МИНИСТЕРСТВО НАУКИ
И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования «Новосибирский государственный технический университет»

НГТУ НЭТИ

Кафедра прикладной математики

Практическая работа №3

по дисциплине «Языки программирования и методы трансляции»

ФПМИ

Группа	ПМ-92
Вариант	7
Студенты	Кутузов Иван
	Иванов Владислав
Преподаватель	Еланцева И. Л.
Дата	17.03.2022

Новосибирск

Цель работы:

Изучить табличные методы синтаксического анализа. Получить представление о методах диагностики и исправления синтаксических ошибок. Научиться проектировать синтаксический анализатор на основе табличных методов.

Задание:

Подмножество языка C++ включает:

- данные типа int, float, массивы из элементов указанных типов;
- инструкции описания переменных;
- операторы присваивания в любой последовательности;
- операции +, – , *, = =, != , <, > .

В соответствии с выбранным вариантом задания к лабораторным работам разработать и реализовать синтаксический анализатор с использованием одного из табличных методов (LL-, LR – метод, метод предшествования).

Грамматика языка:

Начальный символ:
S:=“void”, “main”, “(“, “)”, “{“, <command>, “}”

Типы:
types:=“int” | “float” | “int”, “[“, “]”

Операции:
operation:=“+” | “-” | “*” | “!=” | “==” | “<” | “>”

Команды:
command:={<type>, <id>, “;”, <program>} | {<expr>, “;”, <program>} | eps

id:=”identifer”, {“,”, ”identifer”}, {<eq>, <expr>} | eps

expr:=<id> | “const”, <operation>, <id> | “const”, {<operation>, <id> | “const”} | eps

Схема разбора:

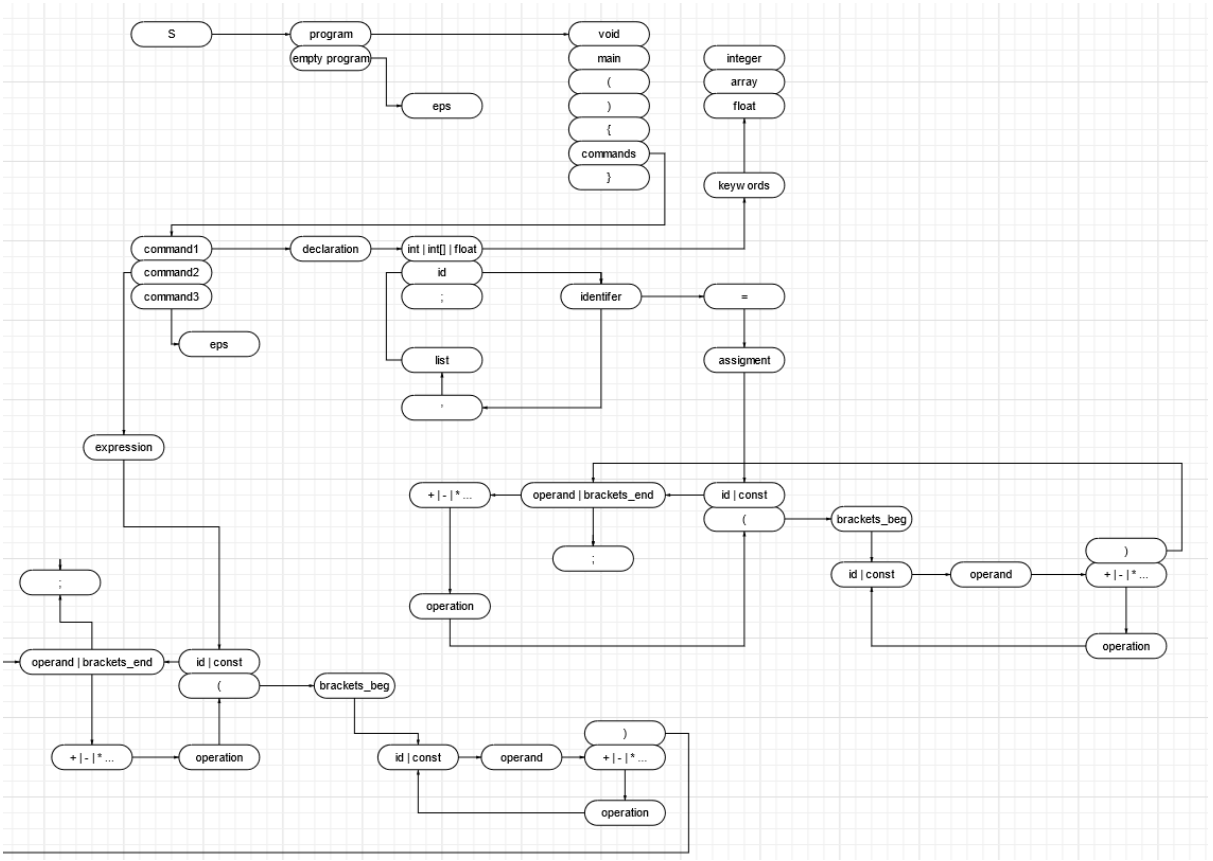

Таблица разбора:

	int	[	ARR	]	id	IDE	;	=	ASSI	{	const	,	};
5	s6												
6		s7			s#								
7			r8										
8				s9									
9					s10								
10						r11							
11							r5	s12					
12									r13				
13										s14			
14											s15		
15												s14	r5

	id	[	ARR_IDE	INT_ID	const	]	+-*<>	OPER	;	END_OF_COMMAND
5	s16									
16		s17		r20						
17					s18					
18						s19				
19			r20							
20							s21		s23	
21								r22		
22	s16									
23										r5

Тесты:

Без ошибок

```
void main() {
    int[] a = {1, 2, 3, 4, 5};
    int b = 5;
    float c = 1.5;
    a[0] + (b - 155);
}
```

Постфиксная запись:

```
a 1 2 3 4 5 = ; b 5 = ; c 1.5 = ; a [ 0 ] ( b 155 - ) + ; } { ) ( main void
C:\Users\ivale\source\repos\Compiler\Debug\Compiler.Tables.exe (процесс 22276) завершил работу с кодом 0.
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" ->"Параметры" ->"Отладка" -> "Ав
томатически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно:
```

Индекс за пределами допустимого

```
void main() {
    int[] a = {1, 2, 3, 4, 5};
    a[5] + 4;
}
```

Ошибка:

```
index of {a} out of range. in line: 2
C:\Users\ivale\source\repos\Compiler\Debug\Compiler.Tab
Чтобы автоматически закрывать консоль при остановке от
томатически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно:
```

Неинициализированная переменная и отсутствие конца команды

```
void main() {
    float b;
    b + 5.0
}
```

Ошибка:

```
syntax error: uninitialized variable {b} in line 2
syntax error: expected { | + | - | * | ; | == | != | < | > | } in line 2

C:\Users\ivale\source\repos\Compiler\Debug\Compiler.Tables.exe (процесс 4
Чтобы автоматически закрывать консоль при остановке отладки, включите пар
томатически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно:
```

Несовпадение типов

```
void main() {
    float b;
    int a = b;
}
```

Ошибка:

```
syntax error: expected { | ( | int const | int identifer | array identifer | } in line 2

C:\Users\ivale\source\repos\Compiler\Debug\Compiler.Tables.exe (процесс 20348) завершил ра
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" ->
томатически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно:
```

Переопределение переменной (повторное объявление)

```
void main() {
    float b;
    int[] b;
}
```

Ошибка:

```
syntax error: re-declaring a variable {b} in line 2

C:\Users\ivale\source\repos\Compiler\Debug\Compiler.Tables
Чтобы автоматически закрывать консоль при остановке отлад
томатически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно:
```

Арифметические операции с массивами

```
void main() {
    int[] a = {1, 2, 3};
    a + 3;
}
```

Ошибка:

```
syntax error: expected { | [ | } in line 2

C:\Users\ivale\source\repos\Compiler\Debug\Compiler.
Чтобы автоматически закрывать консоль при остановке
томатически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно:
```

Текст программы:

Syntax.h

```
enum class State {
	END_OF_COMMAND,
	SHIFT_TERMINAL,
	SHIFT_NONTERMINAL,
	CAST_MAIN,
	CAST_FLOAT_IDENTIFER,
	CAST_FLOAT_EXPRESSION,
	CAST_FLOAT_OPERATION,
	CAST_INTEGER_IDENTIFER,
	CAST_INTEGER_EXPRESSION,
	CAST_INTEGER_OPERATION,
	CAST_ARRAY_IDENTIFER,
	CAST_ARRAY_BRACKETS,
	CAST_ARRAY_ASSIGMENT,
	CAST_ARRAY_ELEMENT,
	CAST_ARRAY_DEREFERENCE,
	CAST_ARRAY_EXPRESSION,
	CAST_ARRAY_OPERATION,
	CAST_LIST,
	CAST_ASSIGMENT,
	CAST_BRACKETS,
	LEXICAL_ERROR,
	SYNTAX_ERROR,
};

class CommandStack {
private:
	stack<Token> _tokens;

public:
	void addToken(Token token) {
		_tokens.push(token);
	}

	vector<Token> getTokensByRule(int rule) {
		vector<Token> tokens;

		for (int i = 0; i < rule; i++) {
			auto token = _tokens.top();
			tokens.push_back(token);
			_tokens.pop();
		}

		return tokens;
	}
};

class StateStack {
private:
	stack<int> _stack;

public:
	StateStack() {
		_stack.push(0);
	}

	void pushState(int state) {
		_stack.push(state);
	}

	int currentState() {
		return _stack.top();
	}
```

```
	void acceptRule(int rule) {
		for (int i = 0; i < rule; i++) {
			_stack.pop();
		}
	}
};

class LrTable {
private:
	vector<vector<pair<string, int>>> _terminals;
	vector<vector<pair<Type, int>>> _nonTerminals;
	vector<vector<pair<State, int>>> _terminalRules;
	vector<vector<pair<State, int>>> _nonTerminalRules;

public:
	LrTable(vector<vector<pair<string, int>>> terminals,
vector<vector<pair<Type, int>>> nonTerminals, vector<vector<pair<State, int>>>
terminalRules, vector<vector<pair<State, int>>> nonTerminalRules) {
		_terminals = terminals;
		_nonTerminals = nonTerminals;
		_terminalRules = terminalRules;
		_nonTerminalRules = nonTerminalRules;
	}

	int nextState(int currentState, Token token) {
		if (isTerminal(currentState, token)) {
			auto terminalsOnState = _terminals[currentState];

			for (auto terminal : terminalsOnState) {
				if (token.key == terminal.first) {
					return terminal.second;
				}
			}
		}

		if (isNonTerminal(currentState, token)) {
			auto nonTerminalsOnState = _nonTerminals[currentState];

			for (auto nonTerminal : nonTerminalsOnState) {
				if (token.type == nonTerminal.first) {
					return nonTerminal.second;
				}
			}
		}
	}

	bool isTerminal(int currentState, Token token) {
		auto terminalsOnState = _terminals[currentState];

		for (auto terminal : terminalsOnState) {
			if (token.key == terminal.first) {
				return true;
			}
		}

		return false;
	}

	bool isNonTerminal(int currentState, Token token) {
		auto nonTerminalsOnState = _nonTerminals[currentState];

		for (auto nonTerminal : nonTerminalsOnState) {
			if (token.type == nonTerminal.first) {
				return true;
			}
		}
```

```
		return false;
	}

	bool isRule(int currentState, Token token) {
		if (isTerminal(currentState, token)) {
			auto terminalRulesOnState = _terminalRules[currentState];

			if (terminalRulesOnState.size() > 0) {
				return true;
			}
		}

		if (isNonTerminal(currentState, token)) {

			auto nonTerminalRulesOnState =
_nonTerminalRules[currentState];

			if (nonTerminalRulesOnState.size() > 0) {
				return true;
			}
		}

		return false;
	}

	pair<State, int> ruleByTokenOnState(int currentState, Token token) {
		if (isTerminal(currentState, token)) {
			return _terminalRules[currentState].front();
		}

		if (isNonTerminal(currentState, token)) {
			return _nonTerminalRules[currentState].front();
		}
	}

	bool nextIsTerminalOnState(int currentState) {
		if (_terminals[currentState].size() > 0) {
			return true;
		}

		return false;
	}

	vector<string> expectedOnState(int currentState) {
		vector<string> expected;

		auto terminalsOnState = _terminals[currentState];

		for (auto terminal : terminalsOnState) {
			expected.push_back(terminal.first);
		}

		return expected;
	}
};

class ParsingTable {
private:
	LrTable* _table;
	StateStack _stack;
	int _ruleNumber;

public:
	ParsingTable(LrTable* table) {
		_table = table;
	}
```

```
State nextStateByToken(Token token) {
	if (token.type == Type::ERROR) {
		return State::LEXICAL_ERROR;
	}

	switch (token.type)
	{
		case Type::IDENTIFER: {
			token.key = "identifer";
			break;
		}

		case Type::FLOAT: {
			token.key = "float identifer";
			break;
		}

		case Type::ARRAY: {
			token.key = "array identifer";
			break;
		}

		case Type::FLOAT_CONSTANT: {
			token.key = "float const";
			break;
		}

		case Type::INTEGER: {
			token.key = "int identifer";
			break;
		}

		case Type::INTEGER_CONSTANT: {
			token.key = "int const";
			break;
		}

		default:
			break;
	}

	auto state = _stack.currentState();

	if (_table->isTerminal(state, token)) {
		if (_table->isRule(state, token)) {
			auto rule = _table->ruleByTokenOnState(state, token);
			_ruleNumber = rule.second;
			_stack.acceptRule(_ruleNumber);
			_stack.pushState(_table->nextState(state, token));

			return rule.first;
		}

		_stack.pushState(_table->nextState(state, token));
		return State::SHIFT_TERMINAL;
	}

	if (_table->isNonTerminal(state, token)) {
		if (_table->isRule(state, token)) {
			auto rule = _table->ruleByTokenOnState(state, token);
			_ruleNumber = rule.second;
			_stack.acceptRule(_ruleNumber);
			_stack.pushState(_table->nextState(state, token));

			return rule.first;
		}
```

```
                    _stack.pushState(_table->nextState(state, token));
                    return State::SHIFT_NONTERMINAL;
                }

                return State::SYNTAX_ERROR;
        }

        int rule() {
                return _ruleNumber;
        }

        bool nextIsTerminal() {
                auto state = _stack.currentState();

                return _table->nextIsTerminalOnState(state);
        }

        vector<string> expected() {
                return _table->expectedOnState(_stack.currentState());
        }
};
```

Compiler.h

```
class Compiler {
public:
        virtual CompilationResult* compilationResult(vector<Token> tokens) = 0;
};

class BaseCompiler : public Compiler {
private:
        CommandStack _commandStack;
        ParsingTable* _table;

public:
        BaseCompiler(ParsingTable* table) {
                _table = table;
        }

        CompilationResult* compilationResult(vector<Token> tokens) override {
                HashTable<string, Type> identiferTable;
                HashTable<string, Type> initializedIdentifers;

                stack<Token> identifers;

                Token arrayOnState = Token(Type::ERROR, "error");
                int numberOfElements = 0;
                HashTable<string, int> arrays;

                string errorMessage = {};
                vector<Token> result;

                int numberOfCommand = 1;

                for (int i = 0; i < tokens.size();) {
                        if (tokens[i].type == Type::IDENTIFER) {
                                if (identiferTable.contains(tokens[i].key)) {
                                        tokens[i].type =
identiferTable.get(tokens[i].key);
                                }
                        }

                        auto state = _table->nextStateByToken(tokens[i]);

                        switch (state)
```

```
                        {
                                case State::SHIFT_TERMINAL: {
                                        _commandStack.addToken(tokens[i]);

                                        break;
                                }

                                case State::SHIFT_NONTERMINAL: {


                                        break;
                                }

                                case State::CAST_MAIN: {
                                        auto rule = _table->rule();

                                        auto newCommand =
_commandStack.getTokensByRule(rule);

                                        for (auto token : newCommand) {
                                                result.push_back(token);
                                        }

                                        break;
                                }

                                case State::CAST_FLOAT_IDENTIFER: {
                                        auto rule = _table->rule();

                                        auto identifer =
_commandStack.getTokensByRule(rule).front();

                                        if (identiferTable.contains(identifer.key)) {
                                                errorMessage += "syntax error:
re-declaring a variable {" + identifer.key + "} " + "in line " +
to_string(numberOfCommand) + "\n";
                                        }
                                        else {
                                                identifer.type = Type::FLOAT;
                                                identiferTable.add(identifer.key,
identifer.type);
                                        }

                                        identifers.push(identifer);
                                        result.push_back(identifer);

                                        break;
                                }

                                case State::CAST_FLOAT_EXPRESSION: {
                                        auto rule = _table->rule();

                                        auto floatInstance =
_commandStack.getTokensByRule(rule).front();

                                        if (floatInstance.type == Type::FLOAT) {
                                                if
(!identiferTable.contains(floatInstance.key)) {
                                                        errorMessage += "syntax error:
undeclared variable {" + floatInstance.key + "} " + "in line " +
to_string(numberOfCommand) + "\n";
                                                }
                                                if
(!initializedIdentifers.contains(floatInstance.key)) {
                                                        errorMessage += "syntax error:
uninitialized variable {" + floatInstance.key + "} " + "in line " +
to_string(numberOfCommand) + "\n";
```

```
                                        }
                                }

                                result.push_back(floatInstance);

                                break;
                        }

                        case State::CAST_FLOAT_OPERATION: {
                                auto rule = _table->rule();

                                auto expression =
_commandStack.getTokensByRule(rule);

                                auto floatInstance = expression[0];
                                auto operation = expression[1];

                                if (floatInstance.type == Type::FLOAT) {
                                        if
(!identiferTable.contains(floatInstance.key)) {
                                                errorMessage += "syntax error:
undeclared variable {" + floatInstance.key + "} " + "in line " +
to_string(numberOfCommand) + "\n";
                                        }
                                        if
(!initializedIdentifers.contains(floatInstance.key)) {
                                                errorMessage += "syntax error:
uninitialized variable {" + floatInstance.key + "} " + "in line " +
to_string(numberOfCommand) + "\n";
                                        }
                                }

                                result.push_back(floatInstance);
                                result.push_back(operation);

                                break;
                        }

                        case State::CAST_INTEGER_IDENTIFER: {
                                auto rule = _table->rule();

                                auto identifer =
_commandStack.getTokensByRule(rule).front();

                                if (identiferTable.contains(identifer.key)) {
                                        errorMessage += "syntax error:
re-declaring a variable {" + identifer.key + "} " + "in line " +
to_string(numberOfCommand) + "\n";
                                }
                                else {
                                        identifer.type = Type::INTEGER;
                                        identiferTable.add(identifer.key,
identifer.type);
                                }

                                identifers.push(identifer);
                                result.push_back(identifer);

                                break;
                        }

                        case State::CAST_INTEGER_EXPRESSION: {
                                auto rule = _table->rule();

                                auto integerInstance =
_commandStack.getTokensByRule(rule).front();
```

```
                                    if (integerInstance.type == Type::INTEGER) {
                                          if
(!identiferTable.contains(integerInstance.key)) {
                                                errorMessage += "syntax error:
undeclared variable {" + integerInstance.key + "} " + "in line " +
to_string(numberOfCommand) + "\n";
                                          }
                                          if
(!initializedIdentifers.contains(integerInstance.key)) {
                                                errorMessage += "syntax error:
uninitialized variable {" + integerInstance.key + "} " + "in line " +
to_string(numberOfCommand) + "\n";
                                          }
                                    }

                                    result.push_back(integerInstance);

                                    break;
                              }

                              case State::CAST_INTEGER_OPERATION: {
                                    auto rule = _table->rule();

                                    auto expression =
_commandStack.getTokensByRule(rule);

                                    auto integerInstance = expression[0];
                                    auto operation = expression[1];

                                    if (integerInstance.type == Type::INTEGER) {
                                          if
(!identiferTable.contains(integerInstance.key)) {
                                                errorMessage += "syntax error:
undeclared variable {" + integerInstance.key + "} " + "in line " +
to_string(numberOfCommand) + "\n";
                                          }
                                          if
(!initializedIdentifers.contains(integerInstance.key)) {
                                                errorMessage += "syntax error:
uninitialized variable {" + integerInstance.key + "} " + "in line " +
to_string(numberOfCommand) + "\n";
                                          }
                                    }

                                    result.push_back(integerInstance);
                                    result.push_back(operation);

                                    break;
                              }

                              case State::CAST_ARRAY_IDENTIFER: {
                                    auto rule = _table->rule();

                                    auto identifer =
_commandStack.getTokensByRule(rule).front();

                                    if (identiferTable.contains(identifer.key)) {
                                          errorMessage += "syntax error:
re-declaring a variable {" + identifer.key + "} " + "in line " +
to_string(numberOfCommand) + "\n";
                                    }
                                    else {
                                          identifer.type = Type::ARRAY;
                                          identiferTable.add(identifer.key,
identifer.type);
                                    }
```

```
                                    arrayOnState = identifer;
                                    arrays.add(arrayOnState.key, 0);
                                    identifers.push(identifer);
                                    result.push_back(identifer);

                                    break;
                             }

                             case State::CAST_ARRAY_ELEMENT: {

                                    auto rule = _table->rule();

                                    auto integerInstance =
_commandStack.getTokensByRule(rule).front();

                                    result.push_back(integerInstance);

                                    numberOfElements++;

                                    break;
                             }

                             case State::CAST_ARRAY_ASSIGMENT: {
                                    for (int i = 0; i < identifers.size(); i++) {
                                           auto identifer = identifers.top();
                                           identifers.pop();
                                           if
(!initializedIdentifers.contains(identifer.key)) {

initializedIdentifers.add(identifer.key, identifer.type);
                                           }
                                    }

                                    if (arrays.contains(arrayOnState.key)) {
                                           arrays.remove(arrayOnState.key);
                                    }

                                    arrays.add(arrayOnState.key, numberOfElements);
                                    numberOfElements = 0;

                                    result.push_back(Token(Type::ASSIGMENT, "="));

                                    break;


                                    break;
                             }

                             case State::CAST_ARRAY_EXPRESSION: {

                                    auto rule = _table->rule();

                                    auto arrayInstance =
_commandStack.getTokensByRule(rule).front();

                                    if (arrayInstance.type == Type::ARRAY) {
                                           if
(!identiferTable.contains(arrayInstance.key)) {
                                                  errorMessage += "syntax error:
undeclared variable {" + arrayInstance.key + "} " + "in line " +
to_string(numberOfCommand) + "\n";
                                           }
                                           if
(!initializedIdentifers.contains(arrayInstance.key)) {
                                                  errorMessage += "syntax error:
uninitialized variable {" + arrayInstance.key + "} " + "in line " +
to_string(numberOfCommand) + "\n";
```

```
                                        }
                                }

                                arrayOnState = arrayInstance;

                                result.push_back(arrayInstance);

                                break;
                        }

                        case State::CAST_ARRAY_OPERATION: {

                                auto rule = _table->rule();

                                auto expression =
_commandStack.getTokensByRule(rule);

                                auto bracket = expression[0];
                                auto operation = expression[1];

                                result.push_back(bracket);
                                result.push_back(operation);

                                break;
                        }

                        case State::CAST_ARRAY_DEREFERENCE: {
                                auto rule = _table->rule();

                                auto index =
_commandStack.getTokensByRule(rule).front();

                                auto numberOfElements =
arrays.get(arrayOnState.key);

                                if (std::stoi(index.key) < 0 ||
std::stoi(index.key) >= numberOfElements) {
                                        errorMessage += "index of {" +
arrayOnState.key + "} out of range. in line: " + std::to_string(numberOfCommand);
                                }

                                result.push_back(index);

                                break;
                        }

                        case State::CAST_BRACKETS: {
                                auto rule = _table->rule();

                                auto brackets =
_commandStack.getTokensByRule(rule);

                                for (auto token : brackets) {
                                        result.push_back(token);
                                }

                                break;
                        }

                        case State::CAST_ARRAY_BRACKETS: {
                                auto rule = _table->rule();

                                auto brackets =
_commandStack.getTokensByRule(rule);

                                break;
                        }
```

```
                        case State::CAST_LIST: {
                                auto rule = _table->rule();

                                _commandStack.getTokensByRule(rule);

                                break;
                        }

                        case State::CAST_ASSIGMENT: {
                                for (int i = 0; i < identifers.size(); i++) {
                                        auto identifer = identifers.top();
                                        identifers.pop();
                                        if
(!initializedIdentifers.contains(identifer.key)) {

initializedIdentifers.add(identifer.key, identifer.type);
                                        }
                                }

                                result.push_back(Token(Type::ASSIGMENT, "="));

                                break;
                        }

                        case State::END_OF_COMMAND: {
                                auto rule = _table->rule();

                                _commandStack.getTokensByRule(rule);

                                result.push_back(Token(Type::END_OF_COMMAND,
";"));


                                for (int i = 0; i < identifers.size(); i++) {
                                        identifers.pop();
                                }

                                numberOfCommand++;

                                break;
                        }

                        case State::LEXICAL_ERROR: {
                                errorMessage += "lexical error: unresolved
symbols chain in {" + tokens[i].key + "} " + "in line " +
to_string(numberOfCommand) + "\n";

                                break;
                        }

                        case State::SYNTAX_ERROR: {
                                auto expected = _table->expected();

                                if (tokens[i].type == Type::FLOAT ||
tokens[i].type == Type::FLOAT_CONSTANT) {
                                        //errorMessage += "syntax error: type
missmatch, expression is integer, but {" + tokens[i].key + "} is float\n";
                                }

                                if (tokens[i].type == Type::INTEGER ||
tokens[i].type == Type::INTEGER_CONSTANT || tokens[i].type == Type::ARRAY) {
                                        //errorMessage += "syntax error: type
missmatch, expression is float, but {" + tokens[i].key + "} is integer\n";
                                }

                                if (tokens[i].type == Type::IDENTIFER) {
                                        errorMessage += "syntax error: undeclared
```

```
variable {" + tokens[i].key + "} in line: " + to_string(numberOfCommand) + "\n";
                                    } else if (expected.front() == "identifer") {
                                          errorMessage += "syntax error:
re-declaring a variable {" + tokens[i].key + "} " + "in line " +
to_string(numberOfCommand) + "\n";
                                    }
                                    else {
                                          errorMessage += "syntax error: expected {
| ";
                                          for (auto word : expected) {
                                                errorMessage += word + " | ";
                                          }
                                          errorMessage += "} in line " +
to_string(numberOfCommand) + "\n";
                                    }

                                    return new ErrorMessage(errorMessage);
                              }

                              default:
                                    break;
                        }

                        if (_table->nextIsTerminal()) {
                              i++;
                        }
                  }

                  if (errorMessage.empty()) {
                        return new Code(identiferTable, arrays, result);
                  }

                  return new ErrorMessage(errorMessage);
            }
};
```