

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 1]]

input stack: ['a', 'S', 'b', 'S', 'b', 'S']

another_try

state State.NORMAL

position:2

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 2]]

input stack: ['c', 'b', 'S', 'b', 'S']

advance

state State.NORMAL

position:3

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 2], 'c']

input stack: ['b', 'S', 'b', 'S']

advance

state State.NORMAL

position:4

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 2], 'c', 'b']

input stack: ['S', 'b', 'S']

expand

state State.NORMAL

position:4

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 2], 'c', 'b', ['S', 0]] input stack: ['a', 'S', 'b', 'S', 'b', 'S']

momentary insuccess

state State.BACK

position:4

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 2], 'c', 'b', ['S', 0]]

input stack: ['a', 'S', 'b', 'S', 'b', 'S']

another_try

state State.NORMAL

position:4

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 2], 'c', 'b', ['S', 1]]

input stack: ['a', 'S', 'b', 'S']

momentary insuccess

state State.BACK

position:4

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 2], 'c', 'b', ['S', 1]]

input stack: ['a', 'S', 'b', 'S']

another_try

state State.NORMAL

position:4

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 2], 'c', 'b', ['S', 2]]

input stack: ['c', 'b', 'S']

advance

state State.NORMAL

position:5

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 2], 'c', 'b', ['S', 2], 'c']

input stack: ['b', 'S']

momentary insuccess

state State.BACK

position:5

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 2], 'c', 'b', ['S', 2], 'c']

input stack: ['b', 'S']

back

state State.BACK

position:4

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 2], 'c', 'b', ['S', 2]]

input stack: ['c', 'b', 'S']

another_try

state State.BACK

position:4

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 2], 'c', 'b']

input stack: ['S', 'b', 'S']

back

state State.BACK

position:3

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 2], 'c']

input stack: ['b', 'S', 'b', 'S']

back

state State.BACK

position:2

working stack: [['S', 0], 'a', ['S', 0], 'a', ['S', 2]]

input stack: ['c', 'b', 'S', 'b', 'S']

another_try

state State.BACK

position:2

working stack: [['S', 0], 'a', ['S', 0], 'a']

input stack: ['S', 'b', 'S', 'b', 'S']

back

state State.BACK

position:1

working stack: [['S', 0], 'a', ['S', 0]]

input stack: ['a', 'S', 'b', 'S', 'b', 'S']

another_try

state State.NORMAL

position:1

working stack: [['S', 0], 'a', ['S', 1]] input stack: ['a', 'S', 'b', 'S']

advance

state State.NORMAL

position:2

working stack: [['S', 0], 'a', ['S', 1], 'a']

input stack: ['S', 'b', 'S']

expand

state State.NORMAL

position:2

working stack: [['S', 0], 'a', ['S', 1], 'a', ['S', 0]]

input stack: ['a', 'S', 'b', 'S', 'b', 'S']

momentary insuccess

state State.BACK

position:2

working stack: [['S', 0], 'a', ['S', 1], 'a', ['S', 0]]

input stack: ['a', 'S', 'b', 'S', 'b', 'S']

another_try

state State.NORMAL

position:2

working stack: [['S', 0], 'a', ['S', 1], 'a', ['S', 1]]

input stack: ['a', 'S', 'b', 'S']

momentary insuccess

state State.BACK

position:2

working stack: [['S', 0], 'a', ['S', 1], 'a', ['S', 1]]

input stack: ['a', 'S', 'b', 'S']

another_try

state State.NORMAL

position:2

working stack: [['S', 0], 'a', ['S', 1], 'a', ['S', 2]]

input stack: ['c', 'b', 'S']

advance

state State.NORMAL

position:3

working stack: [['S', 0], 'a', ['S', 1], 'a', ['S', 2], 'c']

input stack: ['b', 'S']

advance

state State.NORMAL

position:4

working stack: [['S', 0], 'a', ['S', 1], 'a', ['S', 2], 'c', 'b']

input stack: ['S']

expand

state State.NORMAL

position:4

working stack: [['S', 0], 'a', ['S', 1], 'a', ['S', 2], 'c', 'b', ['S', 0]]

input stack: ['a', 'S', 'b', 'S']

momentary insuccess

state State.BACK

position:4

working stack: [['S', 0], 'a', ['S', 1], 'a', ['S', 2], 'c', 'b', ['S', 0]]

input stack: ['a', 'S', 'b', 'S']

another_try

state State.NORMAL

position:4

working stack: [['S', 0], 'a', ['S', 1], 'a', ['S', 2], 'c', 'b', ['S', 1]]

input stack: ['a', 'S']

momentary insuccess

state State.BACK

position:4

working stack: [['S', 0], 'a', ['S', 1], 'a', ['S', 2], 'c', 'b', ['S', 1]]

input stack: ['a', 'S']

another_try

state State.NORMAL

position:4

working stack: [['S', 0], 'a', ['S', 1], 'a', ['S', 2], 'c', 'b', ['S', 2]] input stack: ['c']

advance

state State.NORMAL

position:5

working stack: [['S', 0], 'a', ['S', 1], 'a', ['S', 2], 'c', 'b', ['S', 2], 'c']

input stack: []

success

success

S0 S1 S2 S2

(0, 'S', 0, 1)

(1, 'a', 0, 2)

(2, 'S', 2, 3)

(3, 'a', 2, None)

(4, 'S', 4, None)

(5, 'c', None, None)

(6, 'b', None, None)

(7, 'S', 7, None)

(8, 'c', None, None)