Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования «Кубанский государственный университет»

Кафедра вычислительных технологий

**ОТЧЕТ**

о выполнении лабораторной работы № 4

по дисциплине Методы разработки трансляторов

Тема: Построение синтаксического анализатора

Выполнил: ст. гр. 36/1

Воробьев А.Д.

Проверил: Вишняков Ю.М.

Краснодар

2024

**Цель работы:**

Разработать программу синтаксического анализатора исходного теста.

**Задание:**

На вход программы подается файл, содержащий результат лексического анализа и выполняет синтаксический анализ исходной программы. Результатом работы должно быть сообщение о корректности программы или сообщение о первой обнаруженной ошибке с указанием строки и конструкции языка, при разборе которых обнаружена ошибка.

**Ход работы.**

|  |  |  |
| --- | --- | --- |
| Вариант | Входной язык | Выходной язык |
| 2 | Python | Java |

1. **Грамматика заданного подмножества входного языка**

Начальным символом грамматики является нетерминал <программа>. Грамматика имеет следующий вид:

<программа> ::= <операторы>

<операторы> ::= <оператор>; {<оператор>;}

<оператор> ::= [<метка>] (<описание>

| <функция>

| <оператор вызова функции>

| <составной оператор>

| <опер. присв.>

| <условный оператор>

| <цикл for>

| <цикл while>

| <оператор goto>

| <оператор break>

| <оператор continue>

| <оператор return>)

<метка> ::= <имя>:

<имя> ::= <буква> {<буква> | <цифра>}

<буква> ::= *A* | *B* | … | *Z*

<цифра> ::= 0 | 1 | … | 9

<описание> ::= <описание переменных> | <описание массива>

<описание переменных> ::= *<*тип*>* <описание переменной> {, <описание переменной>}

<описание переменной> ::= <имя> [= <выражение>]

<функция> ::= *PUBLIC* <имя>«(»[<список имён>]«)» <составной оператор>

<составной оператор> ::= «{»<операторы>«}»

<оператор вызов функции> ::= <имя>«(»<выражение> {, <выражение>}«)»

<выражение> ::= <оператор вызова функции>

| <число>

| <строка>

| <переменная>

| «(»<выражение>«)»

| <выражение> <арифметическая операция> <выражение>

<число> ::= <целое число> | <вещественное число>

<целое число> ::= <цифра> {, <цифра>}

<вещественное число> ::= <целое число>

| <целое число>.<целое число> [(e | E) [(- | +)] <целое число>]

<строка> ::= ''{символ}'' | '{символ}'

<переменная> ::= <имя>[«[»<выражение>«]»]

<арифметическая операция> ::= % | \* | \*\* | + | - | /

<опер. присв.> ::= <переменная> = <выражение>

<условный оператор> ::= *IF* «(»<условие>«)» <оператор> [*ELSE* <оператор>]

<условие> ::= <унарная лог. операция> «(»<лог. выражение>«)»

| <лог. выражение> {<бинарная лог. операция> <лог. выражение>}

<унарная лог. операция> ::= !

<лог. выражение> ::= <выражение> <операция сравнения> <выражение>

<операция сравнения> ::= != | < | <= | == | > | >=

<бинарная лог. операция> ::= && | ||

<цикл for>::= *FOR* «(»[<опер. присв.>]; [<условие>]; [<опер. присв.>]«)» <оператор>

<цикл while> ::= *WHILE* «(»<условие>«)» <оператор>

<оператор break> ::= *BREAK*

<оператор continue> ::= *CONTINUE*

<оператор return> ::= *RETURN* <выражение>

1. **Описание алгоритма синтаксического разбора**

Основная задача синтаксического анализа – проверка исходной программы на соответствие грамматике языка программирования. Следует еще раз напомнить, что синтаксический анализ производится над кодом программы, который получен на выходе лексического анализа. Результат синтаксического анализа, который часто называется разбором, представляется в виде дерева разбора. Данное дерево должно демонстрировать вывод исходной программы как цепочки символов из начального символа грамматики.

Для построения дерева разбора используются различные методы синтаксического анализа, каждый из которых имеет свои особенности. В нашем случае будет рассмотрен пример разбора «сверху-вниз», который называется рекурсивным спуском. Основная особенность рекурсивного спуска заключается в том, что для анализа каждого нетерминала используется отдельная семантическая процедура, а многократное обращение к ней в процессе анализа и дало название методу – «рекурсивный спуск».

Рассмотрим суть синтаксического анализа методом рекурсивного спуска на примере.

Вначале построим грамматику языка программирования Java. Его грамматика приведена в приложении А.

При программной реализации данного алгоритма были определены следующие переменные и процедуры, за которыми закреплены определенные функциональные назначения.

1. Переменная NXTSYMB – глобальная переменная, содержащая символ, следующий за разбираемым символом. NXTSYMB содержит символ (лексему) исходной программы, который будет обрабатываться следующим, а при поиске новой цели в переменной NXTSYMB всегда находится первый символ, который должен быть исследован.

При выходе из процедуры с сообщением об успехе символ, следующий за закрытой подцепочкой, помещается в NXTSYMB.

1. Процедура SCAN готовит очередной символ исходной программы и помещает его в NXTSYMB.
2. Процедура ERROR предназначена для обработки ошибочных ситуаций.

Ниже в таблице 1 собраны нетерминалы грамматики и имена соответствующих им рекурсивных семантических процедур.

*Таблица 1. Нетерминалы грамматики и имена соответствующих им рекурсивных семантических процедру*

|  |  |
| --- | --- |
| Нетерминальные символы грамматики | Имена рекурсивных процедур |
| <программа> | PROGRAM |
| <операторы> | OPERATORS |
| <оператор> | OPERATOR |
| <имя> | NAME |
| <описание> | DESCRIPTION |
| <описание переменных> | DESCRIPTION\_OF\_VARIABLES |
| <список имен> | LIST\_OF\_NAMES |
| <описание массивов> | DESCRIPTION\_OF\_ARRAYS |
| <функция> | FUNCTION |
| <составной оператор> | COMPOUND\_OPERATOR |
| <выражение> | EXPRESSION |
| <число> | NUMBER |
| <целое число> | INTEGER |
| <вещественное число> | REAL\_NUMBER |
| <строка> | LINE |
| <переменная> | VARIABLE |
| <арифметическая операция> | ARITHMETIC\_OPERATION |
| <оператор присваивания> | ASSIGNMENT\_OPERATOR |
| <условный оператор> | CONDITIONAL\_OPERATOR |
| <условие> | CONDITION |
| <унарная логическая операция> | UNARY\_LOGIC\_OPERATION |
| <логическое выражение> | LOG\_EXPRESSION |
| <операция сравнения> | COMPARISON\_OPERATION |
| <бинарная логическая операция> | BINARY\_LOG\_OPERATION |
| <цикл for> | FOR\_LOOP |
| <цикл while> | WHILE\_LOOP |
| <оператор break> | BREAK\_OPERATOR |
| <оператор continue> | CONTINUE\_OPERATOR |
| <оператор return> | RETURN\_OPERATOR |

1. **Листинг программы**

import ast

from \_ast import Module, AST

from tkinter import \*

import tkinter.scrolledtext as st

from translate import Translator

translator = Translator(from\_lang="autodetect", to\_lang="ru")

binaryOperationNameToSymbol = {

'Mult': '\*',

'Add': '+',

'Sub': '-',

'Div': '/',

'FloorDiv': '//',

'Mod': '%',

'Pow': '\*\*',

}

compareEqNameToSymbol = {

'Lt': '<',

'LtE': '<=',

'Gt': '>',

'GtE': '>=',

'Eq': '==',

'NotEq': '!=',

}

compareBoolNameToSymbol = {

'And': 'and',

'Or': 'or',

}

compareUnaryNameToSymbol = {

'Not': 'not',

}

compareAugAssignNameToSymbol = {

'Add': '+=',

'Sub': '-=',

'Mult': '\*=',

'Div': '/=',

'MatMult': '\*\*='

}

def calcTreeNodes(node: AST | Module, result=0) -> int:

if isinstance(node, list):

return result + sum([calcTreeNodes(node, 0) for node in node])

elif hasattr(node, 'left'):

return result + calcTreeNodes(node.left, 0) + calcTreeNodes(node.right, 0)

elif hasattr(node, 'values'):

return result + sum([calcTreeNodes(node, 0) for node in node.values])

else:

return result + 1

def convert\_to\_rpn(node):

if isinstance(node, ast.BinOp):

left = convert\_to\_rpn(node.left)

right = convert\_to\_rpn(node.right)

return left + right + binaryOperationNameToSymbol.get(node.op.\_\_class\_\_.\_\_name\_\_) + " "

elif isinstance(node, ast.Num):

return str(node.n) + " "

elif isinstance(node, ast.Expr):

return convert\_to\_rpn(node.value)

elif isinstance(node, ast.FunctionDef):

args = ", ".join([arg.arg for arg in node.args.args])

body = " ".join([convert\_to\_rpn(n) for n in node.body])

return node.name + "(" + args + ") " + "НФ " + body + "КФ "

elif isinstance(node, ast.Assign):

target = convert\_to\_rpn(node.targets[0])

value = convert\_to\_rpn(node.value)

return target + value + "= "

elif isinstance(node, ast.Compare):

left = convert\_to\_rpn(node.left)

ops = " ".join([compareEqNameToSymbol.get(op.\_\_class\_\_.\_\_name\_\_) for op in node.ops])

comparators = " ".join([convert\_to\_rpn(comp) for comp in node.comparators])

return left + comparators + ops + " "

elif isinstance(node, ast.Return):

value = convert\_to\_rpn(node.value)

return value + "return "

elif isinstance(node, ast.Yield):

value = convert\_to\_rpn(node.value)

return value + "yield "

elif isinstance(node, ast.AugAssign):

target = convert\_to\_rpn(node.target)

value = convert\_to\_rpn(node.value)

return target + value + compareAugAssignNameToSymbol.get(node.op.\_\_class\_\_.\_\_name\_\_) + " "

elif isinstance(node, ast.For):

target = convert\_to\_rpn(node.target)

print(node.target)

iter = convert\_to\_rpn(node.iter)

body = " ".join([convert\_to\_rpn(n) for n in node.body])

return target + iter + "in " + "НИЦ " + body + "КИЦ "

elif isinstance(node, ast.While):

test = convert\_to\_rpn(node.test)

body = " ".join([convert\_to\_rpn(n) for n in node.body])

return test + "НУЦ " + body + "КУЦ "

elif isinstance(node, ast.List):

elts = ", ".join([convert\_to\_rpn(elt).strip() for elt in node.elts])

return "[" + elts + "] "

elif isinstance(node, ast.Dict):

keys = [convert\_to\_rpn(key) for key in node.keys]

values = [convert\_to\_rpn(value) for value in node.values]

key\_value\_pairs = [k + ": " + v for k, v in zip(keys, values)]

pairs\_str = ", ".join(key\_value\_pairs)

return "{ " + pairs\_str + "} "

elif isinstance(node, ast.Attribute):

value = convert\_to\_rpn(node.value)

attr = node.attr

return value + attr + " "

elif isinstance(node, ast.Num):

return str(node.n) + " "

elif isinstance(node, ast.Call):

args = [convert\_to\_rpn(arg) for arg in node.args]

return node.func.id + " " + "".join(args) + str(calcTreeNodes(node.args)) + "Ф "

elif isinstance(node, ast.Expr):

return convert\_to\_rpn(node.value)

elif isinstance(node, ast.If):

test = convert\_to\_rpn(node.test)

body = " ".join([convert\_to\_rpn(n) for n in node.body])

if len(node.orelse) == 0:

return test + "M1 УПЛ " + body + "М1 "

orelse = " ".join([convert\_to\_rpn(n) for n in node.orelse])

return test + "M1 УПЛ " + body + "М2 БП М1 " + orelse + "М2 "

elif isinstance(node, ast.Subscript):

value = convert\_to\_rpn(node.value)

slice\_value = convert\_to\_rpn(node.slice)

return value + slice\_value + "АЭМ "

elif isinstance(node, ast.Slice):

lower = convert\_to\_rpn(node.lower) if node.lower is not None else ""

upper = convert\_to\_rpn(node.upper) if node.upper is not None else ""

step = convert\_to\_rpn(node.step) if node.step is not None else ""

return lower + upper + step + "SLICE "

elif isinstance(node, ast.UnaryOp):

operand = convert\_to\_rpn(node.operand)

return operand + compareUnaryNameToSymbol.get(node.op.\_\_class\_\_.\_\_name\_\_) + " "

elif isinstance(node, ast.Name):

return node.id + " "

elif isinstance(node, ast.BoolOp):

if len(node.values) == 2:

left = convert\_to\_rpn(node.values[0])

right = convert\_to\_rpn(node.values[1])

return left + right + compareBoolNameToSymbol.get(node.op.\_\_class\_\_.\_\_name\_\_) + " "

else:

print(node.op, node.values)

elif isinstance(node, ast.Constant):

return node.value + " "

elif isinstance(node, ast.IfExp):

test = convert\_to\_rpn(node.test)

body = convert\_to\_rpn(node.body)

orelse = convert\_to\_rpn(node.orelse)

return test + "M1 УПЛ " + body + "М2 БП М1 " + orelse + "М2 "

else:

print(node)

return ""

def python\_to\_rpn(source\_code):

try:

tree = ast.parse(source\_code)

rpn\_expression = ""

for node in tree.body:

rpn\_expression += convert\_to\_rpn(node)

return "Ошибок нет"

except Exception as error:

text = str(error)

print(text)

res = translator.translate(text

.replace("(<unknown>", "")

.split(",")[0]

).replace("никогда", "").replace("непревзойденный", "Не обнаружена открывающая пара")

res2 = translator.translate(text

.replace("(<unknown>", "")

.split(",")[1]

)

return res + " \n" + res2.replace(')', "")

def prog():

f = open('./resources/python.txt', 'r')

input\_sequence = f.read()

f.close()

out\_seq = python\_to\_rpn(input\_sequence)

# файл, содержащий обратную польскую запись

f = open('gen/error\_text.txt', 'w')

f.write(out\_seq)

f.close()

def write\_txt(data, to):

with open(to, 'w') as file:

file.write(data)

def clicked():

write\_txt(codetxt.get("1.0", "end"), 'resources/python.txt')

errorText.delete("1.0", END)

prog()

f1 = open('gen/error\_text.txt', 'r')

text = f1.read()

errorText.insert("1.0", text)

f1.close()

window = Tk()

window.title("LR4")

f1 = open('resources/python.txt', 'r')

text = f1.read()

window.geometry('1340x640')

codetxt = st.ScrolledText(window, font=("Arial", 18))

codetxt.insert("1.0", text)

codetxt.place(x=20, y=20, width=500, height=600)

errorText = st.ScrolledText(window, font=("Arial", 18))

errorText.place(x=820, y=20, width=500, height=600)

btngo = Button(window, text="Проверить \n ошибки", command=clicked, font=("Arial", 20))

btngo.place(x=510 + 150 - 100, y=10 + 300 - 40, width=200, height=80)

window.mainloop()

1. **Пример лексического разбора**

При запуске программы пользователь видит окно, изображенное на рисунке 1:

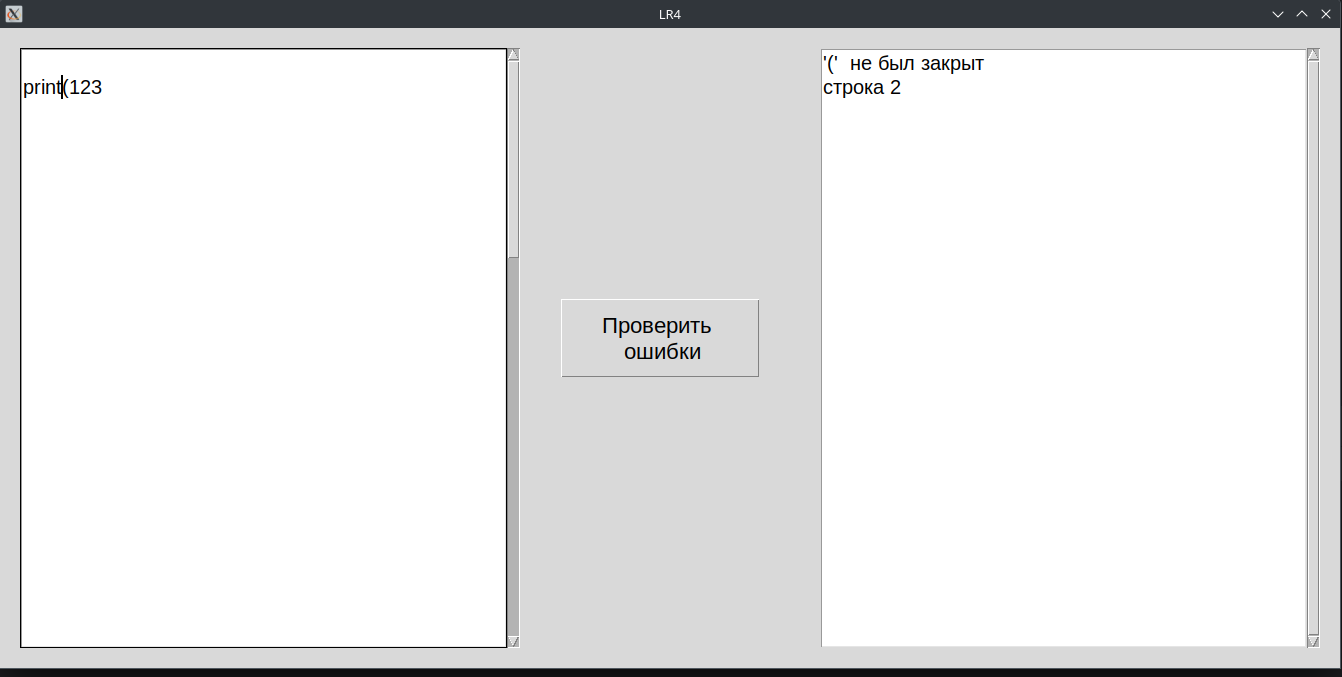


Рисунок 1 – Пример ошибки

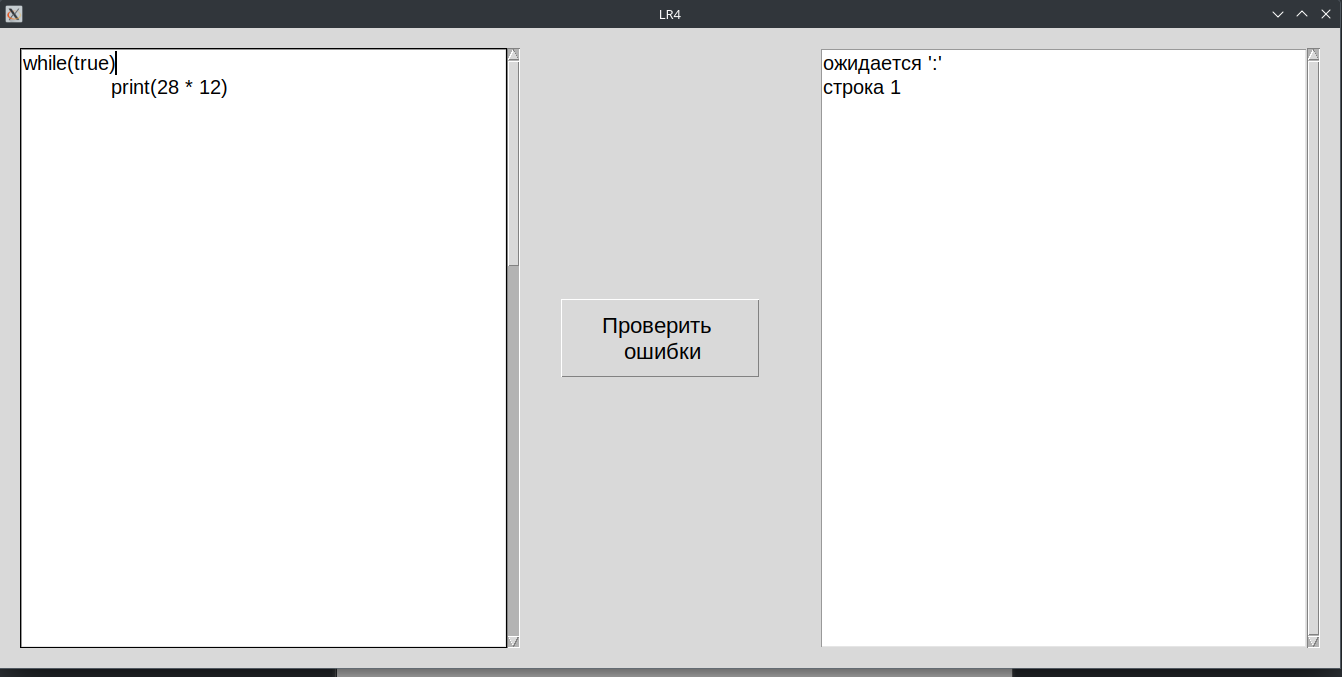


Рисунок 2 – Пример ошибки

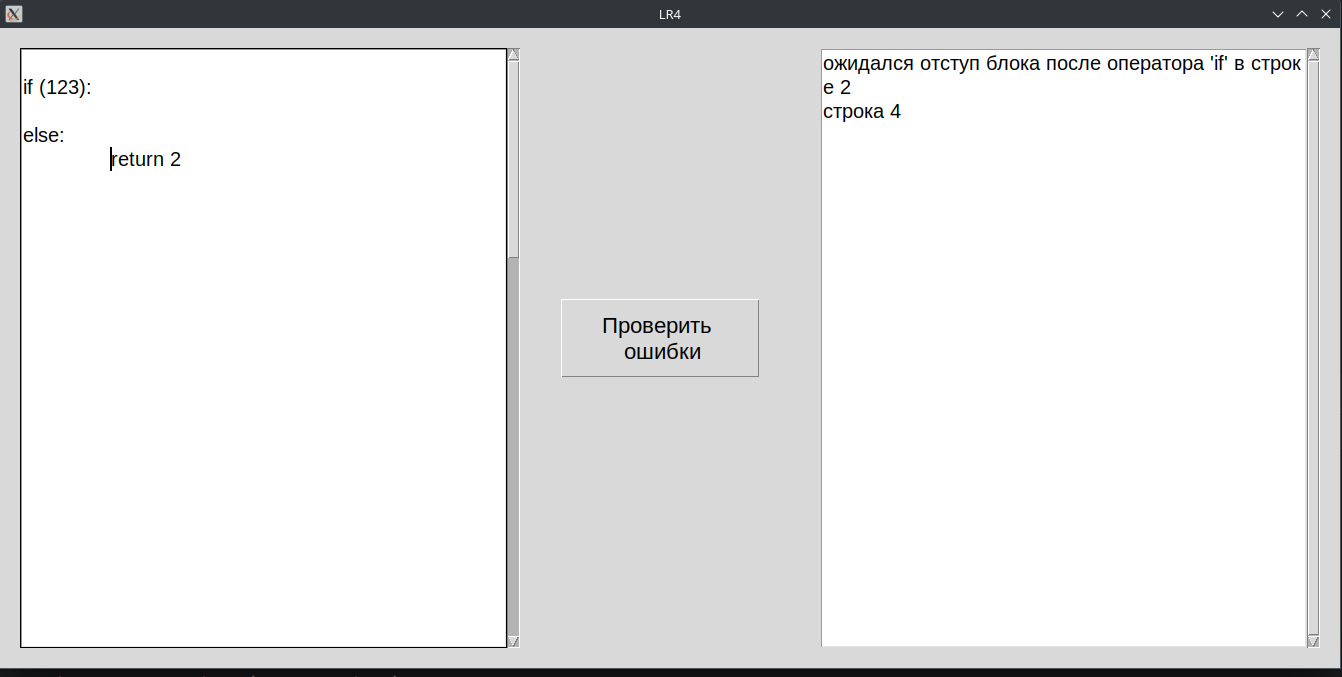


Рисунок 3 – Пример ошибки

Вывод:

В результате выполнения лабораторной работы №4 была создана программа для синтаксического анализа программы на входном языке. Программа предоставляет возможность пользователю ввести код на Python и получить его синтаксический анализ. В дальнейшем полученные навыки будут применены для написания более сложных программ.