## <https://github.com/KovacsAndrea/FLCD/tree/main/Assignment7>

## Lab 7 DOCUMENTATION

**Write a program that:**

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

# **REGARDING THE PARSER**

Syntactical Analysis is defined, in the most general way, as the process of structuring a sequence based on a set of rules. As a phase of the compilation process, Syntactical analysis can be defined as the component that receives as input the sequence of atoms and determines if it is correct from a syntactical point of view. In the case it is, then the syntax tree is built.

This definition suggests the elements we will need in this step:

* a language to describe valid sequences;
* a method to distinguish between correct and incorrect sequences;

The instrument used in the lexical analysis - the regular grammar - is not powerful enough to express the ᶭ constraints that the syntax that a language imposes on the atoms. For that particular reason, syntactical analysis is based on grammars independent of context G = (N, ∑, P, S)

* ∑ - the alphabet, contains all atoms(terminals)
* N – nonterminals, the syntactical constructions of the language
* P- the productions
* S – starting symbol

Therefore, the problem of determining the correctness of a sequence of atoms can be reduced to verifying that a sequence ᶭ belonging to ∑\* also belongs to L(G)

Depending on the strategy chosen for the construction of the syntax tree, syntactic analyzers can be classified by:

* The direction of in which the tree is built: ascending or descending
* The number of possibilities at one step of construction: recursive or linear

**LL(K)**

LL(K) parser comes from Left-Left and signifies that the input sequence is read from left to right and the syntax tree is built with leftmost derivations.

LL(K) is characterized by the fact that in any rewriting, the choice of the production if uniquely determined:

* The sequence: ᶭ’ = a1…ai
* A – the symbol to be rewritten
* The production of length k: ai+1…ai+k

The production of length k represents the next k symbols that should be generated from the current configuration. For this we introduce the FIRSTk function, that computes the first k symbols that can be obtained by sequential derivations from a certain propositional form

**LL(1)**

Considering the case in which k = 1, we get an efficient algorithm where at each step the symbol A(the nonterminal positioned leftmost in our propositional form) and the production ai (the next symbol in the input) determine uniquely the choice of a production A->α.

A more special situation arises when the first symbol resulting from rewriting the symbol A, FIRST(A), is the empty sequence. Analyzing the already built tree, we see that in this case that we must take into consideration the symbol obtained from what “follows” after A. For this we will introduce a new function, FOLLOW.

FOLLOW: (N U ∑) -> P(∑)

* **added class Parser**
  + **Attributes**
    - private final MyGrammar grammar – the LL(1) grammar
    - public HashMap<String, Set<String>> first; - the set of First Symbols
    - private HashMap<String, Set<String>> follow; - the set of Follow Symbols
  + Methods
    - public void FIRST()
    - public void FOLLOW()
* added class FirstFollowTestRunner
* added TEST\_FIRST\_FOLLOW in ACTIONS enum in class SETTINGS
* to run the tests for First and Follow functions, the ACTION\_SETTINGS variable in Main must be set to SETTINGS.ACTIONS.TEST\_FIRST\_FOLLOW
* **For Lab 7**
* added class ParserOutput
* added class ParserRunner
* added class Node
* added TEST\_PARSER in ACTIONS enum in class SETTINGS

# **EDNF Definitions**

* ***lowerCaseLetter*** *= “a” | “b” | … | “z”*
* ***upperCaseLetter*** *= “A” | “B” | … | “Z”*
* ***letter*** *= lowerCaseLetter | upperCaseLetter*
* ***specialCharacter*** *= “!” | “%” | “\*” | “(” | “)” | “[” | “]” | “{” | “}” | “’” | “”” | “:” | “:” | “|” | “\” | “/” | “.” | “,” | “\_” | “+” | “-”*
* ***digit*** *= “0” | “1” | “2” | … | “9”*
* ***nonZeroDgit*** *= “1” | “2” | … | “9”*
* ***letterSequence*** *= letter {letter}*
* ***digitSequence*** *= {digit}*
* ***nonZeroDigitSequence*** *= {nonZeroDigit}*
* ***sign*** *= “+” | “-“*
* ***state* = letter**
* ***stateSequence* = sequence {sequence}**
* ***initialState* = state**
* ***finalStates* = sequence {sequence}**
* ***alphabetCharacter* = letter | digit | specialCharacter**
* ***alphabet* = alphabetCharacter { alphabetCharacter }**
* ***transition* = state alphabetCharacter alphabet**
* ***transitionSequence* = {transition}**
* ***FA.in* = stateSequence “\n” alphabet “\n” initialState “\n” finalStates “\n” transitionSequence**

# **Settings & Runner Packages & Main Class**

Implemented Settings Class that stores enums.

* ACTIONS.TEST\_FA if this option is selected, tests for the Finite Automata will be run. All requirements for the Lab will be executed, (1, 2 and 3)
* ADTIONS.RUN\_PROGRAMS
  + If this option is selected, the program will scan the problems from the selected files, will generate their symbol table and the pif files
  + RUNNER.STANDARD if this option is selected, the program will use the configurations from the previous lab (elements will be determined using a regex)
  + RUNNERS.FINITE\_AUTOMATA if this option is selected, the program will use the new configurations (elements will be determined using the Finite Automata)
  + Users can choose which problem to run with the PROBLEMS enum

***Main Class***

***Text

Description automatically generated***

* An instance of SETTINGS.ACTIONS and SETTRINGS.RUNNER is created
* If ACTION\_SETTINGS is set to TEST\_FA, the run() function of TestRunner class is called
* IF ACTION\_SETTINGS is set to RUN\_PROGRAMS, we check for the value of RUNNER\_SETTINGS
  + RUNNER\_SETTINGS is set to STANDARD, the run() function of StandardRunner class is called
  + RUNNER\_SETTINGS is set to STANDARD, the run() function of StandardRunner class is called

***Test Runner***

***Text

Description automatically generated***

* An instance of SETTINGS.FA\_TESTS is created
  + FA\_TEST.DFA\_TEST this option holds the path to the DFA.in file. If this option is selected, the test will be run for DFA.in, which stores the data for a deterministic finite automaton
  + FA\_TEST.NFA\_TEST this option holds the path to the NFA.in file. If this option is selected, the test will be run for DFA.in, which stores the data for a non - deterministic finite automaton
* The acceptedSequences holds the sequences that are accepted by the DFA and badSequences has sequences that are not accepted by the DFA. In the run() function, both will be tested.

# **DFA.in**

Calendar

Description automatically generated with medium confidence

***Graphical Representation:***

Diagram

Description automatically generated

Text

Description automatically generated

# **NFA.in**

Graphical user interface, text, application

Description automatically generated

***A screenshot of a computer

Description automatically generated with low confidence***

# **Exceptions Package**

LiteralException Class

* Extends Exception class
* Input: line number, position, file and message
  + message: can be one of the following “ ” expected”, “ ‘ expected”, “Illegal literal”
  + file: the path of the program file being scanned
  + line number: the number of the line where the Lexical Error is found
  + position: the position in the line where the error is encountered
* getMessage function will return a String regarding the details of the Lexical Error encountered

# **Files Package**

* has all .in or .text files used in the project
* .in files for Finite Automata
* .text files for the program file, symbol table files and program internal form files

# **Finite Automata Package**

# **My Components Package**

Component Class

* Parametrized class
* Attributes
  + protected Set<K> \_components: will hold a set of elements
* Methods
  + boolean contains(K k): returns true if \_components contains element k and false otherwise
  + String toString(): returns a string consisting of all elements saved in \_components separated by a space

Alphabet Class

* Extends Component Class

State Class

* Extends Component Class

FinalState Class

* Extends State Class

Transition Class

* Transitions are kept in a map, where the key is a pair of two Strings(Source State and Alphabet Value) and the Value is Set of Strings with Resulting States

***FA Class***

Attributes

* public State \_states;
* public Alphabet \_alphabet;
* public String \_initialState;
* public FinalState \_finalStates;
* public Transition \_transitions;

Methods

* void initFA(String filename): The Fa will be read from a file, given as parameter to the constructor. First line will represent states, second line will represent the alphabet, third row will represent the initial state, fourth row will represent the final states, and all following rows will represent transitions. Duplicate or invalid transitions will be ignored
* public boolean isDFA():

For a non-deterministic finite automaton (NFA), for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. For a deterministic finite automaton (DFA), a pair of transition and symbol can compute to only one state.

Below is the representation for a DFA:

Calendar

Description automatically generated with medium confidence

Each pair of State and Symbols computes to only one state, so we know exactly to what state the machine will move.

A NFA will look like this:

Text

Description automatically generated with medium confidence

Therefore, the isDFA() function will return true if all sets have length 1 and false otherwise

* boolean accepts(String sequence): On the first run, the current state will be the initial state. We parse each character of the given sequence, and we check that the pair formed by the current state and the character itself is found in the key set of \_transitions. If it is, the state that the pair computes to, will become the current state and we move on to the next character. If the pair is not found in the transitions key set, we return false. If we reach the end of the sequence and the execution was not interrupted, we return true if the current state is the final state and false otherwise.

***DFA Class***

* creates four instances of FA
  + public final FA \_identifier: the FA for identifiers. The data for this FA is stored in DFAIdentifier.in
  + public final FA \_string: the FA for string constants. The data for this FA is stored in DFAString.in
  + public final FA \_char: the FA for char constants. The data for this FA is stored in DFAChar.in
  + public final FA \_integer: the FA for integer constants. The data for this FA is stored in DFAInteger.in

# **Specification Package**

Constants

* static Class
* has three static attributes: a regex for numeric patterns (signed and unsigned), a regex for char patterns, and a regex for string patterns
* has three static methods that are getters for the three regexes

Identifiers

* has the regex for identifiers and a getter

Operators

* stores all operators
* getAll() getter for the stored operators
* isOperator function returns true if a string is an operator and false otherwise
* isPartOfOperator is used for operators that contain more than one character, such as ==, !=, <=, =>, ++, --, returns true if the given string is =, !, <, + or – and false otherwise

ReservedWords

* Stores all reserved words
* getAll(): getter for reserved words
* isReservedWord function returns true if a string is a reserved word and false otherwise;

Separators

* Stores all separators
* getAll(): getter for separators
* isSeparator function returns true if a string is a separator and false otherwise;

***Specifications***

* Attributes
  + final static HashMap<String, Integer> codification – will hold sets of Strings and Integers, where the Strings are Reserved Words, Operators, Separators, indicator for Constants and Identifiers. Will hold 0 for identifiers and 1 for constants;
* Methods
  + isConstant(String s) returns true if s is a Constant and false otherwise
  + isIdentifier(String s) returns true is f is an Identifier and false otherwise
  + isSymbol(String s) returns true if s is an Operator, Reserved Word, or Separator and false otherwise
  + createCodifications() places tuples 0 “identifiers” and 1 “constants” in the codification HashMap; for all other symbols, increments the code and places the tuple in the HashMap
  + getCode(String token) returns the code of a token

# **Core Package**

Pair

* parametrized record class
* getKey() returns the key of the element
* getValue() returns the value of the element
* equals() override for the equals function, returns true if the keys and the values are equal

HashTable

* class has 2 attributes
* \_variablesAndConstants is an array of arrays (hash table)
* \_size is an integer, initialized in the constructor
* getSize(): returns the value stored in \_size
* hash(): computes the hash value of a token(string); hash value will be computed by sum%\_size where sum is the sum of the ascii values of all characters in the string, and the \_size is the size given initially to the hash table
* add(): will add a token in the table, if the key already exists, the token will not be added again; if two elements hash to the same position, the new element will be added on the next position in the array
* contains(): returns true if the hash table contains the string and false otherwise
* getPosition(): returns Pair<Integer, Integer>, where the key is the position of the secondary array in the main array, and the value is the position of the element in the secondary array
* remove(): removes an element from the hash table

PIF

* Attributes
  + final List<Pair<Integer, Pair<Integer, Integer>>> pif – a list of pairs where the first value is an Integer and the second value is a pair of integer and integer
* Methods
  + void add(Integer code, Pair<Integer, Integer> value)
  + receives a code and a Pair of integer and integer and adds is to the pif attribute

MyScanner

* Attributes
  + private final HashTable \_symbolTable – will hold a reference to an instance of HashTable class
  + private final PIF \_pif - will hold a reference to an instance of PIF class
  + private final String \_programFile – will hold a reference to the program file
  + private final String \_PIFFile – will hold a reference to the file where the resulting symbol table will be written
  + private final String \_STFile – will hold a reference to the file where the resulting program internal form will be written
  + private int lineNr – will hold the number of the current line that is being scanned; initialized with 1
  + private boolean \_lexicalCorrect – will be true if the program is lexically correct and false otherwise; initialized with true
* Methods
  + void scan() – lineNr is initialized with 1, program file is opened. For each line calls the runTokens() function and receives an Array of strings representing the tokens found on that specific line. Adds the tokens in a List of pairs, where the first value is a token and the second value is the line number. After iterating the file, if \_lexicalCorrect is true, it prints a message stating as such. Calls buildPIF() and writeResults() functions.
  + List<String> runTokens(String line) - a wrapper for tokenize() function. It runs the tokenize() function and if it encounters any exceptions, it prints its message and sets \_lexicalCorrect to false
  + List<String> tokenize() receives a line and checks for constants, identifiers, operators, separators and reserved words, and adds them in the \_hashTable
  + String getStringConstant(String line, int position) checks for the next position of the character “. If it is not fount, it throws a new LiteralException with the message “” expected”. If it is found, it generates a substring from the first “ and checks if it matches the string pattern
  + String getCharConstant(String line, int position) checks for the next position of the character ‘. If it is not fount, it throws a new LiteralException with the message “’ expected”. If it is found, it generates a substring from the first ‘ and checks if it matches the string pattern
  + String getIdentifier(String line, int position) checks for the position of the next separator, operator or white space. If it is not found, a substring from the given position and to the end of the line is generator. If it is found, a substring from the given position and the new found position is generated. Returns the substring if it matches the identifier pattern, throws a LiteralException otherwise. Function will also return reserved words. In the tokenize function e will check if the string is a reserved word. If it is, we will not add it to the hash table
  + String getInteger(String line, int position): If integer is signed, checks if there is a space between + or - and the integer. If what follows after the sign is an identifier, it returns null. Otherwise it checks for the position of the next white space, arithmetic operator or relational operator. If it is found, generates a string from the given position to the found symbol, otherwise, to the end of the line. If the string does not match the numeric pattern, it throws a Literal Exception, or returns it otherwise.
  + void buildPIF(List<Pair<String, Integer>> tokens) – receives a list of Pairs of String and Integer.
    - If it is a symbol, \_pif will add the code saved in Specifications Class, and the Pair will fold values -1 and -1
    - If it is an identifier \_pif will add code 0 and the position of the identifier saved in the symbol table
    - If it is a constant \_pif will add code 1 and the position of the constant saved in the symbol table
  + void writeResults() will open the \_PIFFile and will write the data in \_pif. Will open \_STFile and will write the data in \_symbolTable.

MyFAScanner

* works exactly like MyScanner Class but uses the Finite Automata class (FA) instead of regexes to determine the correctness of constants, identifiers and symbols.