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

Юрьев В.А.

Проверил

Ассистент кафедры информатики

Гриценко Н.Ю.

Минск 2023

СОДЕРЖАНИЕ

[1 Цель работы 3](#_Toc132985495)

[2 Результаты 4](#_Toc132985496)

[2.1 Построение синтаксического дерева 4](#_Toc132985497)

[2.2 Обнаружение синтаксических ошибок 5](#_Toc132985498)

[Приложение А Код программы 8](#_Toc132985499)

# **ЦЕЛЬ РАБОТЫ**

Разработка синтаксического анализатора языка программирования, определенного в лабораторной работе 1. Построение синтаксического дерева. Показать корректность работы анализатора допущением синтаксических ошибок.

# **РЕЗУЛЬТАТЫ**

## **Построение синтаксического дерева**

В качестве тестовой программы был использован алгоритм для возведения целого числа в степень (рисунок 2.1).

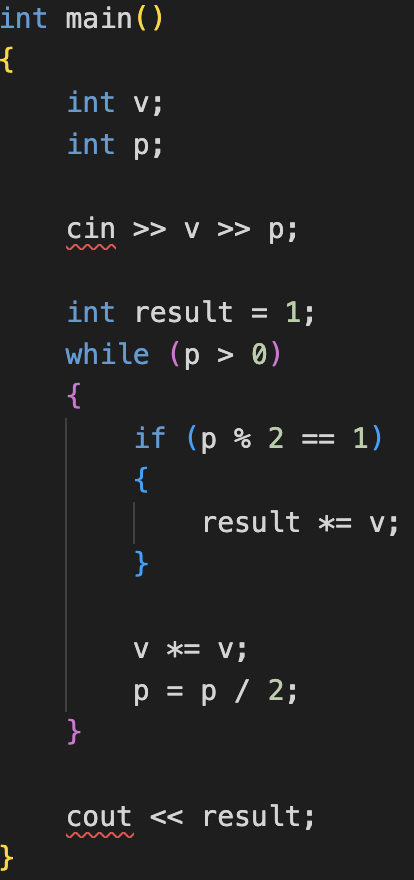


Рисунок 2.1 – Тестовая программа

Результат построения синтаксического дерева анализатором при отсутствии ошибок (рисунки 2.2, 2.3).

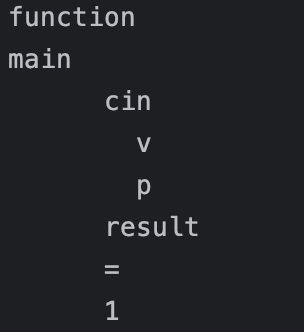


Рисунок 2.2 – Первая часть синтаксического дерева

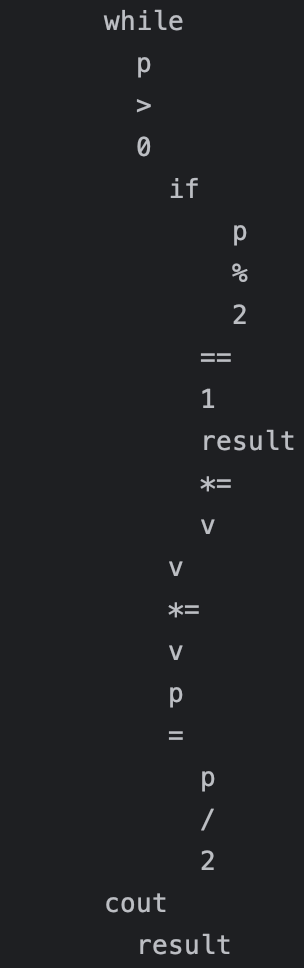


Рисунок 2.3 – Вторая часть синтаксического дерева

## **Обнаружение синтаксических ошибок**

1. ) Отсутствие ; после объявления переменной (см. рисунки 2.4, 2.5).

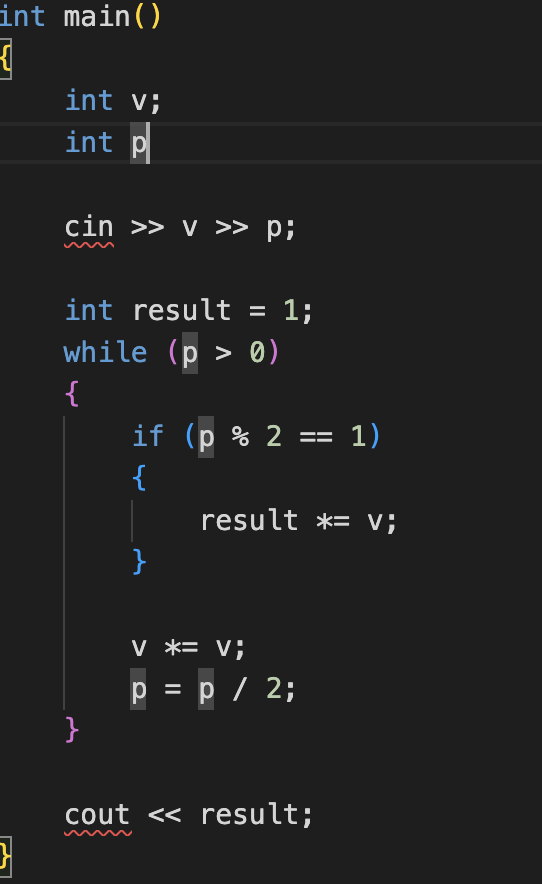


Рисунок 2.4 – Код программы с отсутствующим символом ;



Рисунок 2.5 – Пример реакции анализатора на отсутствие символа ;

1. Отсутствие блока условия для цикла while (см. рисунки 2.6, 2.7).

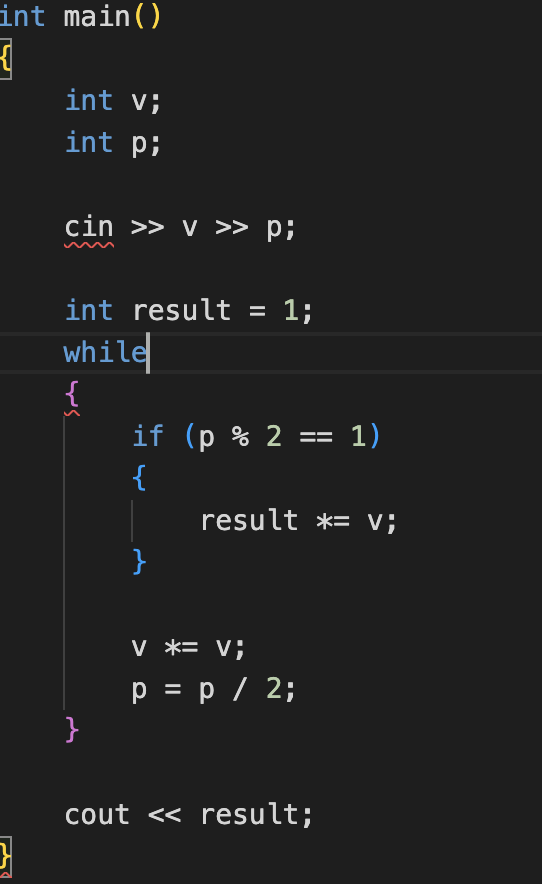


Рисунок 2.6 – Код программы с отсутствием условия для цикла while



Рисунок 2.7 – Пример реакции анализатора на отсутствие условия для цикла while

1. Ошибочная попытка вывода оператором cout (см. рисунки 2.8, 2.9).

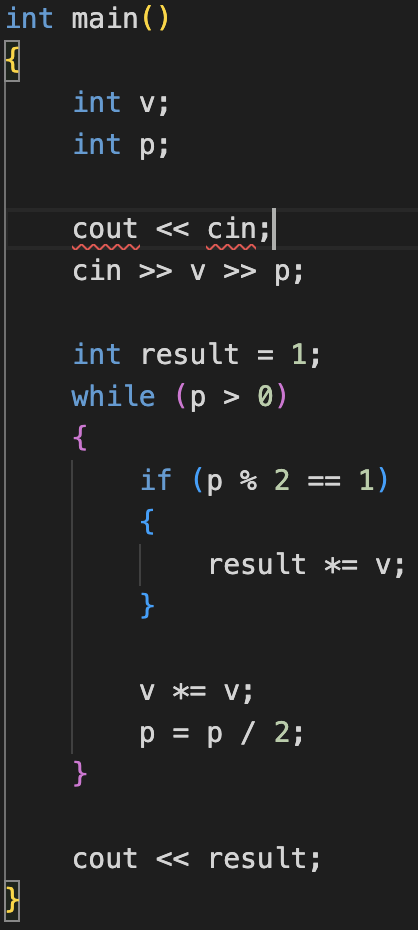


Рисунок 2.8 – Код программы с ошибочной попыткой вывода



Рисунок 2.9 – Реакции анализатора на ошибочную попытку вывода

1. Попытка выполнения ошибочного действия (см. рисунки 2.10, 2.11).

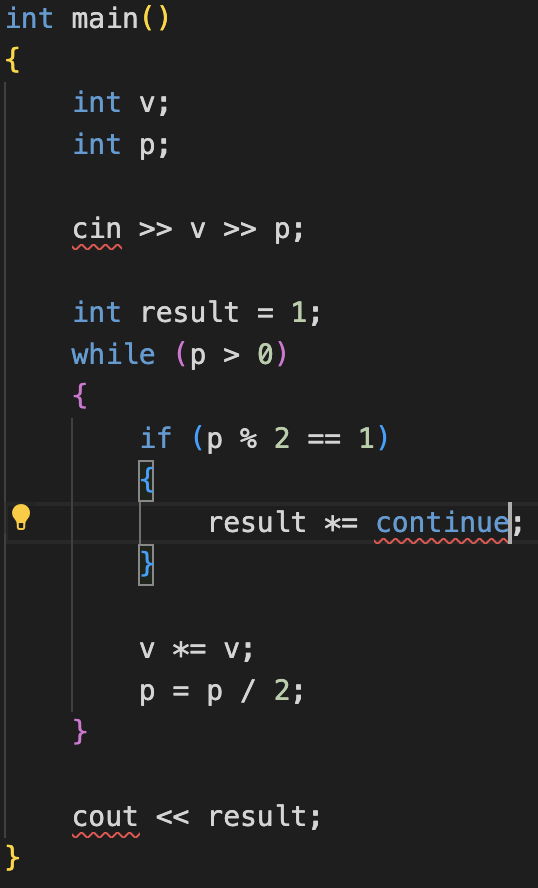


Рисунок 2.10 – Код программы с ошибочным действием



Рисунок 2.11 – Реакция анализатора на ошибочное действие

# **ПРИЛОЖЕНИЕ А**

**Код** **программ**

from functions.lexer import Lexer

from nodes.nodes\_module import \*

from entities.constants import all\_operators, ignore, libs, namespaces

class Parser:

def \_\_init\_\_(self, lexer: Lexer):

self.tokens = lexer.tokens

self.position = 0

self.scope = {}

self.lexer = lexer # instead of

def match(self, expected: []) -> Token:

if self.position < len(self.tokens):

current\_token = self.tokens[self.position]

print(current\_token.word, expected, current\_token.word in expected)

if current\_token.word in expected:

self.position += 1

return current\_token

return None

def get\_prev(self):

return self.tokens[self.position - 1].word

def require(self, expected):

token = self.match(expected)

if token is None:

raise Exception(f'Expected {expected} after {self.get\_prev()}')

return token

def parse\_variable\_or\_constant(self) -> Node:

constant = self.match(self.lexer.constants\_tokens.keys())

if constant:

return ConstantNode(constant)

var = self.match(self.lexer.var\_tokens.keys())

if var:

return VariableNode(var)

constant = self.tokens[self.position]

if constant.word == 'true' or constant.word == 'false':

self.position += 1

return ConstantNode(constant)

elif constant.word == "!":

self.position += 1

return UnaryOperationNode(constant, self.parse\_formula())

raise Exception(f'Expected number or variable after {self.get\_prev()}')

def parse\_parentheses(self) -> Node:

if self.match(['(']):

node = self.parse\_formula()

self.require([')'])

return node

else:

return self.parse\_variable\_or\_constant()

def parse\_formula(self) -> Node:

left\_node = self.parse\_parentheses()

operation = self.match(all\_operators)

while operation:

if operation.word == '<<' or operation.word == '>>':

self.position -= 1

break

elif operation.word == '++' or operation.word == '--':

left\_node = UnaryOperationNode(operation, left\_node)

operation = self.match(all\_operators + [':'])

else:

right\_node = self.parse\_parentheses()

left\_node = BinaryOperationNode(operation, left\_node, right\_node)

operation = self.match(all\_operators + [':'])

return left\_node

def parse\_variable\_definition(self, variable\_token):

var = VariableNode(variable\_token)

comma = self.match([','])

while comma:

new\_var = self.require(self.lexer.var\_tokens.keys())

if new\_var:

var = VariableNode(new\_var)

else:

raise Exception(f'Expected variable after "," after {self.get\_prev()}')

comma = self.match([','])

operation = self.match(['='])

if operation:

value = self.parse\_formula()

self.require([';'])

return BinaryOperationNode(operation, var, value)

self.require([';'])

return None

def parse\_cin(self):

operation = self.match(['>>'])

expression = []

while operation:

expression.append(self.parse\_formula())

operation = self.match(['>>'])

self.require([';'])

return CinNode(expression)

def parse\_cout(self):

operation = self.match(['<<'])

expression = []

while operation:

endl = self.match(['endl'])

if endl:

expression.append(KeyWordNode(endl))

else:

expression.append(self.parse\_formula())

operation = self.match(['<<'])

self.require([';'])

return CoutNode(expression)

def parse\_while(self):

self.require(['('])

condition = self.parse\_formula()

self.require([')'])

self.require(['{'])

body = self.parse\_code()

self.require(['}'])

return WhileNode(condition, body)

def parse\_for(self):

self.require(['('])

self.match(self.lexer.var\_types\_tokens)

begin = self.parse\_formula()

self.require([';'])

condition = self.parse\_formula()

self.require([';'])

step = self.parse\_formula()

self.require([')'])

self.require(['{'])

body = self.parse\_code()

self.require(['}'])

return ForNode(begin, condition, step, body)

def parse\_if\_else\_condition(self):

self.require(['('])

condition = self.parse\_formula()

self.require([')'])

self.require(['{'])

body = self.parse\_code()

self.require(['}'])

if self.match(['else']):

if self.match(['if']):

else\_condition = self.parse\_if\_else\_condition()

else:

self.require(['{'])

else\_condition = self.parse\_code()

self.require(['}'])

return IfNode(condition, body, else\_condition)

return IfNode(condition, body, None)

def parse\_function\_parameters(self, types=False):

parameters = []

if self.match([')']):

self.position -= 1

return parameters

if types:

self.require(self.lexer.var\_types\_tokens)

parameters.append(self.require(self.lexer.var\_tokens.keys()))

comma = self.match([','])

while comma:

if types:

self.require(self.lexer.var\_types\_tokens)

parameters.append(self.require(self.lexer.var\_tokens.keys()))

comma = self.match([','])

return parameters

def parse\_function(self, function\_token):

self.require(['('])

parameters = self.parse\_function\_parameters(True)

self.require([')'])

self.require(['{'])

body = self.parse\_code()

self.require(['}'])

return FunctionNode(function\_token, parameters, body)

def parse\_function\_call(self, function\_token):

self.require(['('])

parameters = self.parse\_function\_parameters()

self.require([')'])

self.require([';'])

return FunctionCallNode(function\_token, parameters)

def parse\_switch(self):

self.require(['('])

variable = self.require(self.lexer.var\_tokens.keys())

self.require([')'])

self.require(['{'])

body = self.parse\_code()

self.require(['}'])

return SwitchNode(variable, body)

def parse\_case(self):

constant = self.parse\_variable\_or\_constant()

self.require([':'])

return CaseNode(constant.constant)

def parse\_key\_word(self, key\_word):

self.require([':'] if key\_word.word == 'default' else [';'])

return KeyWordNode(key\_word)

def parse\_ignored\_keywords(self, key\_word):

if key\_word.word == '#include':

self.require(libs)

elif key\_word.word == 'using':

self.require(['namespace'])

self.require(namespaces)

self.require([';'])

return None

def parse\_expression(self) -> Node:

if self.match(self.lexer.var\_tokens.keys()):

self.position -= 1 # current position is variable

var\_node = self.parse\_variable\_or\_constant()

operation = self.match(all\_operators)

if operation:

# unary and array processing

right\_formula\_node = self.parse\_formula()

self.require([';'])

return BinaryOperationNode(operation, var\_node, right\_formula\_node)

if self.match(self.lexer.var\_types\_tokens):

variable\_token = self.match(self.lexer.var\_tokens.keys())

if variable\_token:

return self.parse\_variable\_definition(variable\_token)

function\_token = self.match(self.lexer.func\_tokens.keys())

if function\_token:

return self.parse\_function(function\_token)

raise Exception(f'Expected variable or function after {self.get\_prev()}')

function\_token = self.match(self.lexer.func\_tokens.keys())

if function\_token:

return self.parse\_function\_call(function\_token)

key\_word = self.match(self.lexer.key\_word\_tokens)

if key\_word:

if key\_word.word in ignore:

return self.parse\_ignored\_keywords(key\_word)

elif key\_word.word == 'case':

return self.parse\_case()

elif key\_word.word == 'default':

return self.parse\_key\_word(key\_word)

elif key\_word.word == 'cin':

return self.parse\_cin()

elif key\_word.word == 'cout':

return self.parse\_cout()

elif key\_word.word == 'for':

return self.parse\_for()

elif key\_word.word == 'if':

return self.parse\_if\_else\_condition()

elif key\_word.word == 'switch':

return self.parse\_switch()

elif key\_word.word == 'continue':

return self.parse\_key\_word(key\_word)

elif key\_word.word == 'break':

return self.parse\_key\_word(key\_word)

elif key\_word.word == 'while':

return self.parse\_while()

def parse\_code(self) -> Node: # block parse

root = StatementsNode()

while self.position < len(self.tokens):

if self.match(['}']):

self.position -= 1

return root

code\_string\_node = self.parse\_expression()

if code\_string\_node:

root.add\_node(code\_string\_node)

return root