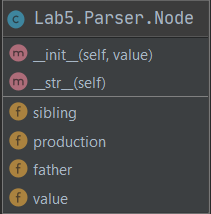
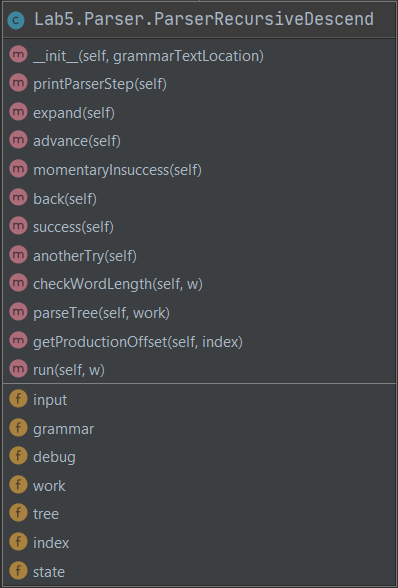
Parsing Labs (L5, L6, L7 Integration)

# Recursive descent parser – implemented in team with Andrei Onița

Github link: <https://github.com/the-coding-cloud/FLCD/tree/main/Lab5>

* Integrated Scanner into this lab (ScannerIntegration folder – for *g2*)
* Implemented the pipeline between the Scanner and Parser – the output of the Scanner (token sequence from PIF) becomes the input for the Parser
* Implemented functions that print to the console each step from parsing
* Wrote g2-codes.txt using the token codes from *tokens.in*, in order to have the matching codification for the grammar of the mini language
* Printed the output of parsing (parse tree as an arbitrary tree with father sibling representation on a table) to an external file (*g1.out, g2-codes.out*)
* Added code review for our classmates (Istvan Olah & Razvan Neta)

# UML Diagrams



# Grammar

## Data Structure

We created the class Grammar which stores the grammar from a text file

Constructor(file)

* Pre:
* In: file: string - location of Grammar definition
* Out: Grammar object
* Post: creates a Grammar object initialised with the terminals, nonTerminals, productions, startSymbol, File

The *terminals, nonTerminals, productions* fields are lists.

## Operations:

Function readFile(filename)

* Pre: an existing fileName path
* In: fileName : String
* Out: -
* Post: the *terminals, nonTerminals, productions, startSymbol* fields are read from the file and initialised
* Desc: reads the Grammar and loads it in the memory

# Parser - Recursive Descent

## Data Structure

We created the class ParserRecursiveDescent

Constructor(file)

* Pre: file - location of Grammar definition
* In: file - string
* Out: a Parser object
* Post: creates a Parser object that initialises with the work and input stacks, index, state

## Operations:

Function expand()

* Pre:
* In:
* Out: -
* Post:
* Desc: expands the non-terminal into its first production of terminals

Function advance()

* Pre:
* In:
* Out: -
* Post:
* Desc: puts one terminal in the work stack

Function momentaryInsuccess()

* Pre:
* In:
* Out: -
* Post:
* Desc: state is change to b

Function back()

* Pre:
* In:
* Out: -
* Post:
* Desc: goes back one index

Function success()

* Pre:
* In:
* Out: -
* Post:
* Desc: state set to s

Function anotherTry()

* Pre:
* In:
* Out: -
* Post:
* Desc: parses the last non terminal to the next set of terminals in its production, or pops the non terminal from work stack to input stack

Function checkWordLength()

* Pre:
* In: word to be checked
* Out: True or False
* Post:
* Desc: checks if the word length wasn’t exceeded by the index and then checks the first letter in the stack with its word counterpart.

Function parseTree()

* Pre: work is the work stack
* In: work – list
* Out: prints tree
* Post: the table respecting the father sibling representation of the parse tree is printed
* Desc: the table respecting the father sibling representation of the parse tree is printed

Function getProductionOffset(index)

* Pre: index is a valid index from the work stack
* In: index - int
* Out: offset
* Post: offset - int
* Desc: returns an offset used in computing the index of a certain node in the parse tree represented using the table with father-sibling notation

Function run(w)

* Pre: w – sequence to parse
* In: w - list
* Out: True/False
* Post: function returns True and prints the parse tree if the sequence is accepted; returns False otherwise
* Desc: function returns True and prints the parse tree if the sequence is accepted; returns False otherwise

# Node

## Data Structure

We created the class Node, which stores the information of a node in the parse tree

Constructor(value)

* Pre:
* In: value - string
* Out: Node object
* Post: creates a Node object initialised with the value given in the constructor and the father, sibling and production fields are initialised with -1 (“invalid” value chosen to represent that no actual value had been assigned to them)