**Белорусский государственный университет информатики и радиоэлектроники**

**Кафедра информатики**

Лабораторная работа № 2

по предмету «Методы трансляции»

**Синтаксический анализатор**

**Выполнил**:Студент группы 453503 Кучинский С.С.

**Проверил**:Ассистент кафедры Информатики Шиманский В.В.

Минск, 2017

**1.Постановка задачи**

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.

В качестве анализируемого подмножества языка программирования будет использован язык программирования Pascal.

Для написания анализатора использован язык программирования Python. Рассмотрены возможности его библиотеки ply.yacc.

Ниже представлен код программы на языке Pascal:

var i, j, counter: integer;

text: string;

begin

i := 5;

j := i + 7;

text := 'abc';

for counter := i to j do

begin

text := text + '\_\*\_';

writeln(text);

end;

writeln(j);

writeln(text);

end.

**2.Теория**

**Синтаксиический 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.

Пример разбора выражения в дерево

Задача синтаксического анализатора – проверить правильность записи выражения и разбить его на лексемы. Лексемой называется неделимая часть выражения, состоящая, в общем случае, из нескольких символов.

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.

Типы алгоритмов

● Нисходящий парсер — продукции грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов, им соответствуют LL-грамматики

● Восходящий парсер — продукции восстанавливаются из правых частей, начиная с токенов и кончая стартовым символом, им соответствуют LR-грамматики.

**3. Программа и комментарии**

На Рис. 3.1. показана функция, используемая для построения синтаксического дерева.

**Рис. 3.1. Функция для построения синтаксического дерева**

Во время выполнения данной работы использовали уже написанный лексический анализатор. Из выражений, уже выделенных, составлялись синтаксические конструкции по некоторым правилам Рис.3.2.

**Рис. 3.2. Одно из правил для построения дерева**

Данное правило говорит, что конструкция while в языке Pascal имеет 2 формы:

1. Только одно правило (конструкция predicate, которая имеет свои собственные формы)

2. Несколько правил (конструкция some\_predicates, которая представляет собой объединение нескольких конструкций predicate, объединенных ключевыми словами and или or)

Код со всеми правилами построения представлен в листинге 1 в конце отчёта.

**4. Результаты выполнения**

В результате дерево программы имеет следующий вид Рис.4.1

**Рис. 4.1. Построенное дерево анализируемой программы**

**Код с ошибками:**

Рассмотрим тот же код программы с добавленными в него ошибок. При обнаружении их происходит вывод уведомления об ошибке.

Код программы с 1-ой ошибкой Рис.4.4.

**Рис. 4.4. Синтаксическая ошибка объявления типа переменной**

Результат работы программы с допущением 1-ой ошибки Рис.4.5.

**Рис. 4.5. Результат работы программы с допущением 1-ой ошибки**

Код программы со 2-ой ошибкой Рис.4.6.

**Рис. 4.6. Синтаксическая ошибка конструкции for**

Результат работы программы с допущением 2-ой ошибки Рис.4.7.

**Рис. 4.7. Результат работы программы с допущением 2-ой ошибки**

Код программы с 3-ей ошибкой Рис.4.8.

**Рис. 4.8. Синтаксическая ошибка №3**

Результат работы программы с допущением 3-ей ошибки Рис.4.9.

**Рис. 4.9. Результат работы программы с допущением 3-ой ошибки**

Код программы с 4-ой ошибкой Рис.4.10.

**Рис. 4.10. Синтаксическая ошибка неправильный вызов функции**

Результат работы программы с допущением 4-ой ошибки Рис.4.11.

**Рис. 4.11. Результат работы программы с допущением 4-ой ошибки**

**5. Вывод**

В результате работы были получены знания о синтаксических деревьях, способах их построения. Для выделения синтаксических структур из кода был использован нисходящий парсер. В итоге работы был простроен простой синтаксический анализатор на основе уже имеющегося лексического с помощью задания правил для синтаксических конструкций, который способен не только строить и выводить синтаксическое дерево, но и находить ошибки, уведомлять о них, генерируя исключения, выводя их на консоль.

В ходе работы изначальная версия анализатора дорабатывалась для улучшения работы парсера. Была заложена основа для создания компилятора для языка Pascal в последующих лабораторных работах.

Конечная версия программы может находить и анализировать ошибки основного синтаксиса. При расширении набора выражений для поиска конструкций языка, анализатор будет способен обрабатывать все необходимые программы на языке, вплоть до создания таблицы токенов и поиска ошибок в полной версии языка. Но данная задача представляется достаточно трудной, из-за большого разнообразия библиотек языка, а так же использовании паттернов.

**6. Листинги**

**Листинг 1. Правила построения дерева**

def p\_consts(p): '''

consts : CONST IDENTIFIER EQUALITY NUMBER SEMICOLON begin\_program

| CONST IDENTIFIER EQUALITY STRING SEMICOLON begin\_program

| CONST IDENTIFIER EQUALITY matrix SEMICOLON begin\_program

| begin\_program '''

if len(p) == 2: p[0] = p[1]

else:

p[0] = [['const', p[2], p[4]], p[6]]

def p\_matrix(p): '''

matrix : OPEN\_SQUARE\_BRACKET identifiers CLOSE\_SQUARE\_BRACKET

'''

p[0] = [p[1] + p[3], p[2]]

def p\_identifiers(p): '''

identifiers : IDENTIFIER ZAPYATAYAidentifiers | NUMBER ZAPYATAYAidentifiers

| STRING ZAPYATAYAidentifiers | IDENTIFIER

| NUMBER | STRING

'''

if len(p) == 2: p[0] = p[1]

else:

p[0] = [[p[1]], p[3]]

def p\_begin\_program(p): '''

begin\_program : VAR declarations BEGIN body END POINT '''

p[0] = [p[2], ['begin program', p[4]]]

def p\_block(p): '''

block : BEGIN body END SEMICOLON '''

p[0] = [p[2]]

def p\_body(p): '''

body : expression '''

p[0] = [p[1]]

def p\_identifier(p): '''

identifier : IDENTIFIER '''

if list(identifiers.keys()).count(p[1]) == 0:

semantic\_errors.append('{0}) There is no {1} variable!'.format(p.lexer.lineno, p[1]))

p[0] = p[1]

def p\_expression(p): '''

expression : assignment expression | if expression

| function expression | empty

| while expression | for expression

| break '''

# empty

if len(p) == 2: p[0] = [p[1]]

else:

p[0] = [p[1], p[2]]

def p\_break(p): '''

break : BREAK SEMICOLON '''

p[0] = [p[1]]

def p\_declarations(p): '''

declarations : declaration declarations | empty

'''

if len(p) == 2: p[0] = p[0]

else:

p[0] = [p[1], p[2]]

def p\_declaration(p): '''

declaration : IDENTIFIER another\_identifiers COLON type SEMICOLON

'''

if type(p[2]) is list:

p[0] = [['declare', p[1], p[4]]] + [['declare', item, p[4]] for item in p[2]] elif p[2] is None:

p[0] = ['declare', p[1], p[4]] else:

p[0] = [['declare', p[1], p[4]], ['declare', p[2], p[4]]]

def p\_another\_identifiers(p): '''

another\_identifiers : ZAPYATAYAIDENTIFIER another\_identifiers | empty

'''

if len(p) == 2: p[0] = p[0]

else:

if p[3] is None: p[0] = p[2]

else:

p[0] = [p[2], p[3]]

def p\_type(p): '''

type : TYPE\_STRING | TYPE\_INTEGER | TYPE\_REAL

'''

p[0] = [p[1]]

def p\_empty(p): '''

empty : '''

p[0] = p[0]

def p\_assignment(p): '''

assignment : identifier ASSIGNMENT arithmetic\_expression SEMICOLON

| identifierASSIGNMENT function SEMICOLON | identifierASSIGNMENT function

| identifierASSIGNMENT arithmetic\_expression '''

p[0] = ['assign ' + p[1], p[2], p[3]]

def p\_arithmetic\_expression(p): '''

arithmetic\_expression : NUMBER | identifier

| STRING | function

| identifier PLUS arithmetic\_expression

| identifier MINUS arithmetic\_expression | identifier MUL arithmetic\_expression

| identifier DIV arithmetic\_expression

| NUMBER PLUS arithmetic\_expression

| NUMBER MINUS arithmetic\_expression | NUMBER MUL arithmetic\_expression

| NUMBER DIV arithmetic\_expression | function PLUS arithmetic\_expression

| function MINUS arithmetic\_expression | function MUL arithmetic\_expression

| function DIV arithmetic\_expression

| identifier OPEN\_SQUARE\_BRACKET arithmetic\_expression CLOSE\_SQUARE\_BRACKET

| arithmetic\_expression PLUS arithmetic\_expression

| arithmetic\_expression MINUS arithmetic\_expression | arithmetic\_expression MUL arithmetic\_expression

| arithmetic\_expression DIV arithmetic\_expression '''

if len(p) == 5:

p[0] = [p[1], p[2] + p[4], p[3]] elif len(p) == 2:

p[0] = p[1] else:

if p[2] in ("+", "-", "/", "\*"):

if type(p[3]) is list and type(p[1]) is list:

if list(identifiers.keys()).count(p[1][0]) > 0 and list(identifiers.keys()).count(p[3][0]) > 0:

if identifiers[p[3][0]][1] in (list, str) and identifiers[p[1][0]][1] in (list, str) and\

identifiers[p[3][0]][0] != identifiers[p[1][0]][0]: semantic\_errors.append("{3}) Operations with different types!

{0} {1} {2}"

.format(p[1], p[2], p[3], p.lexer.lineno))

elif list(identifiers.keys()).count(p[1]) > 0 and list(identifiers.keys()).count(p[3]) > 0:

if identifiers[p[3]][1] != identifiers[p[1]][1]: semantic\_errors.append("{3}) Operations with different types! {0}

{1} {2}"

.format(p[1], p[2], p[3], p.lexer.lineno)) elif list(identifiers.keys()).count(p[1]) > 0:

if ((type(p[3]) is list and p[3][1] == "[]" and identifiers[p[3][0]][1] != identifiers[p[1]][1]) or\

(type(p[3]) is not list and (not (type(p[3]) in (int, float) and identifiers[p[1]][1]

in (int, float)) and not (type(p[3]) is str and identifiers[p[1]][1] is str)))) and p[3] != "0":

semantic\_errors.append("{3}) Operations with different types! {0} {1} {2}"

.format(p[1], p[2], p[3], p.lexer.lineno)) elif list(identifiers.keys()).count(p[3]) > 0:

if (type(p[1]) is list and p[1][1] == "[]" and identifiers[p[1][0]][1] != identifiers[p[3]][1]) or \

(type(p[1]) is not list and (not (type(p[1]) in (int, float) and identifiers[p[3]][1]

in (int, float)) and not (type(p[1]) is str and identifiers[p[3]][1] is str))):

semantic\_errors.append("{0}) Operations with different types! {1} {2} {3}"

.format(p.lexer.lineno, p[1], p[2], p[3]))

if p[2] == "/" and ((type(p[3]) in (int, float) and p[3] == 0) or p[3] == "0"): semantic\_errors.append("{3}) Division by 0 exception. {0} {1}

{2}".format(p[1], p[2], p[3], p.lexer.lineno))

if list(identifiers.keys()).count(p[3]) > 0 and p[2] == "/": if identifiers[p[3]][0] == 0:

semantic\_errors.append(

"{3}) Division by 0 exception. {0} {1} {2}".format(p[1], p[2], p[3], p.lexer.lineno))

p[0] = [p[1], p[2], p[3]]

def perform\_operation(operation, v1, v2): return operation(v1, v2)

def p\_function(p): '''

function : WRITE OPEN\_BRACKET arithmetic\_expression CLOSE\_BRACKET SEMICOLON

| WRITELN OPEN\_BRACKET arithmetic\_expression CLOSE\_BRACKET SEMICOLON

| READ OPEN\_BRACKET identifier CLOSE\_BRACKET SEMICOLON

| READLN OPEN\_BRACKET identifier CLOSE\_BRACKET SEMICOLON

| LENGTH OPEN\_BRACKET identifier CLOSE\_BRACKET

| INC OPEN\_BRACKET identifier CLOSE\_BRACKET SEMICOLON

'''

p[0] = [p[1], p[3]]

def invert\_operation(operation): invertor = {'>': '<', '<': '>', '<=': '>=',

'>=': '<=', '=': '<>', '<>': '=', 'in': 'in'} return invertor[operation]

def p\_predicate(p): '''

predicate : arithmetic\_expression COMPARISON arithmetic\_expression | arithmetic\_expression EQUALITY arithmetic\_expression

| arithmetic\_expression NON\_EQUALITY arithmetic\_expression

| arithmetic\_expression NON\_STRICT\_COMPARISON arithmetic\_expression

| arithmetic\_expression IN identifier | arithmetic\_expression IN matrix

'''

if p[3] in identifiers.keys() and p[2] == "in" and identifiers[p[3]][1] is not list:

semantic\_errors.append("{0}) You can pass operator \"in\" only for matrix.".format(p.lexer.lineno))

old\_len = len(semantic\_errors) check\_comparation(p[1], p[2], p[3], p.lexer.lineno) if old\_len == len(semantic\_errors):

check\_comparation(p[3], p[2], p[1], p.lexer.lineno) if type(p[1]) is list and p[1][1] == '[]':

p[0] = p[1] + [p[2], p[3]] else:

p[0] = [p[1], p[2], p[3]]

def check\_comparation(p1, p2, p3, line): if type(p1) is list and p2 != "in":

tp = None for \_ in p1:

if list(identifiers.keys()).count(\_) > 0: tp = identifiers[\_][1]

break

tp = tp if tp is not None else type(p1[0]) if type(p3) is not list:

if (type(p3) is not tp) or (tp is str and type(p3) in (int, float)): semantic\_errors.append("{0}) You can't pass different types to {1}

operation.".format(line, p2))

elif list(identifiers.keys()).count(p3) > 0 and identifiers[p3][1] != tp: semantic\_errors.append("{0}) You can't pass different types to {1}

operation.".format(line, p2)) else:

tp1 = None for \_ in p3:

if list(identifiers.keys()).count(\_) > 0: tp1 = identifiers[\_][1]

break

tp1 = tp1 if tp1 is not None else type(p3[0]) if tp != tp1:

semantic\_errors.append("{0}) You can't pass different types to {1} operation.".format(line, p2))

else:

if list(identifiers.keys()).count(p1) > 0:

if list(identifiers.keys()).count(p3) > 0:

if identifiers[p3][1] != identifiers[p1][1]: semantic\_errors.append("{0}) You can't pass different types to {1}

operation.".format(line, p2)) else:

if identifiers[p1][1] != type(p3):

semantic\_errors.append("{0}) You can't pass different types to {1} operation.".format(line, p2))

else:

if list(identifiers.keys()).count(p3) > 0: if type(p1) != identifiers[p3][1]:

semantic\_errors.append("{0}) You can't pass different types to {1} operation.".format(line, p2))

else:

if type(p1) != type(p3):

semantic\_errors.append("{0}) You can't pass different types to {1} operation.".format(line, p2))

def p\_some\_predicates(p): '''

some\_predicates : OPEN\_BRACKET predicate CLOSE\_BRACKET AND some\_predicates

| OPEN\_BRACKET predicate CLOSE\_BRACKET OR some\_predicates

| OPEN\_BRACKET predicate CLOSE\_BRACKET

| OPEN\_BRACKET NOT OPEN\_BRACKET predicate CLOSE\_BRACKET CLOSE\_BRACKETAND some\_predicates

| OPEN\_BRACKET NOT OPEN\_BRACKET predicate CLOSE\_BRACKET CLOSE\_BRACKET OR some\_predicates

| OPEN\_BRACKET NOT OPEN\_BRACKET predicate CLOSE\_BRACKET CLOSE\_BRACKET

'''

if len(p) == 9:

p[0] = [[p[2], p[3], p[4], p[5], p[7]], p[8]] elif len(p) == 7:

p[0] = [p[2], p[4]] elif len(p) == 4:

p[0] = p[2]

else:

p[0] = [[p[2], p[4]], p[5]]

def p\_while(p): '''

while : WHILE OPEN\_BRACKET predicate CLOSE\_BRACKET DO block

| WHILE some\_predicates DO block '''

if len(p) == 5:

p[0] = [['while', p[2]], ['do', p[4]]] else:

p[0] = [['while', p[3]], ['do', p[6]]]

def p\_for(p): '''

for : FOR assignment TO arithmetic\_expression DO block

| FOR assignment DOWNTO arithmetic\_expression DO block '''

p[0] = [['for', [p[2], [p[3], p[4]]]], p[5], p[6]]

def p\_if(p): '''

if : IF OPEN\_BRACKET predicate CLOSE\_BRACKET THEN block

| IF OPEN\_BRACKET predicate CLOSE\_BRACKET THEN block else

'''

if len(p) == 8:

p[0] = [['if', p[3]], ['then', p[6]], ['else', p[7]]] else:

p[0] = [['if', p[3]], ['then', p[6]]]

def p\_else(p): '''

else : ELSE block '''

p[0] = p[2]

def p\_error(p):

print('Unexpected token {0}'.format(p))