Parser

Problem statement: Implement a parser algorithm

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)

2. The representation of the parsing tree (output) will be (decided by the team):

   2.a. productions string (max grade = 8.5)

   2.b. derivations string (max grade = 9)

   2.c. table (using father and sibling relation) (max grade = 10)

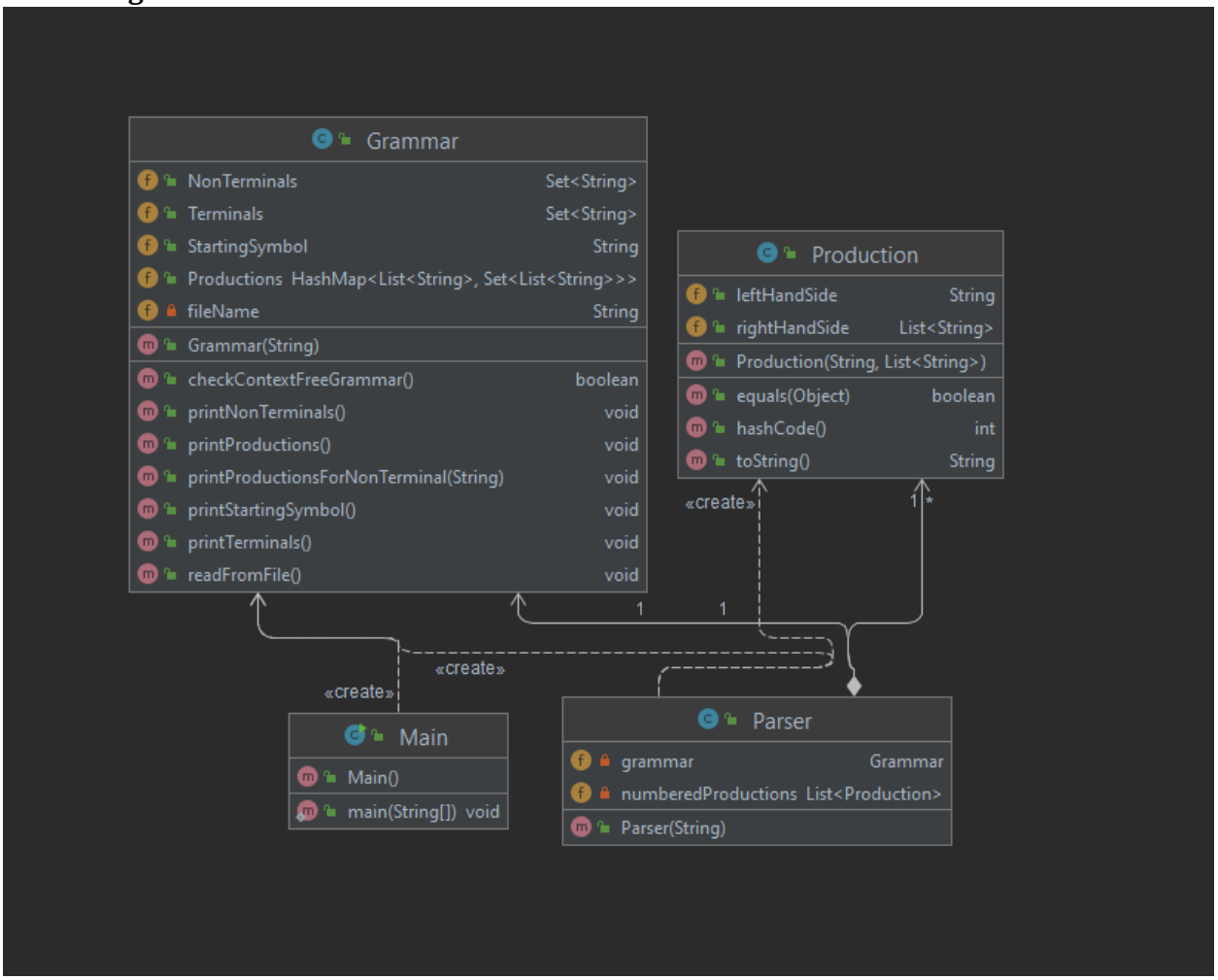
PART 1: Deliverables

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)

Github link:

https://github.com/CimpeanAndreea/FLCD/tree/master/Lab5

Class Diagram:



Grammar class keeps the representation of the grammar and checks if the grammar is context free. The **class Parser** contains just a grammar for now and keeps a list of simple **Productions (eg. S->aB)** in order to have them "numbered" for future development of the parsing algorithm.

<u>Grammar:</u>

NonTerminals: Set(String)

Terminals: Set(String)

StartingSymbol: String

Productions: Map(String[], Set(String[])) -> it is a map between lhs and rhs of a production, where lhs can have more terms (in the case of not CFG) and rhs can represent the multiple results of the production

Important functions:

Function checkContextFreeGrammar():

description: checks if a grammar is context free, by looking at the productions map and checking that the lhs of each production has exactly one element

pre: the grammar was read from a given file, and its attributes were completed based on that

post: returns true/false if the grammar is/is not context free

Tests:

g1.txt

```
S A
a b c
S
S ::= a A
A ::= b A | c | epsilon
```

TERMINALS

[a, b, c]

NON TERMINALS

[A, S]

STARTING SYMBOL

S

PRODUCTIONS

A -> b A | c | epsilon

S -> a A

PRODUCTIONS FOR NON TERMINAL:A

A -> b A | c | epsilon

Is CFG: true

g2.txt

```
program declarations_list declaration statements_list simple_declaration
array_declaration const_declaration simple_type list_identifiers
initialized_identifier list_const_identifiers initialized_const
list_array_identifiers array_identifier statement simple_statement
compound_statement struct_statement assign_statement read_statement
write_statement expression array_element term factor list_identifiers_print
list_outputs output relation if_statement while_statement for_statement
condition
+ - * / < > <= >= == != <- -> << >> % ! , ( ) [ ] { } : ; main declarations
statements integer string boolean character array const and or in out if else
then for while identifier constant do
program
program ::= main -> { declarations declarations_list ; statements
statements_list }
declarations_list ::= declaration | declaration ; declarations_list
declaration ::= simple_declaration | array_declaration | const_declaration
simple_declaration ::= simple_type : list_identifiers
simple type ::= character | integer | boolean | string
list_identifiers ::= identifier | initialized_identifier | identifier ,
list_identifiers | initialized_identifier , list_identifiers
initialized_identifier ::= identifier <- constant
const_declaration ::= simple_type const : list_const_identifiers
list_const_identifiers ::= initialized_const | initialized_const ,
list_const_identifiers
initialized_const ::= identifier <- constant
array_declaration ::= array [ simple_type ] : list_array_identifiers
list_array_identifiers ::= array_identifier | array_identifier ,
list_array_identifiers
array_identifier ::= identifier [ identifier ] | identifier [ constant ]
statements_list ::= statement | compound_statement
statement ::= simple_statement | struct_statement
simple_statement ::= assign_statement | read_statement | write_statement
assign_statement ::= identifier <- expression | array_element <- expression
expression ::= expression + term | expression - term | term
term ::= term * factor | term / factor | factor
factor ::= ( expression ) | identifier | constant | array_element
read_statement ::= in >> list_identifiers | in ( constant ) >>
list_identifiers
list_identifiers_print ::= identifier | array_element | identifier ,
list_identifiers_print | array_element , list_identifiers_print
write_statement ::= out << list_outputs
list_outputs ::= output | output , list_outputs
output ::= identifier | constant | expression | array_element
compound_statement ::= { statement ; { statement ; } }
struct_statement ::= if_statement | while_statement | for_statement
if_statement ::= if ( condition ) simple_statement else simple_statement | if
( condition ) compound_statement else simple_statement | if ( condition )
simple_statement else compound_statement | if ( condition )
compound_statement else compound_statement
while_statement ::= while ( condition ) do simple_statement | while
( condition ) do compound_statement
for_statement ::= for ( assign_statement ; condition ; assign_statement ; )
condition ::= expression relation expression
relation ::= > | < | == | <= | >= | != | and | or
array_element ::= identifier [ identifier ] | identifier [ constant ]
```

TERMINALS

[<<, >>, <=, constant, string, const, for, main, statements, integer, do, while, out, character, array, and, else, [,], if, ==, !, identifier, or, %, in, (,), *, +, then, ,, -, declarations, /, <-, ->, boolean, :, {, ;, <, !=, }, >, >=]

NON TERMINALS

[compound_statement, list_identifiers, simple_type, simple_statement, list_array_identifiers, program, list_const_identifiers, array_identifier, relation, output, list_outputs, read_statement, array_declaration, statement, term, while_statement, factor, initialized_const, struct_statement, array_element, const_declaration, expression, initialized_identifier, list_identifiers_print, declarations_list, statements_list, declaration, for_statement, condition, assign_statement, simple_declaration, if_statement, write_statement]

STARTING SYMBOL

program

PRODUCTIONS

struct_statement -> for_statement | if_statement | while_statement

initialized_identifier -> identifier <- constant

const_declaration -> simple_type const : list_const_identifiers

array_element -> identifier [constant] | identifier [identifier]

list_identifiers_print -> identifier | array_element | identifier , list_identifiers_print | array_element , list_identifiers_print

statements_list -> compound_statement | statement

expression -> expression + term | term | expression - term

declarations_list -> declaration ; declarations_list | declaration

declaration -> array_declaration | const_declaration | simple_declaration

for_statement -> for (assign_statement ; condition ; assign_statement ;)

condition -> expression relation expression

assign_statement -> array_element <- expression | identifier <- expression

if_statement -> if (condition) simple_statement else simple_statement | if (condition) compound_statement else compound_statement | if (condition) simple_statement else compound_statement | if (condition) compound_statement else simple_statement

simple_declaration -> simple_type : list_identifiers

write_statement -> out << list_outputs

compound_statement -> { statement ; { statement ; } }

simple_type -> character | string | integer | boolean

relation -> <= | or | and | < | != | > | >= | ==

list_identifiers -> identifier | initialized_identifier | identifier , list_identifiers | initialized_identifier , list_identifiers

simple_statement -> assign_statement | read_statement | write_statement

list_array_identifiers -> array_identifier , list_array_identifiers | array_identifier

program -> main -> { declarations declarations_list ; statements statements_list }

list_const_identifiers -> initialized_const | initialized_const , list_const_identifiers

array_identifier -> identifier [constant] | identifier [identifier]

list_outputs -> output | output , list_outputs

array_declaration -> array [simple_type] : list_array_identifiers

output -> identifier | array_element | constant | expression

initialized_const -> identifier <- constant

read_statement -> in (constant) >> list_identifiers | in >> list_identifiers

statement -> struct_statement | simple_statement

while_statement -> while (condition) do simple_statement | while (condition) do compound_statement

term -> term / factor | term * factor | factor

factor -> identifier | array_element | constant | (expression)

PRODUCTIONS FOR NON TERMINAL:program

program -> main -> { declarations declarations_list ; statements statements_list }

Is CFG: true