Parser LL(1) documentation

Team Members:

Ana-Maria Demian
Alex Copindean

Github link: https://github.com/anamariademl/lftc-mini-language/tree/lab7/Parser

Structure

The parser is structured in more classes, each with its responsibility:

- Grammar
- FirstFunction
- FollowFunction
- Table
- Parser
- Tree

Grammar class

Properties

- **terminals** - a list containing all terminals of the grammar
- **nonTerminals** - a list containing all non-terminals of the grammar
- **startingSymbol** - a string representing the starting symbol of the grammar
- **productions** - a hash map containing all the productions of the grammar, the key being the non-terminals and values being possible rules, a rule being a list of terminals and non-terminals
- **productionKeyOrder** - a list containing non-terminals from the production's keys of the grammar in the order they were read from the grammar file (the hash map does not keep the original order and for parsing, the read order is important)

Functions

readFromFile(fileName: String)

Reads the grammar from a given file as a parameter and initializes the properties.

readProduction(line: String)

Reads a line as a production and it splits each character accordingly and is used in **readFromFile**.

validateStartingSymbol()

Validates the starting symbol to be part of non-terminals, in case it is not, an error is thrown.

validateProductions()

Validates the productions by checking the keys to be non-terminals and the values to be defined in the grammar as either terminals, non-terminals or epsilon. If one condition is not met, an error is thrown.

getProductions(nonTerminal: String): ArrayList<List<String>>

Retrieves the rules of the given non-terminal that is part of the productions and if there is no match, an error is thrown.

isContextFree(): Boolean

Returns true if all the keys (left-side) of the production are appearing only once in non-terminals.

getProductions(): HashMap<String, ArrayList<List<String>>>

Retrieves all of the productions in the order they were read from the grammar file.

getProduction(index: Int): Pair<String, List<String>>

Returns a pair of the key and rule found at the respective index.

errorProductionByIndex()

Throws an error based on the requested index in case the one provided for **getProduction()** is not in a valid range.

FirstFunction class

Properties

- **grammar** - The previous processed grammar
- **map** - The map holding the *first* values of each non-terminal of the grammar

Functions

get(key: String): HashSet<String>

Returns the *first* hash set of the given key, or, if the key is a terminal or epsilon, returns a hash set containing the given key.

get(tokens: List<String>): HashSet<String>

Returns all the first sets of given tokens concatenated by calling for each token the **get()** function.

add(key: String, tokens: HashSet<String>): HashSet<String>

Adds all tokens for the given key and returns the added values.

computeConcatenationOfOne(tokenSequence: List<String>): HashSet<String>

Computes concatenation of one for the given sequence. If the sequence is empty, then returns an empty hash set. The first token from the sequence is retrieved and if the sequence contains less than 2 elements, then it returns the *first* of the first token. If epsilon is not present in *first* of the first token, then it returns the set, otherwise, epsilon is removed and it is returned the *first* set concatenated with **computeConcatenationOfOne** for the rest of the given sequence.

getOrCreate(key: String): HashSet<String>

If *first* of key is not empty, then it is returned, else it retrieves all rules from productions based on the given key and for each rule it adds the key with concatenation of one of the rule's sequence.

initialize()

Goes through each key of the grammar's productions and calls **getOrCreate**.

getConcatenationOfOne(tokenSequence: List<String>): HashSet<String>

This function is similar to **computeConcatenationOfOne**, but instead of modifying the map holdin the *first* values, it only retrieves the concatenation based on existing *first* values.

FollowFunction class

Properties

- **grammar** - The previous processed grammar
- **firstFunction** - The previous processed first function

- **map** - The map holding the *follow* values of each non-terminal of the grammar

Functions

get(key: String): HashSet<String>

Returns the *follow* hash set of the given key.

add(key: String, token: String): String

Adds the token to the key's hash set and returns it.

add(key: String, tokens: HashSet<String>): HashSet<String>

Adds the tokens to the key's hash set and returns them.

analyzeToken(token: String): HashSet<String>

Initializes *follow* tokens, if the token is the starting symbol, ***$*** is added. For each rule that has the key different from the token and the token is present in the rule, the result of **analyzeTokenSequence()** is added and all of them are returned.

getOrCreate(key: String): HashSet<String>

Returns the *follow* set for the given key if it is not empty or it returns the result of **analyzeToken()**.

analyzeTokenSequence(key: String, token: String, tokenSequence: List<String>): HashSet<String>

Takes a token as reference and analyzes a rules' key and sequence. If the sequence is empty, it returns an empty set. If the given token is the last token of the sequence, it is returned *follow* of the given key. Right of token is computed by retrieving the right side sequence of the token and applying the *first function* on it. If epsilon is not in the result set, then it is returned, else epsilon is removed and it is added the *follow* of the key.

initialize()

Calls **getOrCreate** for the starting symbol and then for each non-terminal from the given grammar.

Table class

Helper classes

- **Operation** - enum that has the *POP* and *ACCEPT* constant values
- **TableData** - data class that is used to pass *rowSymbol*, *columnSymbol*, *tokens* and *index*

Properties

- **grammar** - The previous processed grammar
- **firstFunction** - The previous processed first function
- **followFunction** - The previous processed follow function
- **data** - The map holding for each row and column symbol the rule and it's index in the grammar's productions

Functions

get(row: String, column: String): Pair<List<String>, Int>

Returns the rule and its index from the table for the given row and column symbols.

set(rowSymbol: String, columnSymbol: String, value: Pair<List<String>, Int>)

If there is a value for the given row and column, an error is thrown notifying the grammar is not LL(1), otherwise the value is added.

plusAssign(data: TableData)

Calls the **set()** functions and maps the data to the function's parameters

setAccept()

Setts the accept criteria in the table, ***$*** as row and column with *ACCEPT* as value with index -1

fillPop()

Fills the pop criteria in the table, for each terminal as row and column with *POP* as value with index -1

mapToEntries(index: Int, rowSymbol: String, tokens: List<String>): List<TableData>

Maps each to rule to a list of *TableData*. If the first token is epsilon, the *follow* values of the row symbol are mapped, else values resulting from concatenation of one of the tokens are mapped.

initialize()

Goes through each production in order from the given grammar, applies the **mapToEntries** function for each rule and then adds the results using **plusAssign()**. Afterwards, **fillPop()** and **setAccept()** are called.

Parser class

Properties

- **grammar** - The previous processed grammar
- **firstFunction** - The previous processed first function
- **followFunction** - The previous processed follow function
- **table** - The previous processed table

Functions

logStack(inputStack: ArrayDeque<String>, workingStack: ArrayDeque<String>, outputStack: ArrayDeque<Int>)

Prints a new line for spacing then each stack's contents.

tryExecuteAccept(inputStackFirst: String, workingStackFirst: String): Boolean

If the first symbol of input stack or working stack is not ***$***, then false is returned signifying that accept failed. The value is obtained from the table based on the first symbol of input and working stack, converted to an *Operation* enum then returns if the value is *ACCEPT*.

tryExecutePop(inputStack: ArrayDeque<String>, workingStack: ArrayDeque<String>): Boolean

If the first symbol of input stack or working stack are different then returns false, signifying that pop failed. The value is obtained from the table based on the first symbol of input and working stack, converted to an *Operation* enum and return false if the value is not *POP*. Afterwards the first symbols from the input and working stack are removed and true is returned.

expandWorkingStack(inputStackFirst: String, workingStack: ArrayDeque<String>, outputStack: ArrayDeque<Int>)

The first symbol from the working stack is obtained and then, with it, the value from the table of first of working and input stack is obtained. The first element from the working stack is replaced with all the rule's tokens excluding epsilon and the index is added at the end of the output stack.

evaluate(sequence: List<String>): List<Int>

The input stack is initialized as the sequence with ***$*** appended at the end, working stack as starting symbol and ***$***, and the output stack as an empty stack. While the input stack and working stack are not empty, **logStack** can be called to see the state of the stacks, then first element from the input and working stack are obtained, **tryExecuteAccept** is called and if it returns true, the loop stops, else **tryExecutePop** is called and if it is false then **expandWorkingStack** gets called. When the loop is ended, the function returns the output stack as a list.

evaluate(sequence: String): List<Int>

Takes the sequence as a list separated by spaces, then splits the elements by spaces and returns the above **evaluate()** function.

Tree class

Helper classes

Node - data class that is used to pass *value*, *child* and *sibling*.

Properties

- **grammar** - The previous processed grammar
- **sequence** - The output of the parser
- **root** - The root node of the tree
- **depthIndex** - The current depth index that is used for printing the output

Functions

buildRecursive(tokenList: List<String>, index: Int = 1): Node?

If the last token from list is not epsilor and the list is empty or the index has exceeded the sequence size, null is returned. Based on the first token from the list, if it is in terminals, then it is returned *Node* of current symbol and call to **buildRecursive()** without the first element being the sibling. If it is in non-terminals, the transition number is obtained based on the element from the

sequence at the *index*, then the rule's tokens are obtained based on the production with transition number as index, returning the *Node* of current symbol, having the call **buildRecursive()** with the rule's tokens and incremented index as node child and the call **buildRecursive()** without the first element of token list and incremented index the sibling. If the current symbol is neither, then a *Node* with the value of epsilon is returned.

build()

Builds the tree calling **buildRecursive()** and sets the result as the child node of the root node that has the value key of the production with the first index from the sequence.

breadthFirstSearch(node: Node?, fileWriter: ((String) -> Unit)? = null, fatherIndex: Int? = null, siblingIndex: Int? = null): List<Pair<Node?, Int>>

Prints the depth index, node's value, father's index and sibling's index as the tree is traversed. If the node is null, then an empty list is returned. The depth index is incremented and stored as index, then **breadthFirstSearch()** is called and its result is concatenated with the current node's child and index and stored as result. If the sibling index is null, then the result is retuned, otherwise, for each element within the result, **breadthFirstSearch()** is called, then an empty list is returned.

writeTo(filePath: String)

With a file path, opens a file descriptor, sets the depth index to 1 and sends the descriptor to **breadthFirstSearch()** for writing each iteration to the given file.

print()

Sets the depth index to 1 and calls **breadthFirstSearch()** for printing each iteration to the console.

Main function

For each solution, a grammar is initialized from a given grammar file, it is validated, then used by the parser which then evaluates a sequence hardcoded as string or extracted from a file as an array of symbols. The grammar together with the evaluation result are then passed to the *Tree* and **print()** or **writeTo()** can be called to display the result as a tree relationship.