Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №3

“Синтаксический анализатор”

Выполнил:

студент гр. 853504

Кузьма В. В.

Проверил:

Ст. преподаватель КИ Шиманский В. В.

Минск 2021

**Содержание**

[**1. Цель работы 3**](#_Toc64740995)

[**2. Краткие теоритические сведения 6**](#_Toc64740996)

[**3. Построение синтаксического дерева 8**](#_Toc64741001)

[**4. Нахождение синтаксических ошибок 1**](#_Toc64741001)**3**

[**Приложение. Текст программ 1**](#_Toc64741002)**4**

# **Цель работы**

В ходе синтаксического анализа исходный текст программы проверяется на на соответствие синтаксическим нормам языка с построением **дерево разбора** (синтаксическое дерево), которое отражает синтаксическую структуру входной последовательности и удобно для дальнейшего использования, а также в случае несоответствия – позволяет вывести сообщения об ошибках.

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.

Представление грамматических фраз исходной программы выполнить в виде дерева. Реализовать синтаксический анализатор с использованием одного из табличных методов (**LL**-, **LR**-метод, метод предшествования и пр.).

**Разбор выражения** COST = (PRICE+TAX)\*0.98.

Выходом анализатора служит дерево, которое представляет синтаксическую структуру, присущую исходной программе.

**<ИД1>=(<ИД2>+<ИД3>)\*<ИД4>.**

По этой цепочке необходимо выполнить следующие действия:

1. **<ИД3>** прибавить к **<ИД2>;**
2. результат **(1)** умножить на **<ИД4>;**
3. результат **(2)** поместить в ячейку, резервированную для **<ИД1>.**

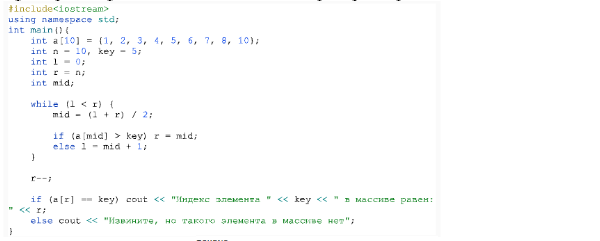
Т.е. мы имеем последовательность шагов в виде помеченного дерева.

Внутренние вершины представляют те действия, которые можно выполнять. Прямые потомки каждой вершины либо представляют аргументы, к которым нужно применять действие (если соответствующая вершина помечена идентификатором или является внутренней), либо помогают определить, каким должно быть это действие, в частности, знаки «+», «\*» и «=». Скобки отсутствуют, т.к. они только определяют порядок действий.

**LL-** и **LR-** методы позволят обнаружить ошибки на самых ранних стадиях, т.е. когда разбор потока токенов от лексического анализатора в соответствии с грамматикой языка становится невозможен.

Можно использовать **нисходящий** (англ. top-down parser) со стартового символа, до получения требуемой последовательности токенов. Для этих целей применим метод рекурсивного спуска либо LL-анализатор. Или использовать **восходящий** (англ. bottom-up parser) - продукции восстанавливаются из правых частей, начиная с токенов и кончая стартовым символом - LR-анализатор и проч.

В качестве тестового примера возьмём один пример из первой лабораторной работы:



# **2. Краткие теоритические сведения**

**Синтаксический анализатор** – это часть программы, преобразующей входные данные в структурированный формат; процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой, обычно применяемый совместно с лексическим анализом. **Парсер** выполняет синтаксический анализ текста.

При реализации синтаксического анализатора надо все символы, которые могут встретиться в обрабатываемом тексте разбить на группы таким образом, чтобы все символы группы вызывали одинаковую реакцию синтаксического анализатора, то есть это и есть лабораторная работа №1 по разбиению на токены. Затем необходимо выделить состояния синтаксического анализатора. Состояние определяет, какие символы в данный момент могут быть на входе синтаксического анализатора, и какова будет реакция на этот символ. Например, если на вход синтаксического анализатора пришла цифра, то он переходит в состояние «Константа», и до завершения константы (т.е. до появления пробела, знака операции, закрывающей скобки или конца строки) на вход синтаксического анализатора могут приходить только цифры и точка, если константа вещественная. Причём, естественно, точка может встречаться только один раз, что приводит к делению состояния «Константа» на два состояния: «Константа до точки», в котором точка может появиться, и «Константа после точки», в котором появление точки будет считаться ошибкой.

Одно состояние является начальным. Именно с него начинается работа синтаксического анализатора, и одно или несколько состояний должны быть конечными. Далее строиться таблица, которая определяет реакцию синтаксического анализатора на входные символы в зависимости от состояния. Реакция обычно заключается в смене состояния синтаксического анализатора и ещё каких-то действиях, например, запись очередного символа во временную переменную, увеличение или уменьшение уровня вложенности при появлении скобок и т.д.

**Наиболее часто встречающиеся виды парсеров:**

* очередь классифицированных лексем;
* абстрактное дерево;
* иерархические структуры;
* таблицы данных;

**Виды парсеров по числу чтений входных данных:**

* однопроходные;
* многопроходные.

**Исходный код парсера может быть:**

* написан программистами;
* сгенерирован специализированными утилитами;

Обычно в программе синтаксический анализатор реализуется как бесконечный цикл (for ( ; ; ) ) с выходами в случае ошибки или при достижении конца обрабатываемой строки. Внутри цикла пишется условный блок, например, по группам символов, и в каждой части этого условного блока – свой условный блок (или оператор-переключатель) по состояниям синтаксического анализатора.

Если выражение записано верно, то в результате работы синтаксического анализатора должен появиться список лексем (список). Поскольку элементы списка должны иметь одинаковый тип, надо выбрать такую структуру, с помощью которой можно представить все возможные лексемы. Это можно сделать, если, например, каждый элемент списка представляет собой структуру, состоящую из двух полей: тип лексемы и её номер в списке лексем этого типа.

Часть лексем в арифметических и логических выражениях состоят из одного символа, но имена переменных и функций и константы в общем случае состоят из нескольких символов. Поэтому входные символы надо записывать во временную переменную, и когда данная лексема заканчивается, проверять, что она из себя представляет.

Следует помнить, что одна и та же переменная может входить в выражение несколько раз, поэтому каждый раз, когда имя переменной сформировалось, надо проверять, была ли она уже в списке переменных.   
А при достижении конца обрабатываемого выражения надо проверить, что количество открывающих и закрывающих скобок было одинаковым (вложенность = 0).

## **3.Построение синтаксического дерева**

В результате дерево программы имеет следующий вид:



Рис 1. Фрагмент дерева

Листинг 1. Все дерево

function:

func\_declaration:

main

args:

block:

body:

assign:

a[10]

arg:

initialization\_list:

1

2

3

4

5

6

7

8

9

10

assign:

n

arg:

10

assign:

key

arg:

5

assign:

l

arg:

0

assign:

r

arg:

n

assign:

mid

arg:

0

while:

condition:

arg:

l

<

arg:

r

block:

body:

assign:

mid

/:

+:

arg:

l

arg:

r

arg:

2

if:

condition:

arg:

a[mid]

>

arg:

key

block:

assign:

r

arg:

mid

else-block:

assign:

l

+:

arg:

mid

arg:

1

assign:

r

--

if:

condition:

arg:

a[mid]

==

arg:

key

block:

cout:

"Индекс элемента "

key

" в массиве равен: "

r

else-block:

cout:

"Извините, но такого элемента в массиве нет"

Конец листинга 1.

1. **Нахождение синтаксических ошибок**

Добавим специально синтаксические ошибки в нашу программу для просмотра обработки ошибочных ситуаций:

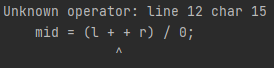


Рис 2. Двойной оператор "+"

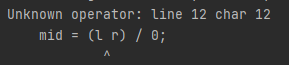


Рис 3. Отсутствие оператора

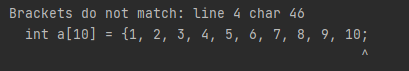


Рис 4. Незакрытая скобка

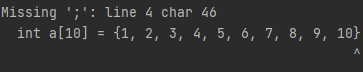


Рис 5. Пропущен оператор ";"

**Приложение. Текст программ**

from ply.lex import LexToken  
  
from Tokenizer import tokens  
from ply import yacc  
  
class Node:  
 def parts\_str(self):  
 st = []  
 for part in self.parts:  
 st.append(str(part))  
 return "\n".join(st)  
  
 def \_\_repr\_\_(self):  
 return self.type + ":\n\t" + self.parts\_str().replace("\n", "\n\t")  
  
 def add\_parts(self, parts):  
 self.parts += parts  
 return self  
  
 def \_\_init\_\_(self, type, parts):  
 self.type = type  
 self.parts = parts  
  
  
variables = []  
  
  
def token\_in\_stack(name, stack):  
 for item in stack:  
 if isinstance(item, LexToken) and item.value == name:  
 return True  
 return False  
  
  
def p\_function(p):  
 *'''function : func\_header func\_body  
 '''* p[0] = Node('function', p[1:])  
  
  
def p\_func\_header(p):  
 *'''func\_header : FUNCTION VARIABLE LPAREN args RPAREN'''* p[0] = Node('func\_declaration', [p[2], p[4]])  
  
  
def p\_args(p):  
 *'''args :  
 | expr  
 | args COMMA expr'''* if len(p) <= 2:  
 p[0] = Node('args', p[1:] if p[1:] else ['EMPTY'])  
 else:  
 p[0] = p[1].add\_parts([p[3]])  
  
  
def p\_func\_body(p):  
 *'''func\_body : block'''* p[0] = p[1]  
  
  
def p\_block(p):  
 *'''block : LFIGURPAREN body RFIGURPAREN'''* p[0] = Node('block', [p[2]])  
  
def p\_else\_block(p):  
 *'''block : LFIGURPAREN body RFIGURPAREN'''* p[0] = Node('else\_block', [p[2]])  
  
  
def p\_body(p):  
 *'''body :  
 | body line semicolons  
 | body multiline'''* if len(p) > 1:  
 if p[1] is None:  
 p[1] = Node('body', [])  
 p[0] = p[1].add\_parts([p[2]])  
 else:  
 p[0] = Node('body', [])  
  
  
def p\_semicolons(p):  
 *'''semicolons : SEMICOLON  
 | semicolons SEMICOLON'''*def p\_multiline(p):  
 *'''multiline : while\_statement'''* p[0] = p[1]  
  
  
def p\_line(p):  
 *'''line : assign  
 | func  
 | RETURN expr  
 | RETURN func'''* if len(p) == 3:  
 p[0] = Node('return', [p[2]])  
 else:  
 p[0] = p[1]  
  
  
def p\_while\_statement(p):  
 *'''while\_statement : WHILE LPAREN condition RPAREN block  
 | WHILE LPAREN condition RPAREN line semicolons'''* p[0] = Node('while', [p[3], p[5]])  
  
def p\_if\_statement(p):  
 *'''while\_statement : IF LPAREN condition RPAREN block  
 | IF LPAREN condition RPAREN line semicolons'''* p[0] = Node('if', [p[3], p[5]])  
  
  
def p\_condition(p):  
 *'''condition : expr cond\_sign expr'''* p[0] = Node('condition', [p[1], p[2], p[3]])  
  
  
def p\_cond\_sign(p):  
 *'''cond\_sign : NOTEQUAL  
 | EQUAL'''* p[0] = p[1]  
  
  
def p\_assign(p):  
 *'''assign : variable EQUAL expr  
 | VAR variable EQUAL expr  
 | VAR variable EQUAL function'''* if len(p) == 5:  
 value = p[2]  
 variables.append(value)  
 p[0] = Node('assign', [p[2], p[4]])  
 else:  
 value = p[1]  
 p[0] = Node('assign', [p[1], p[3]])  
  
  
def p\_func(p):  
 *'''func : VARIABLE LPAREN args RPAREN'''* p[0] = Node('func\_call', [p[1], p[3]])  
  
  
def p\_expr(p):  
 *'''expr : fact  
 | expr OPERATOR fact'''* if len(p) == 2:  
 p[0] = p[1]  
 else:  
 p[0] = Node(p[2], [p[1], p[3]])  
  
  
def p\_fact(p):  
 *'''fact : term  
 | fact OPERATOR term'''* if len(p) == 2:  
 p[0] = p[1]  
 else:  
 p[0] = Node(p[2], [p[1], p[3]])  
  
  
def p\_term(p):  
 *'''term : arg  
 | LPAREN expr RPAREN'''* if len(p) == 2:  
 p[0] = p[1]  
 else:  
 p[0] = p[2]  
  
  
def p\_arg(p):  
 *'''arg : STRING  
 | variable  
 | INT  
 | FLOAT  
 | method  
 | func'''* value = p[1]  
 if isinstance(value, str):  
 p[0] = Node('arg', [p[1]])  
  
  
def p\_method(p):  
 *'''method : variable\_or\_arg DOT func'''* p[0] = Node('method', [p[1], p[3]])  
  
  
def p\_variable\_or\_arg(p):  
 *'''variable\_or\_arg : variable  
 | arg'''* p[0] = p[1]  
  
  
def p\_variable(p):  
 *"""variable : VARIABLE"""* p[0] = p[1]  
  
parser\_errors = []  
  
def p\_error(p):  
 print('Unexpected token in line %d: %s' % (p.lineno, p))  
  
parser = yacc.yacc()  
  
  
def build\_tree(code):  
 p = parser.parse(code)  
 return p  
  
operations = {  
 '+': lambda x, y: x + y,  
 '-': lambda x, y: x - y,  
 '\*': lambda x, y: x \* y,  
 '/': lambda x, y: x / y,  
}  
  
\_\_variables = {}  
  
def parse\_tree(tree):  
 try:  
 tree\_type = tree.type  
 parts = tree.parts  
 except AttributeError:  
 return tree  
 if tree\_type == 'assign':  
 \_\_variables[parts[0]] = parse\_tree(parts[1])  
 return  
 if tree\_type == 'arg':  
 arg = parts[0]  
 if isinstance(arg, int):  
 return arg  
 elif arg in \_\_variables:  
 return \_\_variables[arg]  
 else:  
 return 'method'  
 for part in parts:  
 parse\_tree(part)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 filename = "main.cpp"  
 with open("main.cpp") as f:  
 characters = f.read()  
 tree = build\_tree(characters)  
 parse\_tree(tree)  
 print(tree)