Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы трансляции

ОТЧЁТ

по лабораторной работе

на тему

Синтаксический анализатор

Выполнил

Студент гр. 053502

Макаро М.В.

Проверил

Ассистент кафедры информатики

Гриценко Н.Ю.

Минск 2023

СОДЕРЖАНИЕ

[1 Цель работы 3](#_Toc130992964)

[2 Краткие теоретические сведения 4](#_Toc130992965)

3 Примеры работы синтакчисеского анализато…………………………………….. 5

[3.1 Примеры получения абстрактного синтаксического дерева 5](#_Toc130992966)

[3.2 Синтаксические ошибки 8](#_Toc130992967)

[4Выводы 11](#_Toc130992968)

[ПРИЛОЖЕНИЕ А (обязательное) Код программы 12](#_Toc130992970)

1. **ЦЕЛЬ РАБОТЫ**

Освоение работы с существующими синтаксическими анализаторами. Разработать свой собственный синтаксический анализатор, выбранного подмножества языка программирования.

Построить синтаксическое дерево.

Определить минимум 4 возможных синтаксических ошибки и показать их корректное выявление.

Основной целью работы является написание сценариев, которые задают синтаксические правила для выбранного подмножества языка.

Код модуля для парсинга приведён в приложении А.

1. **КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Синтаксический анализ — это процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево). Обычно применяется совместно с лексическим анализом.

Синтаксический анализатор — это программа или часть программы, выполняющая синтаксический анализ.

Как правило, результатом синтаксического анализа является синтаксическое строение предложения, представленное либо в виде дерева зависимостей, либо в виде дерева составляющих, либо в виде некоторого сочетания первого и второго способов представления.

Таким образом на основе анализа выражений, состоящих из литералов, операторов и круглых скобок выполняется группирование токенов исходной программы в грамматические фразы, используемые для синтеза вывода.

1. **ПРИМЕРЫ РАБОТЫ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА**
   1. **Примеры получения абстрактного синтаксического дерева**

Пример абстрактного синтаксического дерава для первой программы (см. рисунок 1).

print("Enter number: \n");

int num = stoi(scan());

if (num < 0)

{

exit(-1);

}

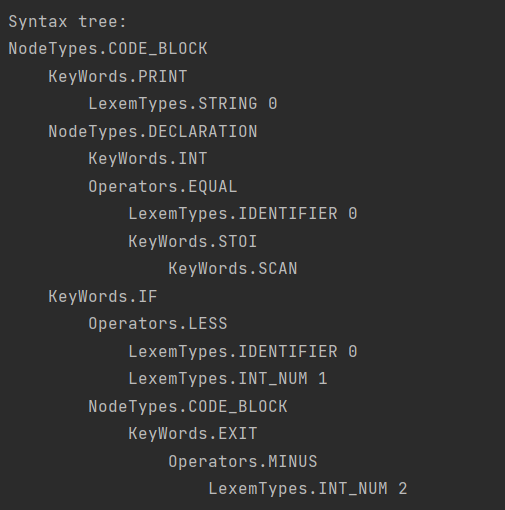


Рисунок 1 – Пример АСД для первой программы

Пример абстрактного синтаксического дерава для второй программы, представлен на рисунках 2, 3, 4, 5.

print(“Enter r: “);

double r = stod(scan());

print(“Enter 1 to calculate square, 2 - circle length: “);

int answer = stoi(scan());

if(answer != 1 && answer != 2)

exit(-1);

bool is\_square = answer == 1;

double res = 0;

double pi = 3.14;

if(is\_square)

res = pi \* r \* r;

else res = 2 \* pi \* r;

print(“Result: “ + to\_string(res) + “\n”);

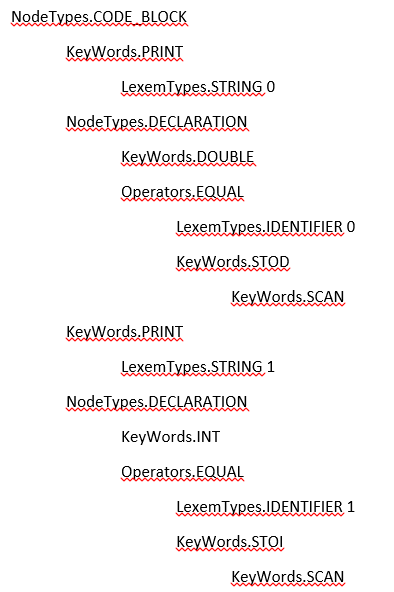


Рисунок 2 – Первая часть АСД для второй программы

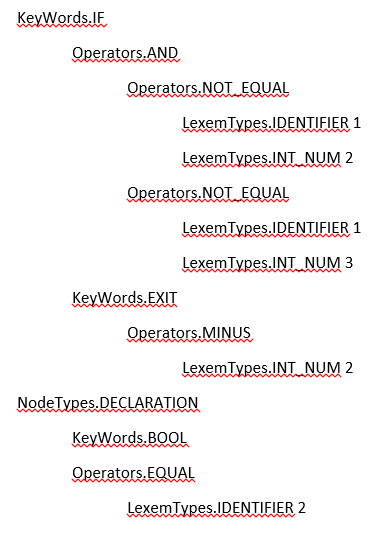


Рисунок 3 – Вторая часть АСД для второй программы

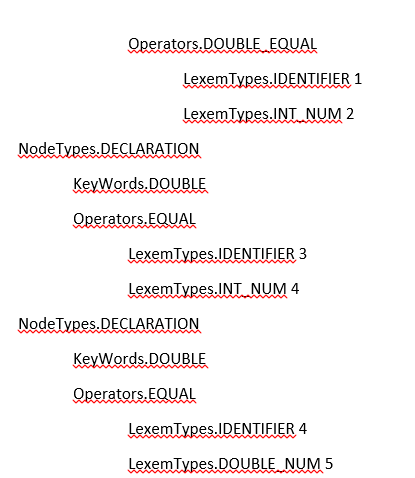


Рисунок 4 – Третья часть АСД для второй программы

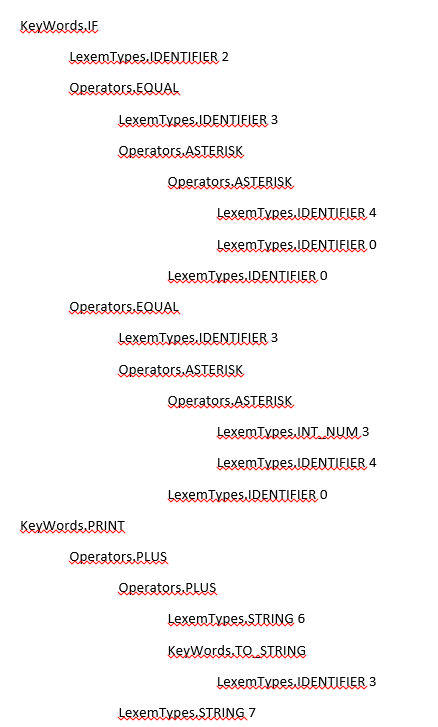


Рисунок 5 – Четвертая часть АСД для второй программы

* 1. **Синтаксические ошибки**

Ошибка ожидаемого токена (см. рисунок 6) — возникает, когда в потоке токенов встречается не тот токен, который ожидался.

Пример программы с ошибкой:

print("Hello")  
int i = 0;



Рисунок 6 – Пример ошибки ожидаемого токена

Ошибка невозможного сравнения (см. рисунок 7) — возникает, когда пытаются сравнить строку и число (сравнивать можно только строки со строками, числа с числами).

Пример программы с ошибкой:

bool b = "str" == (1 + 1);



Рисунок 7 – Пример ошибки невозможного сравнения

Ошибка повторного объявления (см. рисунок 8) — возникает, когда пытаются объявить переменную дважды в одном и том же блоке.

Пример программы с ошибкой:

int a = 3;  
double a = 4.13;



Рисунок 8 – Пример ошибки с двойным объявлением

Ошибка использования необъявленной переменной (см. рисунок 9) — возникает, если пытаются применить переменную, которая ещё не была объявлена.

Пример программы с ошибкой:

a = 3;



Рисунок 9 – Ошибка использования необъявленной переменной

Ошибка неправильного типа переменной (см. рисунок 10).

Входная программа:

bool b = true;  
double value = b + "hello";



Рисунок 10 – Пример ошибки неправильного типа переменной

Ошибка запрещённой инструкции (см. рисунок 11) — возникает, если попытаться использовать инструкцию внутри блока, в котором её не должно быть (например, break внутри if).

Пример программы с ошибкой:

if(1 < 2)  
{  
 break;  
}



Рисунок 11 – Пример ошибки запрещённой инструкции

1. **ВЫВОДЫ**

Таким образом, в ходе лабораторной работы было изучено понятие синтаксического анализа в теории трансляции. Был разработан собственный синтаксический анализатор выбранного подмножества языка программирования

ПРИЛОЖЕНИЕ А

**(обязательное)**

Код программы

from common import \*  
from tables import \*  
  
  
def is\_variable\_type(key\_word: KeyWords) -> bool:  
 return key\_word == KeyWords.INT or key\_word == KeyWords.DOUBLE or \  
 key\_word == KeyWords.BOOL or key\_word == KeyWords.STRING or key\_word == KeyWords.VOID  
  
  
def is\_addop(op: Operators) -> bool:  
 return op == Operators.PLUS or op == Operators.MINUS  
  
  
def is\_mulop(op: Operators) -> bool:  
 return op in (Operators.ASTERISK, Operators.SLASH, Operators.PERCENT)  
  
  
class ExpressionTypes(Enum):  
 ARITHMETIC = 0,  
 BOOL = 1,  
 STRING = 2  
  
  
class ComparisonTypes(Enum):  
 ARITHMETIC = 0,  
 STRING = 1  
  
  
class ParserError(Exception):  
 def \_\_init\_\_(self, text: str, fname: str, line\_num: int, ch\_num: int):  
 self.\_txt = 'File "' + fname + '", line ' + str(line\_num) + ' col ' + str(ch\_num) + ': ' + text  
 super().\_\_init\_\_(self.\_txt)  
  
  
class ExpectedError(ParserError):  
 def \_\_init\_\_(self, expected: str, fname: str, line\_num: int, ch\_num: int):  
 super().\_\_init\_\_(f"{expected} expected", fname, line\_num, ch\_num)  
  
  
class UsingBeforeDeclarationError(ParserError):  
 def \_\_init\_\_(self, var\_name: str, fname: str, line\_num: int, ch\_num: int):  
 super().\_\_init\_\_(f"Varible {var\_name} using before declaration", fname, line\_num, ch\_num)  
  
  
class DoubleDeclarationError(ParserError):  
 def \_\_init\_\_(self, var\_name: str, fname: str, line\_num: int, ch\_num: int):  
 super().\_\_init\_\_(f"Double declaration of variable {var\_name}", fname, line\_num, ch\_num)  
  
  
class NodeTypes(Enum):  
 COMMON = 0,  
 DECLARATION = 1,  
 CODE\_BLOCK = 2  
  
  
class Node:  
 def \_\_init\_\_(self, lexeme: LexTableItem or None, type: NodeTypes = NodeTypes.COMMON):  
 self.\_lexeme = lexeme  
 self.\_childs = []  
 self.\_type = type  
  
 def add\_child(self, node) -> None:  
 self.\_childs.append(node)  
  
 def get\_childs(self) -> list:  
 return self.\_childs  
  
 def get\_lexeme(self) -> LexTableItem:  
 return self.\_lexeme  
  
 def \_\_str\_\_(self):  
 if self.\_lexeme is None:  
 return str(self.\_type)  
 elif self.\_lexeme.type == LexemTypes.IDENTIFIER or \  
 self.\_lexeme.type in (LexemTypes.INT\_NUM, LexemTypes.DOUBLE\_NUM, LexemTypes.STRING):  
 return str(self.\_lexeme.type) + " " + str(self.\_lexeme.value)  
 else:  
 return str(self.\_lexeme.value)  
  
  
def print\_tree(root, depth: int = 0):  
 if root is None:  
 return  
  
 print('\t' \* depth + str(root))  
 for child in root.get\_childs():  
 print\_tree(child, depth + 1)  
  
  
class Parser:  
 def \_\_init\_\_(self, fname: str, lexemes: list, literal\_table: LiteralTable, variable\_table: list):  
 self.\_fname = fname  
 self.\_lexemes = lexemes  
 self.\_literal\_table = literal\_table  
 self.\_variable\_table = variable\_table  
 self.\_curr\_lex\_index = 0  
 self.\_block\_level = 0  
 self.\_block\_id = 0  
 self.\_scope\_stack = [(self.\_block\_level, self.\_block\_id)]  
 self.\_root = Node(None, NodeTypes.CODE\_BLOCK)  
 self.\_nesting\_while = 0  
 self.parse()  
  
 def print\_syntax\_tree(self) -> None:  
 print\_tree(self.\_root)  
  
 def \_is\_break\_available(self) -> bool:  
 return self.\_nesting\_while > 0  
  
 def \_is\_continue\_available(self) -> bool:  
 return self.\_nesting\_while > 0  
  
 def \_go\_to\_next\_lexeme(self) -> None:  
 self.\_curr\_lex\_index += 1  
  
 def \_is\_match\_cur\_lexeme(self, type) -> bool:  
 if type == LexemTypes.IDENTIFIER or type == LexemTypes.INT\_NUM or \  
 type == LexemTypes.DOUBLE\_NUM or type == LexemTypes.STRING:  
 return type == self.\_lexemes[self.\_curr\_lex\_index].type  
 else:  
 return type == self.\_lexemes[self.\_curr\_lex\_index].value  
  
 def \_expect\_key\_word(self, key\_word: KeyWords) -> None:  
 lexeme = self.\_get\_curr\_lexeme()  
 if lexeme.value != key\_word:  
 raise ExpectedError(str(key\_word), self.\_fname, lexeme.line\_num, lexeme.col\_num)  
  
 def \_expect\_delimiter(self, delimiter: Delimiters) -> None:  
 lexeme = self.\_get\_curr\_lexeme()  
 if lexeme.value != delimiter:  
 raise ExpectedError(str(delimiter), self.\_fname, lexeme.line\_num, lexeme.col\_num)  
  
 def \_expect\_operator(self, operator: Operators) -> None:  
 lexeme = self.\_get\_curr\_lexeme()  
 if lexeme.value != operator:  
 raise ExpectedError(str(operator), self.\_fname, lexeme.line\_num, lexeme.col\_num)  
  
 def \_expect\_var\_type(self, lexeme: LexTableItem, type: VariableTypes or tuple) -> None:  
 if lexeme.type != LexemTypes.IDENTIFIER:  
 raise ExpectedError(str(type) + " variable", self.\_fname, lexeme.line\_num, lexeme.col\_num)  
 var = self.\_get\_variable(lexeme)  
 if isinstance(type, VariableTypes):  
 if var.type != type:  
 raise ExpectedError(str(type) + " variable", self.\_fname, lexeme.line\_num, lexeme.col\_num)  
 else:  
 if var.type not in type:  
 raise ExpectedError("One of the following variable types: " + str(type), self.\_fname,  
 lexeme.line\_num, lexeme.col\_num)  
  
 def \_expect\_identifier(self) -> None:  
 lexeme = self.\_get\_curr\_lexeme()  
 if lexeme.type != LexemTypes.IDENTIFIER:  
 raise ExpectedError("identifier", self.\_fname, lexeme.line\_num, lexeme.col\_num)  
  
 def \_expect\_stoid(self) -> None:  
 lexeme = self.\_get\_curr\_lexeme()  
 if not (lexeme.value in (KeyWords.STOI, KeyWords.STOD)):  
 raise ExpectedError("stoi or stod", self.\_fname, lexeme.line\_num, lexeme.col\_num)  
  
 def \_expect\_comparison\_operator(self) -> None:  
 lexeme = self.\_get\_curr\_lexeme()  
 if not (lexeme.value in (Operators.DOUBLE\_EQUAL, Operators.NOT\_EQUAL, Operators.LESS,  
 Operators.LESS\_OR\_EQUAL, Operators.GREATER, Operators.GREATER\_OR\_EQUAL)):  
 raise ExpectedError("One of the comparison operators", self.\_fname, lexeme.line\_num, lexeme.col\_num)  
  
 def \_expect\_bool\_literal(self) -> None:  
 lexeme = self.\_get\_curr\_lexeme()  
 if lexeme.value != KeyWords.TRUE and lexeme.value != KeyWords.FALSE:  
 raise ExpectedError("Bool literal", self.\_fname, lexeme.line\_num, lexeme.col\_num)  
  
 def \_expect\_string\_literal(self) -> None:  
 lexeme = self.\_get\_curr\_lexeme()  
 if lexeme.type != LexemTypes.STRING:  
 raise ExpectedError("String literal", self.\_fname, lexeme.line\_num, lexeme.col\_num)  
  
 def \_are\_lexemes\_remaining(self) -> bool:  
 return self.\_curr\_lex\_index < len(self.\_lexemes)  
  
 def \_get\_curr\_lexeme(self) -> LexTableItem:  
 if not self.\_are\_lexemes\_remaining():  
 lexeme = self.\_lexemes[self.\_curr\_lex\_index - 1]  
 raise ParserError("Unexpected end of file", self.\_fname, lexeme.col\_num, lexeme.col\_num)  
 return self.\_lexemes[self.\_curr\_lex\_index]  
  
 def \_get\_variable(self, lexeme: LexTableItem) -> VariableTableItem:  
 return self.\_variable\_table[lexeme.value]  
  
 def \_enter\_block(self) -> None:  
 self.\_block\_level += 1  
 self.\_block\_id += 1  
 self.\_scope\_stack.append((self.\_block\_level, self.\_block\_id))  
  
 def \_exit\_block(self) -> None:  
 self.\_block\_level -= 1  
 self.\_scope\_stack.pop()  
  
 def \_parse\_block\_code(self) -> Node:  
 self.\_expect\_delimiter(Delimiters.OPEN\_BRACES)  
 self.\_go\_to\_next\_lexeme()  
 self.\_enter\_block()  
  
 code\_block\_node = Node(None, NodeTypes.CODE\_BLOCK)  
  
 while self.\_are\_lexemes\_remaining() and not self.\_is\_match\_cur\_lexeme(Delimiters.CLOSE\_BRACES):  
 code\_block\_node.add\_child(self.\_parse\_statement())  
  
 self.\_expect\_delimiter(Delimiters.CLOSE\_BRACES)  
 self.\_go\_to\_next\_lexeme()  
 self.\_exit\_block()  
  
 return code\_block\_node  
  
 def \_parse\_declare\_identifier(self, var\_type: VariableTypes) -> Node:  
 self.\_expect\_identifier()  
 lexeme = self.\_get\_curr\_lexeme()  
 node = Node(lexeme)  
 curr\_var = self.\_get\_variable(lexeme)  
  
 block\_level = self.\_scope\_stack[-1][0]  
 block\_id = self.\_scope\_stack[-1][1]  
  
 if curr\_var.type != VariableTypes.UNKNOWN:  
 for var in self.\_variable\_table:  
 if curr\_var.name == var.name and block\_id == var.block\_id:  
 raise DoubleDeclarationError(curr\_var.name, self.\_fname, lexeme.line\_num, lexeme.col\_num)  
  
 self.\_variable\_table.append(VariableTableItem(curr\_var.name, var\_type, block\_level, block\_id))  
 curr\_var = self.\_variable\_table[-1]  
 lexeme.value = len(self.\_variable\_table) - 1  
  
 curr\_var.type = var\_type  
 curr\_var.block\_level = block\_level  
 curr\_var.block\_id = block\_id  
  
 self.\_go\_to\_next\_lexeme()  
 return node  
  
 def \_parse\_using\_identifier(self) -> Node:  
 self.\_expect\_identifier()  
 lexeme = self.\_get\_curr\_lexeme()  
 node = Node(lexeme)  
  
 var = self.\_get\_variable(lexeme)  
 if var.type == VariableTypes.UNKNOWN:  
 raise UsingBeforeDeclarationError(var.name, self.\_fname, lexeme.line\_num, lexeme.col\_num)  
  
 var\_real\_id = -1  
 for scope in reversed(self.\_scope\_stack):  
 block\_level = scope[0]  
 block\_id = scope[1]  
 searched\_var = VariableTableItem(var.name, var.type, block\_level, block\_id)  
  
 try:  
 var\_real\_id = self.\_variable\_table.index(searched\_var)  
 break  
 except ValueError:  
 pass  
  
 if var\_real\_id < 0:  
 raise UsingBeforeDeclarationError(var.name, self.\_fname, lexeme.line\_num, lexeme.col\_num)  
  
 lexeme.value = var\_real\_id  
 self.\_go\_to\_next\_lexeme()  
 return node  
  
 def \_parse\_operator(self, op: Operators):  
 self.\_expect\_operator(op)  
 op\_node = Node(self.\_get\_curr\_lexeme())  
 self.\_go\_to\_next\_lexeme()  
 return op\_node  
  
 def \_parse\_stoid(self) -> Node:  
 self.\_expect\_stoid()  
 lexeme = self.\_get\_curr\_lexeme()  
 stoid\_node = Node(lexeme)  
 self.\_go\_to\_next\_lexeme()  
  
 self.\_expect\_delimiter(Delimiters.OPEN\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
 string\_node = self.\_parse\_string\_expression()  
 self.\_expect\_delimiter(Delimiters.CLOSE\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
  
 stoid\_node.add\_child(string\_node)  
 return stoid\_node  
  
 def \_parse\_terminal(self) -> Node:  
 lexeme = self.\_get\_curr\_lexeme()  
  
 if self.\_is\_match\_cur\_lexeme(Delimiters.OPEN\_PARENTHESIS):  
 self.\_go\_to\_next\_lexeme()  
 node = self.\_parse\_arithmetic\_expression()  
 self.\_expect\_delimiter(Delimiters.CLOSE\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
 elif lexeme.type in (LexemTypes.INT\_NUM, LexemTypes.DOUBLE\_NUM):  
 node = Node(lexeme)  
 self.\_go\_to\_next\_lexeme()  
 elif lexeme.value in (KeyWords.STOI, KeyWords.STOD):  
 node = self.\_parse\_stoid()  
 elif lexeme.type == LexemTypes.IDENTIFIER:  
 node = self.\_parse\_using\_identifier()  
 self.\_expect\_var\_type(node.get\_lexeme(), (VariableTypes.INT, VariableTypes.DOUBLE))  
 else:  
 raise ExpectedError("Number", self.\_fname, lexeme.line\_num, lexeme.col\_num)  
  
 return node  
  
 def \_parse\_terminals\_and\_mul\_ops(self) -> Node:  
 lhs\_node = self.\_parse\_terminal()  
  
 while self.\_are\_lexemes\_remaining() and is\_mulop(self.\_get\_curr\_lexeme().value):  
 lexeme = self.\_get\_curr\_lexeme()  
 op\_node = self.\_parse\_operator(lexeme.value)  
 rhs\_node = self.\_parse\_terminal()  
 op\_node.add\_child(lhs\_node)  
 op\_node.add\_child(rhs\_node)  
 lhs\_node = op\_node  
  
 return lhs\_node  
  
 def \_parse\_unary\_operation\_and\_terminal(self) -> Node:  
 lexeme = self.\_get\_curr\_lexeme()  
 op\_node = None  
 if is\_addop(lexeme.value):  
 op\_node = self.\_parse\_operator(lexeme.value)  
  
 node = self.\_parse\_terminals\_and\_mul\_ops()  
  
 if op\_node is None:  
 return node  
  
 op\_node.add\_child(node)  
 return op\_node  
  
 def \_parse\_arithmetic\_expression(self) -> Node:  
 lhs\_node = self.\_parse\_unary\_operation\_and\_terminal()  
  
 while self.\_are\_lexemes\_remaining() and is\_addop(self.\_get\_curr\_lexeme().value):  
 lexeme = self.\_get\_curr\_lexeme()  
 op\_node = self.\_parse\_operator(lexeme.value)  
 rhs\_node = self.\_parse\_terminals\_and\_mul\_ops()  
 op\_node.add\_child(lhs\_node)  
 op\_node.add\_child(rhs\_node)  
 lhs\_node = op\_node  
  
 return lhs\_node  
  
 def \_parse\_bool\_literal(self) -> Node:  
 self.\_expect\_bool\_literal()  
 node = Node(self.\_get\_curr\_lexeme())  
 self.\_go\_to\_next\_lexeme()  
 return node  
  
 def \_parse\_comparison\_term(self) -> (Node, ComparisonTypes):  
 lexeme = self.\_get\_curr\_lexeme()  
 old\_lex\_index = self.\_curr\_lex\_index  
  
 try:  
 node = self.\_parse\_arithmetic\_expression()  
 return node, ComparisonTypes.ARITHMETIC  
 except ExpectedError:  
 self.\_curr\_lex\_index = old\_lex\_index  
 try:  
 node = self.\_parse\_string\_expression()  
 return node, ComparisonTypes.STRING  
 except ExpectedError:  
 raise ExpectedError("arithmetic or string expression", self.\_fname, lexeme.line\_num, lexeme.col\_num)  
  
 def \_parse\_comparison(self) -> Node:  
 lexeme = self.\_get\_curr\_lexeme()  
 lhs\_node, lhs\_type = self.\_parse\_comparison\_term()  
 self.\_expect\_comparison\_operator()  
 op\_node = Node(self.\_get\_curr\_lexeme())  
 self.\_go\_to\_next\_lexeme()  
 rhs\_node, rhs\_type = self.\_parse\_comparison\_term()  
  
 if lhs\_type != rhs\_type:  
 raise ParserError("Can't compare " + str(lhs\_type) + " and " + str(rhs\_type),  
 self.\_fname, lexeme.line\_num, lexeme.col\_num)  
  
 op\_node.add\_child(lhs\_node)  
 op\_node.add\_child(rhs\_node)  
 return op\_node  
  
 def \_parse\_bool\_term(self) -> Node:  
 lexeme = self.\_get\_curr\_lexeme()  
 if lexeme.type == LexemTypes.IDENTIFIER and self.\_get\_variable(lexeme).type == VariableTypes.BOOL:  
 node = self.\_parse\_using\_identifier()  
 elif self.\_is\_match\_cur\_lexeme(Delimiters.OPEN\_PARENTHESIS):  
 old\_lex\_index = self.\_curr\_lex\_index  
 self.\_go\_to\_next\_lexeme()  
  
 # Can be bool experession or a comparison.  
 try:  
 node = self.\_parse\_bool\_expression()  
 self.\_expect\_delimiter(Delimiters.CLOSE\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
 except ParserError:  
 self.\_curr\_lex\_index = old\_lex\_index  
 node = self.\_parse\_comparison()  
 elif lexeme.value in (KeyWords.TRUE, KeyWords.FALSE):  
 node = self.\_parse\_bool\_literal()  
 else:  
 node = self.\_parse\_comparison()  
  
 return node  
  
 def \_parse\_bool\_term\_with\_possible\_not(self) -> Node:  
 was\_not = False  
 not\_node = None  
 if self.\_is\_match\_cur\_lexeme(Operators.NOT):  
 not\_node = self.\_parse\_operator(Operators.NOT)  
 was\_not = True  
  
 node = self.\_parse\_bool\_term()  
 if not was\_not:  
 return node  
  
 not\_node.add\_child(node)  
 return not\_node  
  
 def \_parse\_bool\_and(self) -> Node:  
 lhs\_node = self.\_parse\_bool\_term\_with\_possible\_not()  
  
 while self.\_are\_lexemes\_remaining() and self.\_is\_match\_cur\_lexeme(Operators.AND):  
 and\_node = self.\_parse\_operator(Operators.AND)  
 rhs\_node = self.\_parse\_bool\_term\_with\_possible\_not()  
 and\_node.add\_child(lhs\_node)  
 and\_node.add\_child(rhs\_node)  
 lhs\_node = and\_node  
  
 return lhs\_node  
  
 def \_parse\_bool\_expression(self) -> Node:  
 lhs\_node = self.\_parse\_bool\_and()  
  
 while self.\_are\_lexemes\_remaining() and self.\_is\_match\_cur\_lexeme(Operators.OR):  
 or\_node = self.\_parse\_operator(Operators.OR)  
 rhs\_node = self.\_parse\_bool\_and()  
 or\_node.add\_child(lhs\_node)  
 or\_node.add\_child((rhs\_node))  
 lhs\_node = or\_node  
  
 return lhs\_node  
  
 def \_parse\_to\_string(self) -> Node:  
 self.\_expect\_key\_word(KeyWords.TO\_STRING)  
 to\_string\_node = Node(self.\_get\_curr\_lexeme())  
 self.\_go\_to\_next\_lexeme()  
 self.\_expect\_delimiter(Delimiters.OPEN\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
 # TODO: Add bool\_expression.  
 expr\_node = self.\_parse\_arithmetic\_expression()  
 self.\_expect\_delimiter(Delimiters.CLOSE\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
 to\_string\_node.add\_child(expr\_node)  
 return to\_string\_node  
  
 def \_parse\_scan(self) -> Node:  
 self.\_expect\_key\_word(KeyWords.SCAN)  
 scan\_node = Node(self.\_get\_curr\_lexeme())  
 self.\_go\_to\_next\_lexeme()  
 self.\_expect\_delimiter(Delimiters.OPEN\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
 self.\_expect\_delimiter(Delimiters.CLOSE\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
 return scan\_node  
  
 def \_parse\_string\_terminal(self) -> Node:  
 lexeme = self.\_get\_curr\_lexeme()  
 if lexeme.type == LexemTypes.IDENTIFIER:  
 node = self.\_parse\_using\_identifier()  
 self.\_expect\_var\_type(node.get\_lexeme(), VariableTypes.STRING)  
 elif lexeme.value == KeyWords.TO\_STRING:  
 node = self.\_parse\_to\_string()  
 elif lexeme.value == KeyWords.SCAN:  
 node = self.\_parse\_scan()  
 else:  
 self.\_expect\_string\_literal()  
 node = Node(lexeme)  
 self.\_go\_to\_next\_lexeme()  
  
 return node  
  
 def \_parse\_string\_expression(self):  
 lhs\_node = self.\_parse\_string\_terminal()  
  
 while self.\_are\_lexemes\_remaining() and self.\_is\_match\_cur\_lexeme(Operators.PLUS):  
 op\_node = self.\_parse\_operator(Operators.PLUS)  
 rhs\_node = self.\_parse\_string\_terminal()  
 op\_node.add\_child(lhs\_node)  
 op\_node.add\_child(rhs\_node)  
 lhs\_node = op\_node  
  
 return lhs\_node  
  
 def \_parse\_print(self) -> Node:  
 self.\_expect\_key\_word(KeyWords.PRINT)  
 print\_node = Node(self.\_get\_curr\_lexeme())  
 self.\_go\_to\_next\_lexeme()  
 self.\_expect\_delimiter(Delimiters.OPEN\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
 string\_expr\_node = self.\_parse\_string\_expression()  
 self.\_expect\_delimiter(Delimiters.CLOSE\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
 self.\_expect\_delimiter(Delimiters.SEMICOLON)  
 self.\_go\_to\_next\_lexeme()  
 print\_node.add\_child(string\_expr\_node)  
 return print\_node  
  
 def \_parse\_var\_type(self) -> (Node, VariableTypes):  
 type\_lexeme = self.\_get\_curr\_lexeme()  
 type\_node = Node(type\_lexeme)  
 var\_type = VariableTypes.UNKNOWN  
 if type\_lexeme.value == KeyWords.INT:  
 var\_type = VariableTypes.INT  
 elif type\_lexeme.value == KeyWords.DOUBLE:  
 var\_type = VariableTypes.DOUBLE  
 elif type\_lexeme.value == KeyWords.STRING:  
 var\_type = VariableTypes.STRING  
 elif type\_lexeme.value == KeyWords.BOOL:  
 var\_type = VariableTypes.BOOL  
 else:  
 raise ParserError("Unknown variable type", self.\_fname, type\_lexeme.line\_num, type\_lexeme.col\_num)  
  
 self.\_go\_to\_next\_lexeme()  
 return type\_node, var\_type  
  
 def \_parse\_assignment(self, identifier\_node: Node) -> Node:  
 self.\_expect\_operator(Operators.EQUAL)  
 equal\_lexeme = self.\_get\_curr\_lexeme()  
 equal\_node = Node(equal\_lexeme)  
 equal\_node.add\_child(identifier\_node)  
 self.\_go\_to\_next\_lexeme()  
  
 identifier\_lexeme = identifier\_node.get\_lexeme()  
 var = self.\_get\_variable(identifier\_lexeme)  
 var\_type = var.type  
  
 if var\_type in (VariableTypes.INT, VariableTypes.DOUBLE):  
 rhs\_node = self.\_parse\_arithmetic\_expression()  
 elif var\_type == VariableTypes.STRING:  
 rhs\_node = self.\_parse\_string\_expression()  
 elif var\_type == VariableTypes.BOOL:  
 rhs\_node = self.\_parse\_bool\_expression()  
 else:  
 ParserError("Unknown type of identifier.", self.\_fname,  
 identifier\_lexeme.line\_num, identifier\_lexeme.col\_num)  
  
 equal\_node.add\_child(rhs\_node)  
 return equal\_node  
  
 def \_parse\_optional\_initialization(self, identifier\_node: Node) -> Node:  
 if not self.\_are\_lexemes\_remaining() or not self.\_is\_match\_cur\_lexeme(Operators.EQUAL):  
 return identifier\_node  
 return self.\_parse\_assignment(identifier\_node)  
  
 def \_parse\_var\_declaration(self) -> Node:  
 type\_node, var\_type = self.\_parse\_var\_type()  
 identifier\_node = self.\_parse\_declare\_identifier(var\_type)  
 declaration\_node = Node(None, NodeTypes.DECLARATION)  
 declaration\_node.add\_child(type\_node)  
 # TODO: Доделать self.\_parse\_optional\_initialization()  
 ident\_or\_eq\_node = self.\_parse\_optional\_initialization(identifier\_node)  
 declaration\_node.add\_child(ident\_or\_eq\_node)  
 self.\_expect\_delimiter(Delimiters.SEMICOLON)  
 self.\_go\_to\_next\_lexeme()  
 return declaration\_node  
  
 def \_parse\_if(self) -> Node:  
 self.\_expect\_key\_word(KeyWords.IF)  
 if\_node = Node(self.\_get\_curr\_lexeme())  
 self.\_go\_to\_next\_lexeme()  
 self.\_expect\_delimiter(Delimiters.OPEN\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
 condition\_node = self.\_parse\_bool\_expression()  
 self.\_expect\_delimiter(Delimiters.CLOSE\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
 if\_statement\_node = self.\_parse\_statement()  
 if\_node.add\_child(condition\_node)  
 if\_node.add\_child(if\_statement\_node)  
  
 if self.\_are\_lexemes\_remaining() and self.\_is\_match\_cur\_lexeme(KeyWords.ELSE):  
 self.\_go\_to\_next\_lexeme()  
 else\_statement\_node = self.\_parse\_statement()  
 if\_node.add\_child(else\_statement\_node)  
  
 return if\_node  
  
 def \_parse\_while(self) -> Node:  
 self.\_expect\_key\_word(KeyWords.WHILE)  
 while\_node = Node(self.\_get\_curr\_lexeme())  
 self.\_go\_to\_next\_lexeme()  
 self.\_expect\_delimiter(Delimiters.OPEN\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
 condition\_node = self.\_parse\_bool\_expression()  
 while\_node.add\_child(condition\_node)  
 self.\_expect\_delimiter(Delimiters.CLOSE\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
 if self.\_is\_match\_cur\_lexeme(Delimiters.SEMICOLON):  
 return while\_node  
  
 self.\_nesting\_while += 1  
 statement\_node = self.\_parse\_statement()  
 self.\_nesting\_while -= 1  
 while\_node.add\_child(statement\_node)  
 return while\_node  
  
 def \_parse\_exit(self) -> Node:  
 self.\_expect\_key\_word(KeyWords.EXIT)  
 exit\_node = Node(self.\_get\_curr\_lexeme())  
 self.\_go\_to\_next\_lexeme()  
 self.\_expect\_delimiter(Delimiters.OPEN\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
 exit\_code\_node = self.\_parse\_arithmetic\_expression()  
 self.\_expect\_delimiter(Delimiters.CLOSE\_PARENTHESIS)  
 self.\_go\_to\_next\_lexeme()  
 self.\_expect\_delimiter(Delimiters.SEMICOLON)  
 self.\_go\_to\_next\_lexeme()  
  
 exit\_node.add\_child(exit\_code\_node)  
 return exit\_node  
  
 def \_parse\_break(self) -> Node:  
 lexeme = self.\_get\_curr\_lexeme()  
 if not self.\_is\_break\_available():  
 raise ParserError("break is not available in this context.", self.\_fname, lexeme.line\_num, lexeme.col\_num)  
 self.\_expect\_key\_word(KeyWords.BREAK)  
 node = Node(lexeme)  
 self.\_go\_to\_next\_lexeme()  
 self.\_expect\_delimiter(Delimiters.SEMICOLON)  
 self.\_go\_to\_next\_lexeme()  
 return node  
  
 def \_parse\_continue(self) -> Node:  
 lexeme = self.\_get\_curr\_lexeme()  
 if not self.\_is\_continue\_available():  
 raise ParserError("continue is not available in this context.", self.\_fname, lexeme.line\_num, lexeme.col\_num)  
 self.\_expect\_key\_word(KeyWords.CONTINUE)  
 node = Node(lexeme)  
 self.\_go\_to\_next\_lexeme()  
 self.\_expect\_delimiter(Delimiters.SEMICOLON)  
 self.\_go\_to\_next\_lexeme()  
 return node  
  
 def \_parse\_statement(self) -> Node:  
 lexeme = self.\_get\_curr\_lexeme()  
 if lexeme.type == LexemTypes.KEY\_WORD:  
 if lexeme.value == KeyWords.PRINT:  
 return self.\_parse\_print()  
 elif is\_variable\_type(lexeme.value):  
 return self.\_parse\_var\_declaration()  
 elif lexeme.value == KeyWords.IF:  
 return self.\_parse\_if()  
 elif lexeme.value == KeyWords.WHILE:  
 return self.\_parse\_while()  
 elif lexeme.value == KeyWords.EXIT:  
 return self.\_parse\_exit()  
 elif lexeme.value == KeyWords.BREAK:  
 return self.\_parse\_break()  
 elif lexeme.value == KeyWords.CONTINUE:  
 return self.\_parse\_continue()  
 else:  
 raise ParserError("Unexpected lexeme: " + str(lexeme), self.\_fname, lexeme.line\_num, lexeme.col\_num)  
 elif lexeme.type == LexemTypes.DELIMITER:  
 if lexeme.value == Delimiters.OPEN\_BRACES:  
 return self.\_parse\_block\_code()  
 else:  
 raise ParserError("Unexpected lexeme: " + str(lexeme), self.\_fname, lexeme.line\_num, lexeme.col\_num)  
 elif lexeme.type == LexemTypes.IDENTIFIER:  
 identifier\_node = self.\_parse\_using\_identifier()  
 node = self.\_parse\_assignment(identifier\_node)  
 self.\_expect\_delimiter(Delimiters.SEMICOLON)  
 self.\_go\_to\_next\_lexeme()  
 return node  
 else:  
 raise ParserError("Unexpected lexeme: " + str(lexeme), self.\_fname, lexeme.line\_num, lexeme.col\_num)  
  
 def parse(self):  
 while self.\_are\_lexemes\_remaining():  
 node = self.\_parse\_statement()  
 self.\_root.add\_child(node)