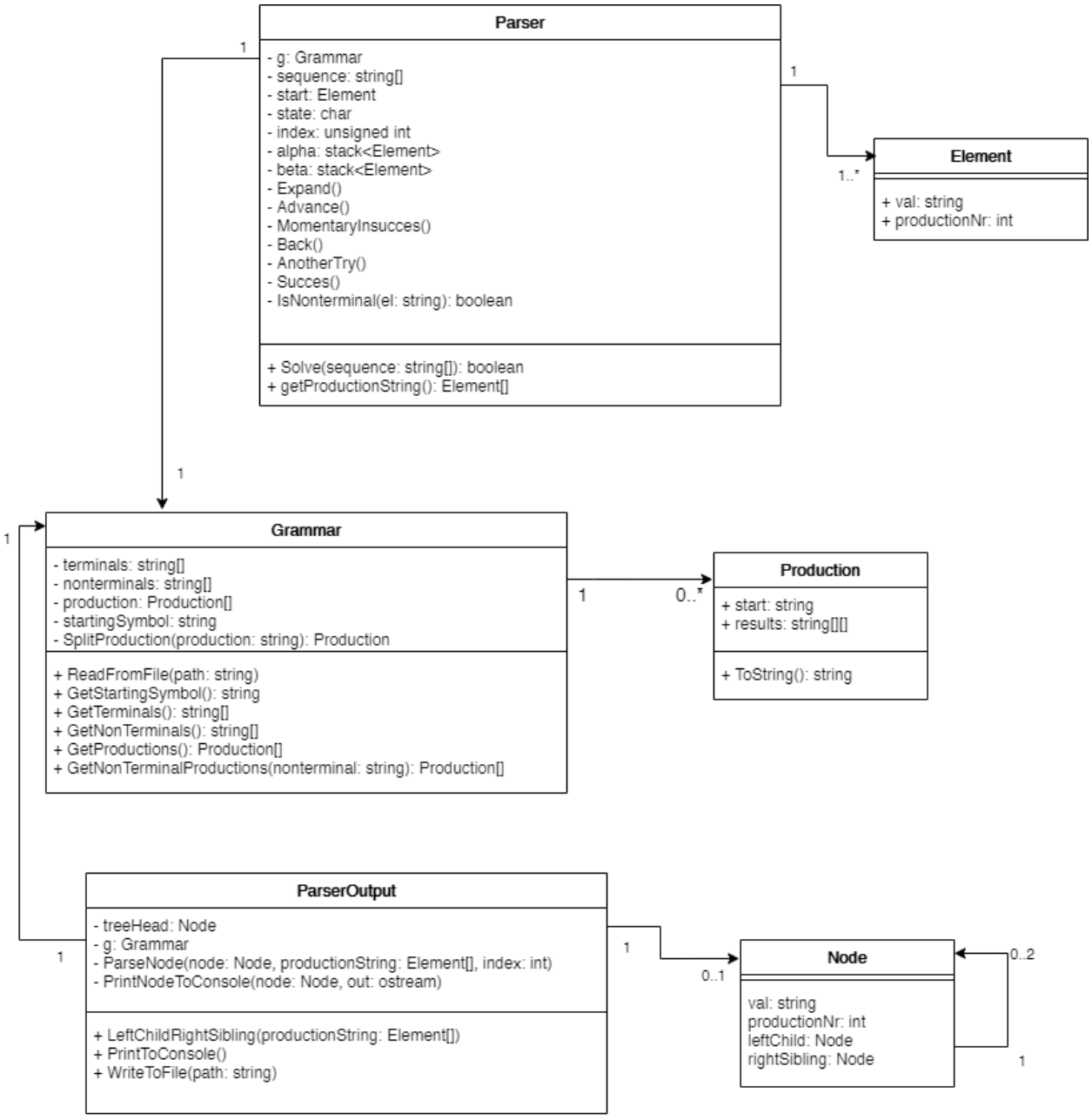
Lab 7 – Dej Architects (Dumbrăvean Bogdan, Trombitaș Richard-Alexandru), 933

GitHub link: https://github.com/BogdanDumbravean/Formal-Languages-and-Compiler-Design/tree/main/Lab9



Parser methods

```
/**
    * performs an "expand" operation when the head of the input stack beta is a non-
terminal
    */
   void Expand();

   /**
    * performs an "advance" operation when the head of the input stack beta is equal
to the current symbol from the input sequence
    */
   void Advance();

   /**
    * performs a "momentary insuccess" operation when the head of the input stack beta
is different from the current symbol from the input sequence
    */
   void MomentaryInsucces();

   /**
    * performs a "back" operation when the head of the working stack alpha is a
terminal
    */
   void Back();

   /**
    * performs an "another try" operation when the head of the working stack alpha is
a non-terminal
    */
   void AnotherTry();

   /**
    * changes the state of the parser to "f" which represents the fact that the
algorithm has successfully finished and the input sequence is accepted
    */
   void Succes();

   /**
    * verifies whether a given string represents a non-terminal element
    * @param el: string
    * @return bool
    */
   bool IsNonterminal(string el);


    /**
     * constructor of a Parser
     * @param _g: Grammar
     */
   Parser(Grammar _g)

   /**
    * verifies whether a given sequence is accepted by the grammar
    * @param sequence: vector<string>
    * @return bool
    */
   bool Solve(vector<string> sequence);
```

```
   /**
    * returns a vector of non terminals from the working stack alpha
    * @return vector<Element>
    */
   vector<Element> GetProductionString();
```

ParserOutput methods

```
/**
    * recursive function which builds the "left child right sibling" tree
starting from a given node, index and production string
    * @param node: Node
    * @param productionString: vector<Element>
    * @param index: int
    */
   void ParseNode(Node* node, const vector<Element> productionString, int&
index);

   /**
    * recursive function which prints a tree starting from a given node to a
given output stream
    * @param node: node
    * @param out: ostream
    */
   void PrintNodeToConsole(Node* node, ostream& out);


    /**
     * constructor of a ParserOutput
     * @param _g: Grammar
     */
   ParserOutput(Grammar _g)

   /**
    * creates a "left child right sibling" tree from a given production
string
    * @param productionString
    */
   void LeftChildRightSibling(vector<Element> productionString);

   /**
    * prints the tree having as root node treeHead to the console
    */
   void PrintToConsole();

   /**
    * prints the tree having as root node treeHead to a file in the specified
location
    * @param path: string
    */
   void WriteToFile(string path);
```

Production methods

```
/**
 * empty constructor of a Production
 */
Production() {}

/**
 * constructor of a Production
 * @param _start: string
 * @param _results: vector<vector<string>>
 */
Production(string _start, vector<vector<string>> _results)


/**
 * returns a string representation of a Production
 * @return string
 */
string ToString();
```

Node methods

```
/**
 * constructor of a Node
 * @param _val: string
 */
Node(string _val = "")
```

Grammar methods

```
/**
 * creates a production from a given string representing the production
 * @param production: string
 * @return Production
 */
Production SplitProduction(string production);

/**
 * reads a grammar from a given file
 * @param path: string
 */
void ReadFromFile(string path);
```

```
/**
 * returns the starting symbol of the grammar
 * @return string
 */
string GetStartingSymbol() { return startingSymbol; }

/**
 * returns the terminals of the grammar
 * @return vector<string>
 */
vector<string> GetTerminals();

/**
 * returns the non-terminals of the grammar
 * @return vector<string>
 */
vector<string> GetNonterminals();

/**
 * returns the productions of the grammar
 * @return vector<Production>
 */
vector<Production> GetProductions();

/**
 * returns the productions corresponding to a given non-terminal
 * @param nonterminal: string
 * @return Production
 */
Production GetNonterminalProductions(string nonterminal);
```

Grammar input (g1.txt):

```
S A
a b
S
S -> a A
A -> a A  | b A | a | b
```

Input sequence (correct):

```
{vector<string>{ "a", "a", "b", "a" }
```

Input sequence (incorrect):

```
{vector<string>{ "a", "c", "b", "a" }
```

Result (for the correct sequence):

```
Sequence Accepted

S
Left child: a

a A

A
Left child: a

a A

A
Left child: b

b A

A
Left child: a

a
```

(The left child right sibling tree is also written to a text file)

Result (for the incorrect sequence):

```
Error! b 0 S 1
```

(We wanted to also print the state, but the algorithm always goes back to the starting state when returning an error)