Comănac Dragoș-Mihail

https://github.com/ComanacDragos/ToyLanguageCompiler

Statement: Implement a parser algorithm (cont.) - as assigned by the coordinating teacher, at the previous lab

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*)

Implementation

In grammar class the following function is added

For a given production id returns the next production of the left-hand side if it exists

Otherwise return null

```
public Production getNextProductionForNonTerminal(Long previousId)
```

The following classes are introduced:

```
public class NonTerminalWithProduction extends NonTerminal
   Long productionId;
```

```
public enum ParserState
   NormalState,
   BackState,
   FinalState,
   ErrorState
```

```
public class Parser
   Grammar grammar;
   ParserState currentState = ParserState.NormalState; // current state of the parser
   int i = 0; // position of the current symbol in input sequence
   Deque<Symbol> workingStack = new ArrayDeque<>();
   Deque<Symbol> inputStack = new ArrayDeque<>();
```

```
// performs Recursive descendent algorithm on the given sequence
public void parse(List<String> sequence)
```

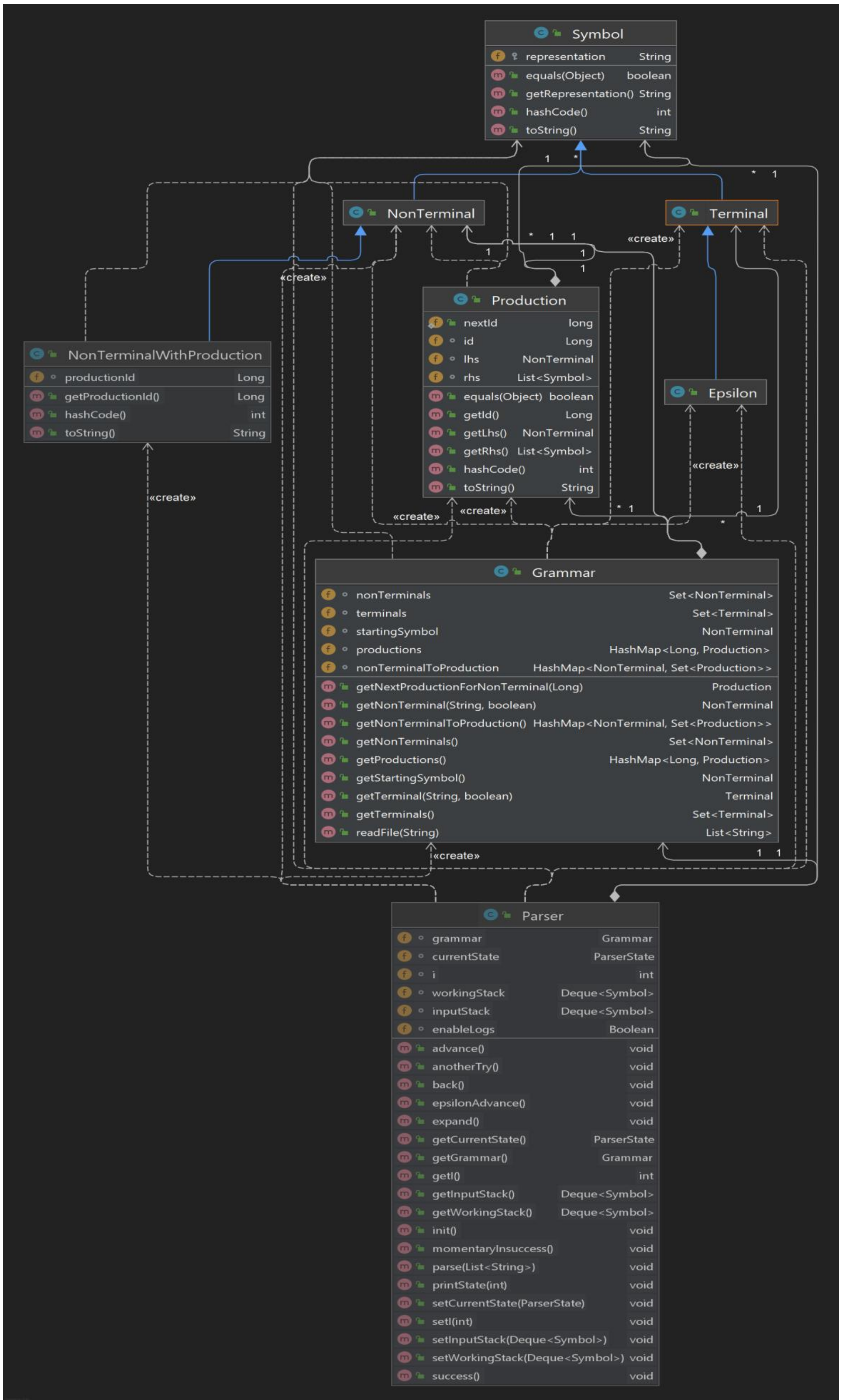
The parse function uses the following functions corresponding to the Descendent Recursive algorithm's actions:

Comănac Dragoș-Mihail

```
// initializes the parser state
public void init()

public void expand()

public void advance()

public void epsilonAdvance()

public void momentaryInsuccess()

public void back()

public void anotherTry()

public void success()
```

Comănac Dragoș-Mihail

Image is also available on GitHub in documentation/lab9 folder


Symbol
representation String
equals(Object) boolean
getRepresentation() String
hashCode() int
toString() String
NonTerminal
Terminal
«create»
«create»
Production
nextId long
id Long
lhs NonTerminal
rhs List<Symbol>
equals(Object) boolean
getId() Long
getLhs() NonTerminal
getRhs() List<Symbol>
hashCode() int
toString() String
NonTerminalWithProduction
productionId Long
getProductionId() Long
hashCode() int
toString() String
Epsilon
«create»
«create»
«create»
«create»
Grammar
nonTerminals Set<NonTerminal>
terminals Set<Terminal>
startingSymbol NonTerminal
productions HashMap<Long, Production>
nonTerminalToProduction HashMap<NonTerminal, Set<Production>>
getNextProductionForNonTerminal(Long) Production
getNonTerminal(String, boolean) NonTerminal
getNonTerminalToProduction() HashMap<NonTerminal, Set<Production>>
getNonTerminals() Set<NonTerminal>
getProductions() HashMap<Long, Production>
getStartingSymbol() NonTerminal
getTerminal(String, boolean) Terminal
getTerminals() Set<Terminal>
readFile(String) List<String>
«create»
Parser
grammar Grammar
currentState ParserState
i int
workingStack Deque<Symbol>
inputStack Deque<Symbol>
enableLogs Boolean
advance() void
anotherTry() void
back() void
epsilonAdvance() void
expand() void
getCurrentState() ParserState
getGrammar() Grammar
getI() int
getInputStack() Deque<Symbol>
getWorkingStack() Deque<Symbol>
init() void
momentaryInsuccess() void
parse(List<String>) void
printState(int) void
setCurrentState(ParserState) void
setI(int) void
setInputStack(Deque<Symbol>) void
setWorkingStack(Deque<Symbol>) void
success() void

Comănac Dragoș-Mihail

Testing

TestParser is a class that tests the Parser functions

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

```
int[256] a;
int i=0;
int n;
<<n;
if (i<n){
  <<a[i];
  i=i+1;
}
n=0;
```

```
program statement_list statement simple_statement compound_statement simple_type
array_type type expression binary_operator unary_operator declaration_statement
iostatement assignment_statement if_statement else_branch while_statement expression'
expression_simple
id constant int char bool string float >> << while if else and or ! + - * / % > < >= <= != == = ; [ ] { } (
) , ^
program
program ::= statement_list
statement_list ::= statement | statement statement_list
statement ::= simple_statement | compound_statement

simple_statement ::= assignment_statement ; | iostatement ; | declaration_statement ;

compound_statement ::= if_statement | while_statement

simple_type ::= bool | char | int | string | float

array_type ::= simple_type [ constant ]
```

Comănac Dragoș-Mihail

type ::= simple_type | array_type

expression_simple ::= constant | id | id [constant] | id [id] | unary_operator expression | (expression)

expression' ::= binary_operator expression expression' | epsilon

expression ::= expression_simple expression'

declaration_statement ::= type id | type id = expression

iostatement ::= << id | << id [constant] | << id [id] | >> expression

assignment_statement ::= id = expression

if_statement ::= if (expression) { statement_list } else_branch
else_branch ::= epsilon | else { statement_list }

while_statement ::= while (expression) { statement_list }

unary_operator ::= !
binary_operator ::= + | - | * | / | ^ | % | and | or | > | < | >= | <=| != | ==