Documentation - Parser

**Team members:**

* Boanca Tudor
* Boda Viktoria: <https://github.com/BodaViktoria/5_FLCD/tree/master/Lab_8>

**Structure:**

1. Grammar class
2. Parser class
3. ParserOutput class
4. Node class
5. Test class

**Details:**

1. Grammar class

Properties:

* terminals: list of terminals of the grammar
* non\_terminals: list of nonterminals of the grammar
* productions: dictionary of productions of the grammar
* starting\_symbol: starting symbol of the enhanced grammar

Operations:

* get\_grammar\_from\_file(file\_name: string): Reads the content of the file given as parameter, which contains the specifications of the grammar and parses the content and divides it into categories, namely terminals, nonterminals, productions and starting\_symbol.
* get\_production\_for\_nonterminal(nonterminal: string): Returns the production or productions for a nonterminal given as parameter, if the nonterminal exists in the list of nonterminals. Otherwise, it raises an exception.
* verify\_cfg(): Checks if the grammar read from the file is context-free, meaning that on the left hand-side of the productions there is at most one nonterminal. Returns the truth value of the verification.

1. Parser class

Properties:

* grammar: The grammar of the language.
* productions: List of productions of the grammar.
* first: First table of the grammar.
* follow: Follow table of the grammar.
* parseTable: Parse table of the grammar.
* input\_stack: A stack used for the parsing strategy of the grammar, it contains the sequence we would like to parse, divided as a list of entries.
* work\_stack: A stack used for the parsing strategy of the grammar, it contains the productions used for parsing a given sequence.
* sequence: A list of integers, representing the production string, built when parsing.
* parserOutputString: List of strings, containing lines of the output file, consisting of the parsing details, namely the work stack, input stack and sequence.

Operations:

* getNumberOfProduction(nonterminal, production): Returns the number of production, numbered in increasing order as they were read form the input file, for a given nonterminal.
* getProductionBasedOnNumber(number): Returns the production based on the order of it, given as parameter.
* returnProductionWhichIsGeneratingTheNonterminal(nonterminal, terminal): Given a nonterminal and a terminal, returns the production associated with the nonterminal which can generate (has in the first table as entry) the given terminal.
* returnEpsilonProductionOfNonterminal(nonterminal): Returns, for a given nonterminal, the epsilon production, if it has one.
* rearrangeProductions(): Given the list of productions from the grammar file, it parses the productions and transforms them from strings to lists of string for more adequate use.
* first(): Computes and returns eturns the table of first for each nonterminal and terminal, i.e. the first terminal which can be generated after the nonterminal.
* follow(): Computes and returns eturns the follow table for each nonterminal, i.e. the list of terminals which might follow (come after) the nonterminal.
* parseTable(): Computes and returns the parse table of the grammar, having as entries for each nonterminal and terminal each terminal.
* parseSequence(sequence): Parses a sequence with the LL(1) parsing strategy, using an input stach for the entries of the sequence, a work stack for the currently used productions and a list of intigers, representing the production string, i.e. the order number of productions which generate the given sequence.
* createParseTree(productionString): Created and returns the parse tree of the parsed sequence, having as input the production string returned from the parsing method. The parse table has Nodes as entries (described in more detail below).

1. ParserOutput class

Properties:

* outputFile: The file in which we write the parsing details and the parse tree.
* inputFile: The file from which we read the sequence to be parsed.
* parser: The associated Parser class.

Operations:

* writeToFile(): Reads from the input file the sequence which has to be parsed and parses the sequence. If the sequence is syntanctically correct, then it constructs the parse tree and writes the parsing details and the parse tree to the output file. If the sequence has lexical errors, writes the parsing details up to the error to the output file, together with an error message, suggesting at which symbol did the problem occur.

1. Node class

Properties:

* father: The index of the parent/father node of the current node.
* sibling: The index of the right sibling node of the current node.
* value: The value of the node, which can be a terminal or a nonterminal.
* Index: The index of the current node in the parsing table.

Operations:

* \_\_str\_\_(): Returns the string representation of the node.

1. Test class

Properties:

* parser: The parser which we would like to test.

Operations:

* testFirst(): Tests arbitrary, but relevant entries of the first table.
* testFollow(): Tests arbitrary, but relevant entries of the follow table.
* testParsingTable(): Tests arbitrary, but relevant entries of the parsing table.
* testParsingTree (): Tests arbitrary, but relevant entries of the parsing tree.
* testParsingMethod(): Tests arbitrary, but relevant sequences and their syntactical correctness.