Documentation for parsing assignments: <https://github.com/Lois-Beltechi/Formal-Languges-and-Compiler-Design>

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**

**LAB 5**

**PART 1: Deliverables**

1. 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)
2. Input files: g1.txt (simple grammar from course/seminar), g2.txt (grammar of the minilanguage - syntax rules from [Lab 1b](https://moodle.cs.ubbcluj.ro/mod/assign/view.php?id=2562))

**Grammar:**

**Fields:**

- ELEMENT\_SEPARATOR - string representing the separator for the grammar

- SEPARATOR\_OR\_TRANSITION - String representing the mark of transition or the separator for the grammar

- TRANSITION\_CONCATENATION - string representing the mark used to concantenate transitions

- EPSILON - epsilon for grammar

- nonTerminals - set containing the non terminals of the grammar

- terminals - set containing the terminals of the grammar

- productions - map of form <List<String>, List<List<String>> containing the productions of the grammar

- starting symbol - string containing the starting symbol of the grammar

- isCFG - flag marking if the grammar is a context free grammar

- isEnriched - flag marking if the grammar is enriched

- enrichedStartingGrammarSymbol - symbol used to mark the starting symbol for the enriched grammar

**Methods:**

/\*\*

\* With this method we first:

\* -> Split the production by the left right hand side separator ("->")

\* -> Then we split the left hand side by space (we can have A B -> something)

\* -> Then we split the right hand side by "|".

\* -> We go through each production from the right hand side and format each of the in order to be added to the map

\*

\* @param production -> represents the production we are about to work

\*/

**private void processProduction(String production)**

/\*\*

\* With this method we load the content from the file (we read every line from the file and

\* classify everything we read as a terminal/non-terminal/production and so on).

\*

\* @param filePath - the path where the file we are reading from is

\*/

**private void loadFromFile(String filePath)**

/\*\*

\* With this method we check if a grammar is a context free grammar

\* -> First we check if the starting symbols is found within the non-terminals

\* -> Second we check if on the left hand side we have only one non-terminal (for each production)

\* -> Third we check if the productions of that left hand side non-terminal can be found within the non-terminals set or terminals set or is equal to the empty sequence

\*

\* @return true if the grammar is a CFG, false otherwise

\*/

**private boolean checkIfCFG()**

/\*\*

\* With this method we prepare the grammar for the LR0.LR(0) algorithm by adding another starting state S0

\* which has the production S0 -> currentStartingSymbol, if it is already enriched, we just throw an error

\*

\* @return the enriched grammar

\*/

**public Grammar getEnrichedGrammar() throws Exception**

/\*\*

\* With this method, we go through all the productions, and for a non-terminal from the left hand side, we write all the productions separately as pairs

\*

\* @return a list of pairs which represents each production individually

\*/

**public List<Pair<String, List<String>>> getOrderedProductions()**

/\*\*

\* With this method we get the productions for a non-terminal

\* @param nonTerminal - the non-terminal for which we want to get the productions

\* @return - productions of the given non-terminal

\*/

**public List<List<String>> getProductionsForNonTerminal(String nonTerminal)**

**LAB 6**

**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

**LR:**

In our LR class we have the grammar, the working grammar and the ordered productions.

The working grammar is the enriched version of our grammar and we use it for computing the canonical collection of states.

The ordered productions, represent a List with pairs, that contain all the productions, placed separately (So if we had A-> a | ab they would be stored as A -> a, A -> ab)

In this class, we have all the computations for the LR(0) algorithm.

**The methods:**

/\*\*

\* With this method we get the non-terminal which is preceded by dot

\*

\* @param item - the item in which we look for the non-terminal

\* @return - the non-terminal if it is found or null otherwise

\*/

**public String getNonTerminalPrecededByDot(Item item)**

/\*\*

\* With this method we compute the closure for an item (an item being of the form [A->alpha.beta])

\*

\* @param item - the analysis element

\* @return - the closure for item given as input

\*/

**public State closure(Item item)**

/\*\*

\* With this method, in state S, we search LR0.LR(0) item that has dot in front of symbol X.

\* Move the dot after symbol X and call closure for this new item.

\*

\* @param state - the state S from which we want to move

\* @param elem - the symbol after we look

\* @return - returns a State.State containing a list of states

\* composed of the states for each computer closure

\*/

**public State goTo(State state, String elem)**

/\*\*

\* With this method we compute the canonical collection for the grammar.

\*

\* @return - the formed canonical collection

\*/

**public CanonicalCollection canonicalCollection(**

/\*\*

\* With this method we create the parsing table, if possible and detect conflicts if there are any

\*

\* @return - the parsing table corresponding to the parsed grammar if we don't have conflicts

\* - otherwise, return a table with no rows in it

\*/

**public ParsingTable getParsingTable(CanonicalCollection canonicalCollection) throws Exception**

/\*\*

\* With this method we parse the input sequence and find if the sequence is accepted by the grammar or not

\* @param inputStack - the input stack which contains actually all the elements from the sequence

\* @param parsingTable - the parsing table which we will use in order to parse

\* @param filePath - the file path where we will display the parse result

\* @throws IOException - in case of input output exception for writing/reading the file

\*/

**public void parse(Stack<String> inputStack, ParsingTable parsingTable, String filePath) throws IOException**

**CanonicalCollection**

**Fields:**

-states - List that contains the states of the canonical collection

-adjacencyList - Map that keeps track of the states created by another states. In form of <Pair<Integer, String>, Integer>

**Methods:**

- conntectStates(Int, String, Int)

- puts in the map a key as a pair formed by the index of the first state and the symbol, and the value as the index of the second state

- addState(State)

- adds a state to the list of the states

**LAB 7**

**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](https://moodle.cs.ubbcluj.ro/mod/assign/view.php?id=2841)) (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.

**Item:**

An item has the form [A->alpha.beta]

\* Here the idea is the following one, the element preceded by the dot is

\* considered to be the element from the dot position (that's how we represent the dot

\* without actually representing it physically)

\* We have a string for the left hand side, a list of strings for the right hand side

\* and the position for the dot

**Fields:**

- leftHandSide - string containing the the left side of the item

- rightHandSide - list of string containing the elements of the right side of the item

- positionForDot - Integer representing the position of the dot on the right side

**Methods:**

- dotIsLast - returns true if the position of dot is the last one or false otherwise

**State:**

**Fields:**

- stateActionType - contains the state action type of the state

- items - a set containing the set of items of the state

**Methods:**

- getSymbolsSucceedingTheDot() - iterates over all items and creates a list containing the all the symbols that are succeeding the dot

- setActionForState() - sets the action state for a state depending on the match

**StateActionType:**

- can be REDUCE, SHIFT, ACCEPT, REDUCE\_REDUCE\_CONFLICT, SHIFT\_REDUCE\_CONFLICT

**ParsingTreeRow:**

This class is used to represend basically a node from the parsing tree which we are going to compute.

Here, for each “node” we store the index, the information which is basically the symbol (terminal or non-terminal), the parent, the right sibling, left child and level.

**OutputTree:**

The output tree class is used to compute the parse tree.

Here we have stored the root of the parse tree, the grammar, currentIndex, the indexInInput, the maximum level and the tree list.

**currentIndex** will represent the index of each node from the parse tree, which will be increased gradually.

**indexInInput** is used for the list which contains the output band from the parse algorithm, to take the corresponding production from the production list

**maximum level** represents the depth of the tree

The tree list is basically the list which contains the node from our parse tree and will be computed gradually.

**The methods:**

/\*\*

\* With this method we start the generation of the parse tree

\* @param inputSequence - the list which contains the production numbers (represents actually the output band from the parse algorithm)

\* @return - the root of the tree

\*/

**public ParsingTreeRow generateTreeFromSequence(List<Integer> inputSequence)**

/\*\*

\* With this method we build recursively each node from the parse tree

\* If it is a terminal, we try to set its right sibling as well

\* If it is a non-terminal, we try to set its right sibling and its left child as well

\* @param level - the level in the tree

\* @param parent - the parent of the current node from the tree

\* @param currentContent - the current elements which compose the production (So if we had A -> a b, then the currentContent is [a, b])

\* @param inputSequence - the list with the production numbers from the output band of the parse algorithm

\* @return - the newly created node

\*/

**public ParsingTreeRow buildRecursive(int level, ParsingTreeRow parent, List<String> currentContent, List<Integer> inputSequence)**

/\*\*

\* With this method we compute the order in which the nodes should be in the parsing tree

\* And we effectively create it

\* @param node - the node from which we start the construction (the root in our case)

\*/

**public void createList(ParsingTreeRow node)**

**ParsingTable**

**Fields:**

- entries - List that contains elements of type RowTable

**RowTable**

**Fields:**

- stateIndex - integer containing the index of the state on the respective row

- action - a state action type containing the action of the state on the respectiv row

- reduceNonTerminal - a string containing the reduce non-termina of the state

- reduceContent - List of strings containing the reduce contents of the state

- shifts - List of pair string, integer containing the shifts of a state