

**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);

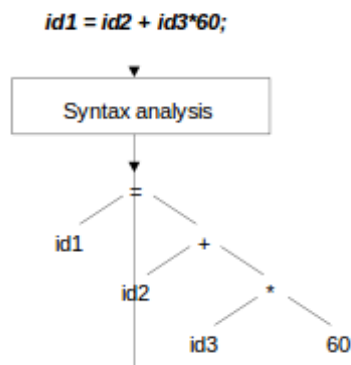
end.
```

## 2.Теория

**Синтаксический 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.

### 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 — productions грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов, им соответствуют LL-грамматики
- Восходящий парсер — productions 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.

```
def tree_print(l, i):
    for item in l:
        if type(item) is list:
            tree_print(item, i)
        else:
            i += 1
            if item:
                print('\t' * i + str(item))
```

**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.

```
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]]]
```

**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

```
seryozhka@Lenovo-G505:~/Desktop/MTran/2lab$ python main.py
declare
    i
        integer
declare
    j
        integer
declare
    counter
        integer
declare
    text
        string
begin program
    assign i
        :=
            5.0
    assign j
        :=
            i
            +
            7.0
    assign text
        :=
            "abc"
    for
        assign counter
            :=
                i
        to
            j
    do
        assign text
            :=
                text
        writeln
            text
    writeln
        j
    writeln
        text
```

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.

```
1  var i, j, counter: int;
2      text: string;
3  begin
4      i := 5;
5      j := i + 7;
6      text := "abc";
7      for counter := i to j do
8          begin
9              text := text;
10             writeln(text);
11         end;
12     writeln(j);
13     writeln(text);
14 end.
15
```

Рис. 4.4. Синтаксическая ошибка объявления типа переменной

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

```
seryozhka@Lenovo-G505:~/Desktop/MTran/2lab$ python main.py
Unexpected token LexToken(IDENTIFIER,'int',1,19)
```

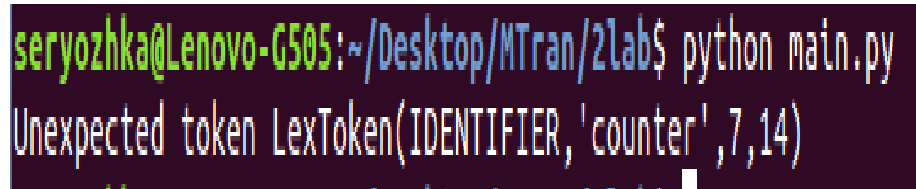
Рис. 4.5. Результат работы программы с допущением 1-ой ошибки

Код программы со 2-ой ошибкой Рис.4.6.

```
1  var i, j, counter: integer;
2      text: string;
3  begin
4      i := 5;
5      j := i + 7;
6      text := "abc";
7      for counter counter := i to j do
8          begin
9              text := text;
10             writeln(text);
11         end;
12     writeln(j);
13     writeln(text);
14 end.
15
```

Рис. 4.6. Синтаксическая ошибка конструкции for

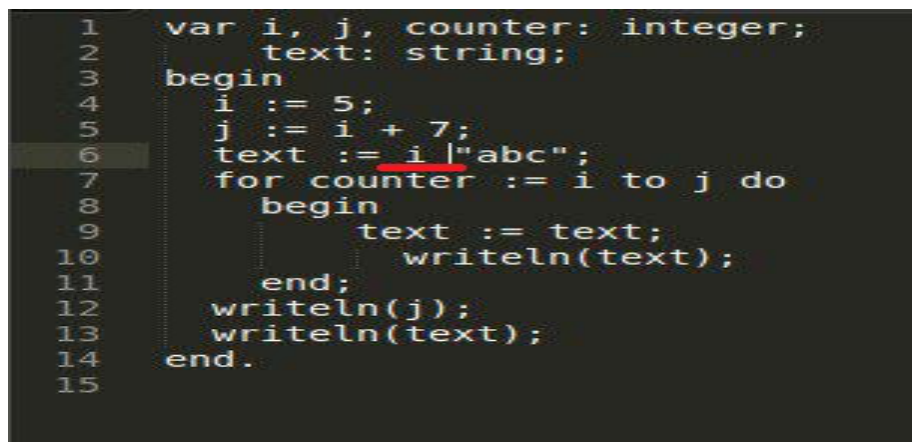
Результат работы программы с допущением 2-ой ошибки Рис.4.7.



```
seryozhka@Lenovo-G505:~/Desktop/MTTran/2lab$ python main.py
Unexpected token LexToken(IDENTIFIER, 'counter', 7, 14)
```

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

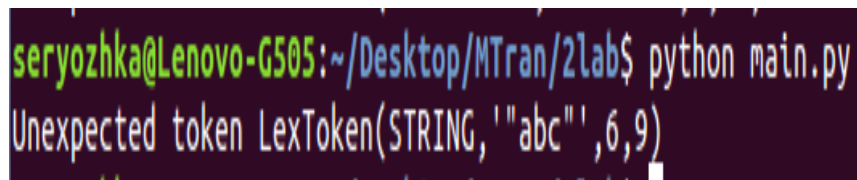
Код программы с 3-ей ошибкой Рис.4.8.



```
1  var i, j, counter: integer;
2      text: string;
3  begin
4      i := 5;
5      j := i + 7;
6      text := i | "abc";
7      for counter := i to j do
8          begin
9              text := text;
10             writeln(text);
11         end;
12     writeln(j);
13     writeln(text);
14 end.
15
```

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

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



```
seryozhka@Lenovo-G505:~/Desktop/MTTran/2lab$ python main.py
Unexpected token LexToken(String, 'abc', 6, 9)
```

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

Код программы с 4-ой ошибкой Рис.4.10.

```
1  var i, j, counter: integer;  
2      text: string;  
3  begin  
4      i := 5;  
5      j := i + 7;  
6      text := writeln|"abc";  
7      for counter := 1 to j do  
8          begin  
9              text := text;  
10             writeln(text);  
11         end;  
12     writeln(j);  
13     writeln(text);  
14 end.  
15
```

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

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

```
seryozhka@Lenovo-G505:~/Desktop/MTtran/2lab$ python main.py  
Unexpected token LexToken(String, 'abc', 6, 18)
```

**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 ZAPYATAYA identifiers
                    | NUMBER ZAPYATAYA identifiers
                    | STRING ZAPYATAYA identifiers
                    | 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 : ZAPYATAYA IDENTIFIER 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
    | identifier ASSIGNMENT function SEMICOLON
    | identifier ASSIGNMENT function
    | identifier ASSIGNMENT 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_BRACKET AND 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 DOWNTTO 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))

```

