Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина «Методы трансляции»

|  |  |
| --- | --- |
|  |  |

**ОТЧЕТ**

к лабораторной работе № 3

на тему «Синтаксический анализатор»

Выполнил:

студент гр. 153504

Михалевич M.П.

Проверил:

Гриценко Н.Ю.

Минск 2024

# СОДЕРЖАНИЕ

[1 Цели работы 3](#_Toc158899741)

[2 Краткие теоретические сведения 4](#_Toc158899742)

[3 Результат выполнения работы 5](#_Toc158899743)

[Вывод 7](#_Toc158899744)

[Список использованных источников 8](#_Toc158899745)

[Приложение А (обязательное) Листинг программного кода 9](#_Toc158899746)

# 1 ЦелИ работы

Целью данной лабораторной работы является проверка исходного текста программы, написанной на языке программирования С на соответствие синтаксическим нормам языка с построением дерева разбора (синтаксического дерева), которое будет отражать синтаксическую структуру входной последовательности, а также в случае несоответствия – позволяет вывести сообщения об ошибках.

# 2 КРАТКИЕ Теоретические сведения

Синтаксический разбор в контексте компиляции — это процедура, направленная на проверку соответствия кода установленным правилам языка программирования, с последующим формированием структурированного представления кода, известного как синтаксическое дерево.

Этот процесс позволяет не только удостовериться в правильности структуры программы, но и облегчает выявление и диагностику ошибок при нарушениях этих правил.

В качестве результата работы синтаксического анализатора получается организованное представление кода, которое может быть выражено через различные типы деревьев, отражающих отношения и иерархии между элементами кода.

Эта процедура ключевая для эффективного синтеза последующего кода, поскольку она организует и группирует базовые компоненты программы, такие как переменные, операции и скобки, в логически связанные блоки.

## РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ

## РАБОТЫ

В ходе лабораторной работы был написан лексический анализатор для языка программирования C. Результат работы анализатора показан на рисунке 3.1.

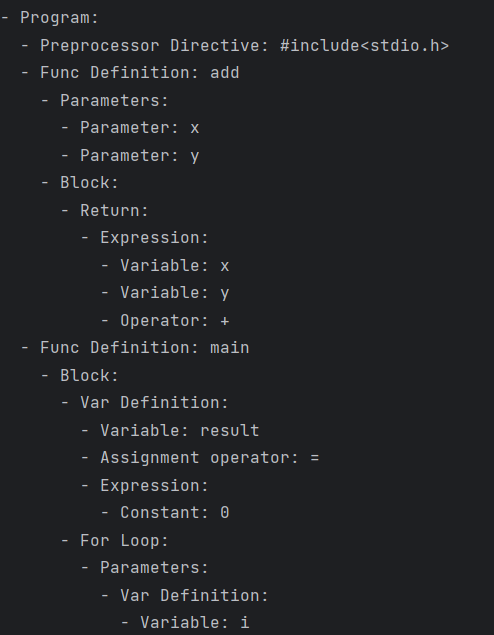


Рисунок 3.1 – Результат работы анализатора

Также программа определяет синтаксические ошибки. Например, если не поставить символ ; в конце строки, анализатор выведет ошибку и подскажет символ, который ожидался. Результат показан на рисунке 3.2.

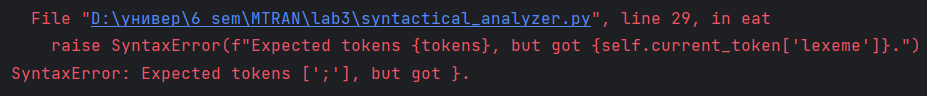


Рисунок 3.2 – Вывод ошибки при использовании некорректного идентификатора

Если оператор “break” или “continue” были использованы вне тела цикла, анализатор покажет соответствующую ошибку. Результат нахождения данной ошибки представлен на рисунке 3.3.

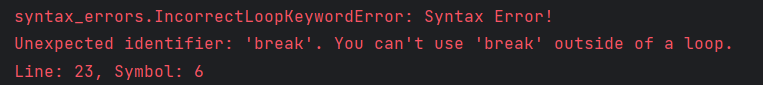


Рисунок 3.3 – Вывод ошибки при некорректном использовании операторов цикла

Если после объявления функции забыть указать тело цикла в фигурных скобках, программа выдаст ошибку. Результат показан на рисунке 3.4.



Рисунок 3.4 – Вывод ошибки при некорректной расстановке скобок

В итоге, был реализован синтаксический анализатор, строит синтаксическое дерево, а также определяет синтаксические ошибки.

# Вывод

В ходе выполнения лабораторной работы был разработан синтаксический анализатор, который проверяет исходный текста программы, написанной на языке программирования С на соответствие синтаксическим нормам языка и строит дерева разбора. Также реализована обработка синтаксических ошибок.

# Список ИСПОЛЬЗОВАННЫХ источников

[1] Синтаксический анализ [Электронный ресурс]. – Режим доступа: https://www.guru99.com/ru/syntax-analysis-parsing-types.html. – Дата доступа: 28.02.2024.

# Приложение а

**(обязательное)**

**Листинг программного кода**

Листинг 1 – Программный код, отвечающий за синтаксический анализ

from syntax\_errors import UnexpectedTokenError, IncorrectLoopKeywordError

from tree\_node import TreeNode

from lexical\_analyzer.constants import DATA\_TYPE, OPERATORS

class Parser:

def \_\_init\_\_(self, tokens):

self.tokens = tokens

self.position = 0

self.current\_token = tokens[0]

self.is\_cycle\_now = False

def next\_token(self):

if self.position + 1 < len(self.tokens):

self.position += 1

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

else:

self.current\_token = {'lexeme': 'EOF', 'type': 'EOF'}

def eat(self, tokens, type=None):

lexeme = self.current\_token['lexeme']

if self.current\_token['lexeme'] in tokens:

self.next\_token()

if (type):

return TreeNode(type, value=lexeme)

else:

raise SyntaxError(f"Expected tokens {tokens}, but got {self.current\_token['lexeme']}.")

def parse(self, type="Block", is\_switch=False):

nodes = []

separators = ['}']

if type == 'Program':

separators.append('EOF')

#print(self.current\_token)

while self.position < len(self.tokens) and self.current\_token and self.current\_token['lexeme'] not in separators:

nodes += self.parse\_program()

return TreeNode(type, children=nodes)

def parse\_program(self):

functions = []

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

if self.current\_token['type'] == "Preprocessor Directive":

functions.append(TreeNode(self.current\_token['type'], value=self.current\_token["lexeme"]))

self.next\_token()

elif self.current\_token['lexeme'] in DATA\_TYPE:

if self.position + 2 < len(self.tokens) and self.tokens[self.position + 2]['lexeme'] == '(':

functions.append(self.parse\_function())

else:

functions += self.parse\_variable\_definition()

elif self.current\_token['type'] == 'KEYWORD':

if self.current\_token['lexeme'] in ['if', 'for', 'while', 'switch', 'return', 'case', 'structure']:

method\_name = f'parse\_{self.current\_token["lexeme"]}'

method = getattr(self, method\_name)

self.next\_token()

res = method()

if isinstance(res, list):

functions += res

else:

functions.append(res)

elif self.current\_token['lexeme'] == 'default':

self.next\_token()

functions.append(self.parse\_case("Default Case", True))

elif self.current\_token['lexeme'] in ['break', 'continue']:

if not self.is\_cycle\_now:

raise IncorrectLoopKeywordError(self.current\_token)

else:

self.next\_token()

else:

self.next\_token()

elif self.current\_token['type'] == "IDENTIFIER":

if self.position + 1 < len(self.tokens) and self.tokens[self.position + 1]['lexeme'] == '(':

functions.append(self.parse\_function\_call())

else:

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

else:

self.next\_token()

return functions

def print\_tree(self, node, level=0):

indent = " " \* level

# print(node.type, node.children)

colon = ":" if node.children or node.value else ""

print(f"{indent}- {node.type}{colon} {node.value if node.value else ''}")

for child in node.children:

if child:

self.print\_tree(child, level + 1)

def parse\_function(self):

self.next\_token()

# print(self.current\_token)

function\_name = self.current\_token['lexeme']

self.next\_token()

function\_node = TreeNode("Func Definition", value=f"{function\_name}")

#print(1)

self.eat('(')

if self.current\_token['lexeme'] != ')':

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

function\_node.add\_child(parameters\_node)

else:

self.next\_token()

#print(3)

self.eat('{')

#print(self.current\_token)

body\_node = TreeNode("Block")

while self.current\_token['lexeme'] != '}':

#print(222)

expressions = self.parse\_program()

for expr in expressions:

if isinstance(expr, list):

for e in expr:

body\_node.add\_child(e)

else:

body\_node.add\_child(expr)

self.next\_token()

#print(5)

function\_node.add\_child(body\_node)

return function\_node

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

parameters\_node = TreeNode("Parameters")

while self.position < len(self.tokens) and self.current\_token['lexeme'] != ')':

if types:

self.next\_token()

if self.current\_token['type'] == "IDENTIFIER":

param\_name = self.current\_token['lexeme']

self.next\_token()

parameter\_node = TreeNode("Parameter", value=f"{param\_name}")

parameters\_node.add\_child(parameter\_node)

else:

previous\_token\_val = self.tokens[self.position - 1]['lexeme'] if self.position > 0 else 'start'

expected = "IDENTIFIER"

current\_token = self.current\_token

raise UnexpectedTokenError(current\_token, previous\_token\_val, expected)

if self.position < len(self.tokens) and self.current\_token['lexeme'] == ',':

self.next\_token()

self.eat([')'])

return parameters\_node

def parse\_variable\_definition(self, type="Var Definition"):

base\_type = self.current\_token["lexeme"]

self.next\_token()

var\_nodes = []

while True:

variable\_name = self.current\_token["lexeme"]

self.next\_token()

var\_node = TreeNode(type=type)

if self.position < len(self.tokens) and self.current\_token["lexeme"] == "=":

self.next\_token()

value\_node = self.parse\_assignment\_value()

var\_node.add\_child(TreeNode("Variable", value=f"{variable\_name}"))

var\_node.add\_child(TreeNode("Assignment operator", value="="))

var\_node.add\_child(value\_node)

var\_nodes.append(var\_node)

if self.position >= len(self.tokens) or self.current\_token["lexeme"] != ',':

break

self.next\_token()

# print('VAR NODES', var\_nodes)

return var\_nodes

def parse\_assignment\_value(self):

if self.position + 1 < len(self.tokens) and self.tokens[self.position + 1]['lexeme'] == '(':

return self.parse\_function\_call()

else:

res = self.parse\_formula()

self.eat([';'])

return res

def parse\_function\_call(self):

function\_name = self.current\_token["lexeme"]

self.position += 1

self.next\_token()

args = []

while self.current\_token["lexeme"] != ')':

arg = self.parse\_formula("")

args.extend(arg)

if self.position < len(self.tokens) and self.current\_token["lexeme"] == ',':

self.next\_token()

self.next\_token()

function\_call\_node = TreeNode("Function Call", value=function\_name)

params\_node = TreeNode("Call params")

params\_node.add\_child(args)

function\_call\_node.add\_child(params\_node)

return function\_call\_node

def parse\_formula(self, type="Expression"):

# print('aaaa', self.current\_token)

node = TreeNode(type=type)

children = []

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

if self.current\_token["lexeme"] not in OPERATORS:

children.append(left\_node)

while self.position < len(self.tokens) and self.current\_token["lexeme"] in OPERATORS:

op = self.current\_token

self.next\_token()

# print(self.current\_token)

operation\_node = TreeNode("Operator", value=op['lexeme'])

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

children.append(left\_node)

children.append(right\_node)

children.append(operation\_node)

left\_node = operation\_node

if type:

node.add\_child(children)

return node

return children

def parse\_variable\_or\_constant(self):

if self.current\_token["type"] == "CONSTANT":

const\_node = TreeNode("Constant", value=self.current\_token["lexeme"])

self.next\_token()

return const\_node

elif self.current\_token["type"] == "IDENTIFIER":

var\_node = TreeNode("Variable", value=self.current\_token["lexeme"])

self.next\_token()

return var\_node

else:

current\_token = self.current\_token

previous\_token\_val = self.tokens[self.position - 1]['lexeme'] if self.position > 0 else 'start'

raise UnexpectedTokenError(current\_token, previous\_token\_val, "")

def parse\_while(self):

while\_node = TreeNode("While Loop")

self.is\_cycle\_now = True

self.eat(['('])

params\_node = TreeNode("Parameters")

params\_node.add\_child(self.parse\_bool("Condition"))

while\_node.add\_child(params\_node)

self.eat([')'])

self.eat(['{'])

while\_node.add\_child(self.parse())

self.eat(['}'])

self.is\_cycle\_now = False

return while\_node

def parse\_for(self):

for\_node = TreeNode("For Loop")

self.is\_cycle\_now = True

self.eat(['('])

params\_node = TreeNode("Parameters")

params\_node.add\_child(self.parse\_variable\_definition())

params\_node.add\_child(self.parse\_bool("Condition"))

self.eat([';'])

params\_node.add\_child(self.parse\_formula("Post Expression"))

self.eat([')'])

self.eat(['{'])

for\_node.add\_child(params\_node)

for\_node.add\_child(self.parse())

self.eat(['}'])

self.is\_cycle\_now = False

return for\_node

def parse\_if(self, type="If"):

nodes = []

if\_node = TreeNode(type)

if type == "Else If":

self.eat(['if'], "Operator")

self.eat(['('], "Symbol")

condition\_node = self.parse\_bool()

if\_node.add\_child(condition\_node)

self.eat([')'], "Symbol")

self.eat(['{'], "Symbol")

true\_branch = self.parse()

if\_node.add\_child(true\_branch)

self.eat(['}'], "Symbol")

nodes.append(if\_node)

if self.position < len(self.tokens) and self.current\_token['lexeme'] == 'else':

self.next\_token()

if self.position < len(self.tokens) and self.current\_token['lexeme'] == 'if':

else\_if\_node = self.parse\_if("Else If")

nodes.extend(else\_if\_node)

else:

self.eat(['{'], "Symbol")

else\_node = TreeNode("Else")

else\_node.add\_child(self.parse())

nodes.append(else\_node)

self.eat(['}'], "Symbol")

return nodes

def parse\_bool(self, type="Condition"):

if self.current\_token['lexeme'] in ["true", "false"]:

bool\_node = TreeNode(type, self.current\_token['lexeme'])

self.next\_token()

return bool\_node

else:

return self.parse\_formula(type)

def parse\_case(self, type="Case", is\_default=False):

case\_node = TreeNode(type)

if not is\_default:

case\_node.add\_child(self.parse\_formula(""))

self.eat([':'])

case\_node.add\_child(self.parse("Block", True))

return case\_node

def parse\_return(self):

return\_node = TreeNode("Return")

expression = self.parse\_formula()

self.eat([';'], "Symbol")

return\_node.add\_child(expression)

return return\_node