

<https://github.com/mocadavid/LFTC/tree/Lab6>

/\*\*

*\* Represents the parsing table.*

*\* Key is formed from the pair row head and column head*

*\* Value is formed from the pair of production result symbols and its number.*

\*/

**public class** ParseTable {

**private** Map<Pair<String, String>, Pair<List<String>, Integer>> **table** = **new** HashMap<>();

/\*\*

*\* Adds a new entry in the parsing table*

*\* @param key given row and column: Pair<String, String>*

*\* @param value formed by the production result symbols and its numbering: Pair<List<String>, Integer>*

\*/

**public void** put(Pair<String, String> key, Pair<List<String>, Integer> value);

/\*\*

*\* Gets the entry from the parsing table by its key.*

*\* @param key given row and column: Pair<String, String>*

*\* @return formed by the production result symbols and its numbering: Pair<List<String>, Integer>*

\*/

**public** Pair<List<String>, Integer> get(Pair<String, String> key);

/\*\*

*\* Checks if a given key is already present in the parsing table.*

*\* @param key given row and column: Pair<String, String>*

*\* @return true if it is contained, false otherwise*

\*/

**public boolean** containsKey(Pair<String, String> key) ;

/\*\*

*\* Transforms the parsing table into a readable string form of key value pairs.*

*\* @return the content of the parsing table :String*

\*/

**@Override**

**public** String toString();

}

```

/**
 * Class which can generate the first and follow sets.
 */
public class Parser {
    private Map<String, Set<String>> firstSet;
    private Map<String, Set<String>> followSet;
    private Grammar grammar;
    private static Stack<List<String>> rules = new Stack<>();
    private ParseTable parseTable = new ParseTable();
    private Map<Pair<String, List<String>>, Integer> productionsNumbered = new HashMap<>();
    private Stack<String> alpha = new Stack<>();
    private Stack<String> beta = new Stack<>();
    private Stack<String> pi = new Stack<>();

    /**
     * Constructor
     */
    public Parser();

    /**
     * getter
     */
    public Map<String, Set<String>> getFirstSet();

    /**
     * getter
     */
    public Map<String, Set<String>> getFollowSet();

    /**
     * getter
     */
    public ParseTable getParseTable();

    /**
     * getter
     */
    public Stack<String> getPi();

    /**
     * getter
     */
    public Map<Pair<String, List<String>>, Integer> getProductionsNumbered();

    /**
     * Initializing the first and follow sets
     */
    private void generateSets();

    /**
     * Generating first for every nonTerminal.

```

```

*/
private void generateFirstSet();

/**
 * Generates the first set for the given nonTerminal.
 * @param nonTerminal: nonTerminal: String
 * @return The set of terminals for the given nonTerminal.
 */
private Set<String> firstOf(String nonTerminal);

/**
 * Generating the follow set for all the nonTerminals.
 */
private void generateFollowSet();

/**
 * Parses a sequence of the form: a list with symbols.
 * @param givenSequence The list with symbols: List<String>
 * @return true if the parsing succeeded, false otherwise
 */
public boolean parse(List<String> givenSequence);

/**
 * Creates the parsing table adding two pairs consisting of the index and column and the production
with its number.
 */
private void createParsingTable();

/**
 * Function used to index the productions in order
 */
private void numberingProductions();

/**
 * Used to set up the alpha, beta and pi stacks.
 * @param givenSequence The list with symbols to be parsed: List<String>
 */
private void initParsing(List<String> givenSequence);

/**
 * Used to add symbols to the stacks used in parsing.
 * @param parsingSequence The list with symbols to be parsed: List<String>
 * @param parsingStacks The stack where to add the symbol.
 */
private void addSymbolsToParsingStacks(List<String> parsingSequence, Stack<String>
parsingStacks);

/*
 * Analyses the productions in which the given nonTerminal is present and calls accordingly the
follow operations with the needed values.
 * @param nonTerminal the given nonTerminal which we examine: String
 * @param initialNonTerminal the starting point for which we search for the follow set: String
 * @return the finalResult of the follow set: Set<String>

```

```

*/
private Set<String> followOf(String nonTerminal, String initialNonTerminal);

/**
 * Decides upon the case of the follow in which we are.
 * @param nonTerminal the nonTerminal for which we search follow: String
 * @param intermediaryResult the list in we save the found elements so far: Set<String>
 * @param terminals the terminals from the grammar: Set<String>
 * @param productionStart the starting nonTerminal of the production
 * @param rule the current production we analyse: String
 * @param indexNonTerminal the index of the nonTerminal: int
 * @param initialNonTerminal the given nonTerminal: String
 * @return the result of the follow operation for the given nonTerminal starting from the
initialNonTerminal: Set<String>
*/
private Set<String> followOperation(String nonTerminal, Set<String> intermediaryResult,
Set<String> terminals, String productionStart, List<String> rule, int indexNonTerminal, String
initialNonTerminal);
}

//manages menus and displaying
public class Console {

    private Parser parser;
    private Grammar grammar;
    private FA fa;

    public Console() {
        this.parser = new Parser();
        this.grammar = new Grammar();
        fa = new FA();
    }

    //shows the menu of the grammar
    private void showGrammarMenu();
    // main function of the console app
    public void start() throws FileNotFoundException;
    //manages options for the parser
    private void startParser() throws FileNotFoundException;

    /**
     * Wrapper function for displaying derivation strings
     * @param pi: The stack with the order of derivations: Stack<String>
     * @param parser the parser which parsed: Parser
     * @return The content to be displayed as derivation Strings: String
     */
    private String displayDerivationString(Stack<String> pi, Parser parser);

```

```

/**
 * Builds the derivations string based on the productions and productions order in the
 pi stack.
 * @param replaceNonTerminal nonTerminals which need to be replaced in order
 given by pi: List<String>
 * @param productionReplacement the production which needs to take place in the
 order given by pi: List<List<String>>
 * @return The content to be displayed as derivation Strings: String
 */
private String buildDerivations(List<String> replaceNonTerminal, List<List<String>>
productionReplacement);

```

```

/**
 * Displays productions string.
 * @param pi: The stack with the order of derivations: Stack<String>
 * @param parser the parser which parsed: Parser
 * @return String as productions string.
 */
private String displayPiProductions(Stack<String> pi, Parser parser);
//manages the options for the grammar
private void startGrammar();
//manages options for the scanner
private void startScanner();
}

```

