Lab 5-7

Corocea Vlad & Guliciuc Stefan

933

Github link: https://github.com/vteccc/FLCD

Statement: Implement a parser algorithm

Lab 5

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

1.a. recursive descendent

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

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

Lab 6

PART 2: Deliverables

1. Algorithm corresponding to parsing tables (if needed) and parsing strategy

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

Lab 7

PART 3: Deliverables

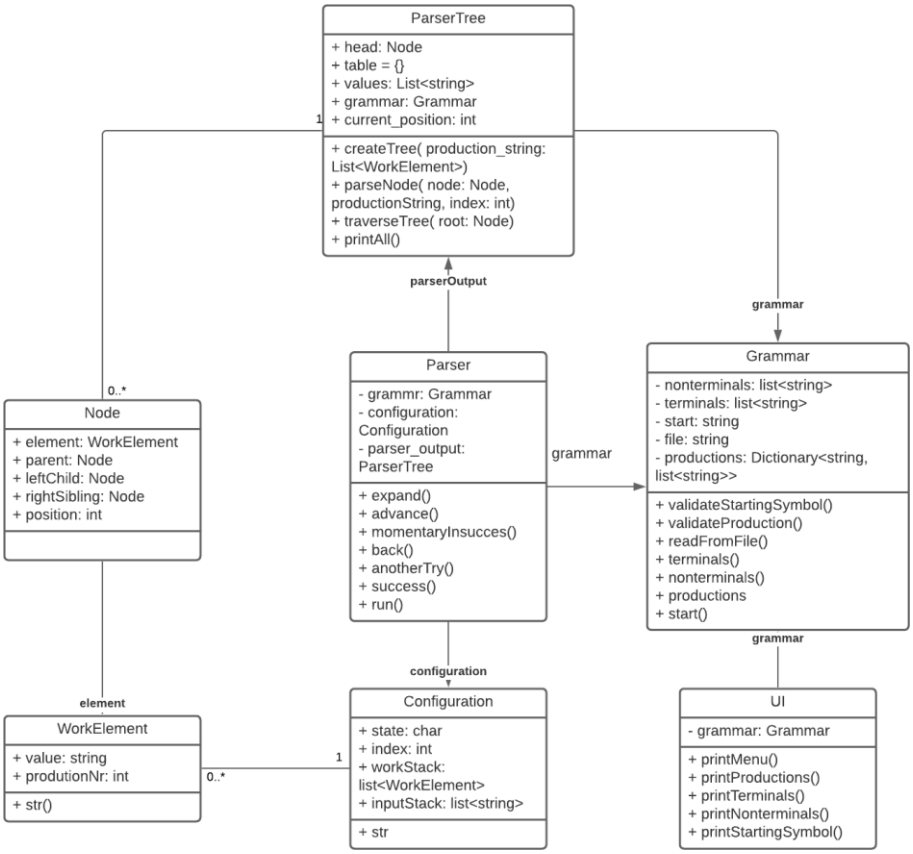
3. Source code
4. Run the program and generate: out1.txt (result of parsing if the input was g1.txt); out2.txt (result of parsing if the input was g2.txt)

Messages: if conflict exists; if syntax error exists (specify location if possible)

5. Code review

Class diagram:

Class diagram: https://lucid.app/lucidchart/invitations/accept/9d6af1aa-b6d9-4a48-94371692b7082e0e



Grammar:

Class structure:

The grammar class stores the necessary information as follows:

- The nonTerminals are stored as a list of strings
- The terminals are stored as a list of strings
- The starting symbol is stored as a string
- The productions are stored as a dictionary that has as key the left hand side, and as value a list which has elements lists of string corresponding to each value in the right hand side

File structure:

The grammar is stored in the file as follows:

- First line: list of nonTerminals
- Second line: list of terminals

- Third line: starting symbol
- Rest of file: the productions as follows: each line has a production, the lhs and rhs are separated by '-' and each possible value of the production is separated by '|'

Recursive descendent parser:

Configuration:

(s, i, α, β) :

- s - current state – represented as an enum
- i - current index
- α - working stack – represented as a list of WorkElements (contains a value and a productionNumber in case it is a nonTerminal)
- β - input stack - represented as a list of strings

Parser:

Class structure:

The class contains the following attributes:

- Grammar: The grammar specific to the parser
- Configuration: The current configuration of the parsing
- ParserOutput: The parserTree used for the output

Method:

- Run – wraps the descendent recursice algorithm , using the correct method corresponding to the move that is required. It stops when the current configuration is equivalent to an error or success.
- Specific methods for each move:
    - State is normal:
        - If the head of the input stack is a nonTerminal: EXPAND
        - If the head of the input stack is a terminal and it is equal to the current symbol in the input: ADVANCE
        - If the head of the input stack is a terminal and it is equal to the current symbol in the input: MOMENTARY INSUCCESS
    - State is back:
        - If the head of the input stack is a nonterminal: ANOTHER TRY
        - If the head of the input stack is a terminal: BACK

Parser Output:

The ParserTree class stores the following information:

- Table: a dictionary which has as key the position, and as value the node itself
- Values: a list of strings corresponding to the values
- Grammar: the grammar of the parser

Method createTree receives as parameter the list of productions string and constructs the table using parseNode recursively. ParseNode gets the first production string and constructs the rest of the tree extracting the corresponding production, adding the production as right siblings, and create another level for the nonTerminals.

Output file structure:

The output file has the following structure:

- Starts with the corresponding tree
- Parent: list with the index of the parent of each node (or –1 if it the root)
- leftChild: list with the index of the left child of each node (or –1 if it does not exist)
- rightChild: list with the index of the right sibling of each node (or –1 if it does not exist)

Grammars (input files):

G1:

```
P Q R
a b
P
P - a Q | b
Q - a Q| b
```

Output for grammar G1 with input: ab

```
P->1
      |--a
      \--Q->2
           \--b

Values: ['P->1', 'a', 'Q', 'b']
Fathers: [-1, 0, 0, 2]
Left children: [1, -1, 3, -1]
Right siblings = [-1, 2, -1, -1]
```