Государственное образовательное учреждение
высшего профессионального образования
“Московский государственный технический университет
имени Н.Э. Баумана”

# Отчет

по лабораторной работе №3
по курсу “Конструирование компиляторов”
по теме
“Синтаксический разбор
с использованием метода рекурсивного спуска”

Студент: Адамова И.О.
Группа: ИУ7-22м
Вариант: 2
Преподаватель: Ступников А.

Москва, 2020

# Цель работы

Приобретение практических навыков реализации метода рекурсивного спуска.

# Задачи работы

1) Изучить метод рекурсивного спуска, понять принцип его работы, принцип работы синтаксического анализатора.
2) Применить полученные навыки устранения левой рекурсии и эпсилон-правил.

# Преобразование грамматики

Грамматика, предложенная в варианте задания, содержит эпсилон-правила и леворекурсивные правила, которые следует устранить для корректной работы программы. (Красным цветом выделены правила, которые следует изменить, зеленым - правила, в которых были внесены изменения)

<программа> → <блок>
<блок> → { <список операторов> }
<список операторов> → <оператор> <хвост>
<хвост> → ; <оператор> <хвост> | ε
<оператор> → <идентификатор> = <выражение> | <блок>
<выражение> → <арифметическое выражение> <операция отношения>
<арифметическое выражение> | <арифметическое выражение>
<арифметическое выражение> → <арифметическое выражение> <операция типа сложения> <терм> | <терм>
<терм> → <терм> <операция типа умножения> <фактор> | <фактор>
<фактор> → <идентификатор> | <константа> | ( <арифметическое выражение> )
<операция отношения> → < | <= | == | <> | > | >=
<операция типа сложения> → + | -
<операция типа умножения> → * | /

Избавимся от эпсилон-правила и получим:

<программа> → <блок>
<блок> → { <список операторов> }
<список операторов> → <оператор> <хвост> | <оператор>
<хвост> → ; <оператор> <хвост> | ; <оператор>
<оператор> → <идентификатор> = <выражение> | <блок>

<выражение> → <арифметическое выражение> <операция отношения>
<арифметическое выражение> | <арифметическое выражение>
<арифметическое выражение> → <арифметическое выражение> <операция типа сложения> <терм> | <терм>
<терм> → <терм> <операция типа умножения> <фактор> | <фактор>
<фактор> → <идентификатор> | <константа> | ( <арифметическое выражение> )
<операция отношения> → < | <= | == | <> | > | >=
<операция типа сложения> → + | -
<операция типа умножения> → * | /

Избавимся от левой рекурсии поэтапно. Вместо альтернативы <блок> из правила для оператора следует подставить альтернативы из правила с левой частью <блок>.
<программа> → <блок>
<блок> → { <список операторов> }
<список операторов> → <оператор> <хвост> | <оператор>
<хвост> → ; <оператор> <хвост> | ; <оператор>
<оператор> → <идентификатор> = <выражение> | { <список операторов> }
<выражение> → <арифметическое выражение> <операция отношения>
<арифметическое выражение> | <арифметическое выражение>
<арифметическое выражение> → <арифметическое выражение> <операция типа сложения> <терм> | <терм>
<терм> → <терм> <операция типа умножения> <фактор> | <фактор>
<фактор> → <идентификатор> | <константа> | ( <арифметическое выражение> )
<операция отношения> → < | <= | == | <> | > | >=
<операция типа сложения> → + | -
<операция типа умножения> → * | /

Избавимся от непосредственной левой рекурсии в правиле для арифметического отношения:
<программа> → <блок>
<блок> → { <список операторов> }
<список операторов> → <оператор> <хвост> | <оператор>
<хвост> → ; <оператор> <хвост> | ; <оператор>
<оператор> → <идентификатор> = <выражение> | { <список операторов> }
<выражение> → <арифметическое выражение> <операция отношения>
<арифметическое выражение> | <арифметическое выражение>
<арифметическое выражение> → <терм><арифметическое выражение'>
<арифметическое выражение'> → <операция типа сложения> <терм>
<арифметическое выражение'> | ε
<терм> → <терм> <операция типа умножения> <фактор> | <фактор>
<фактор> → <идентификатор> | <константа> | ( <арифметическое выражение> )
<операция отношения> → < | <= | == | <> | > | >=

<операция типа сложения> → + | -
<операция типа умножения> → * | /
Избавимся от непосредственной левой рекурсии в правиле для терма:
<программа> → <блок>
<блок> → { <список операторов> }
<список операторов> → <оператор> <хвост> | <оператор>
<хвост> → ; <оператор> <хвост> | ; <оператор>
<оператор> → <идентификатор> = <выражение> | { <список операторов> }
<выражение> → <арифметическое выражение> <операция отношения>
<арифметическое выражение> | <арифметическое выражение>
<арифметическое выражение> → <терм><арифметическое выражение'>
<арифметическое выражение'> → <операция типа сложения> <терм>
<арифметическое выражение'> | ε
<терм> → <фактор><терм'>
<терм'> → <операция типа умножения> <фактор> <терм'> | ε
<фактор> → <идентификатор> | <константа> | ( <арифметическое выражение> )
<операция отношения> → < | <= | == | <> | > | >=
<операция типа сложения> → + | -
<операция типа умножения> → * | /

Таким образом в грамматике не осталось леворекурсивных правил.

# Описание работы программы

Программа получает на вход файл в формате .txt с текстом программы, которую нужно проверить на синтаксическое соответствие грамматике, предложенной в варианте. Текст разбивается на список лексем по пробелам для упрощения программной реализации (поэтому, например, знак ";", разделяющий операторы, следует ставить отдельно от предыдущего символа и тому подобное).
Программа включает в себя класс Parser, в который входят методы для каждого нетерминала из грамматики. Метод просматривает текущую лексему и проверяет её на соответствие альтернативам правила, которому этот метод соответствует. Например, может ли находиться на этом месте данный терминал. Когда в альтернативе настает очередь нетерминала, вызывается соответствующий ему метод. Если в какой-то момент оказывается, что для текущей лексемы не существует никакой альтернативы, то есть предложенный текст программы не соответствует грамматике, то возбуждается исключение. Если же такой ситуации не возникнет, и разбор пройдет до конца, то будет выведено сообщение о соответствии программы грамматике.

# Тесты

Для проверки правильности работы программы были проведены следующие тесты.

```
{
    x = 1 ;
    y = 2 ;
    x = 3 * 2 + 7 ;
    z = 3 < 1 + 7 * 2 ;
    {
       y = 2 == 5 ;
       z = 1 + 3 + ( 5 - 7 )
    }

}
```

Вывод: `Your program is syntactically correct`

Этот текст программы соответствует грамматике.

Теперь добавим к последнему присваиванию переменной `z` “;” - разделеитель операторов. Так как этот оператор присваивания последний в блоке, после него не должна стоять “;”. Иначе за ним должен был бы стоять следующий оператор.

```
{
    x = 1 ;
    y = 2 ;
    x = 3 * 2 + 7 ;
    z = 3 < 1 + 7 * 2 ;
    {
       y = 2 == 5 ;
       z = 1 + 3 + ( 5 - 7 ) ;
    }

}
```

Вывод: `parser.ParserSyntaxError: ParserSyntaxError: Operator expected`

Попробуем ввести текст программы без внешних фигурный скобок, которые должны быть у блока.

```
    x = 1 ;
    y = 2 ;
    x = 3 * 2 + 7 ;
    z = 3 < 1 + 7 * 2 ;
    {
       y = 2 == 5 ;
       z = 1 + 3 + ( 5 - 7 ) ;
    }
```

Вывод: `parser.ParserSyntaxError: ParserSyntaxError: { expected`

Попробуем ввести текст программы, у которой есть 2 следующих друг за другом блока. (Грамматика в себя включает только вложенные блоки)

```
{
    x = 1
}
{
    y = 2 ;
    x = 3 * 2 + 7 ;
    z = 3 < 1 + 7 * 2 ;
    {
       y = 2 == 5 ;
       z = 1 + 3 + ( 5 - 7 ) ;
    }
}
```

Вывод: `parser.ParserSyntaxError: ParserSyntaxError: EOF expected`

Здесь допущена ошибка в первой строке блока:

```
{
    y > 2 ;
    x = 3 * 2 + 7 ;
    z = 3 < 1 + 7 * 2 ;
    {
       y = 2 == 5 ;
       z = 1 + 3 + ( 5 - 7 )
    }
}
```

Вывод: `parser.ParserSyntaxError: ParserSyntaxError: = expected after identificator`

Таким образом, по результатам проведенных тестов можно сделать вывод, что программа работает корректно.

# Заключение

В результате лабораторной работы была разработана программа на языке Python, осуществляющая синтаксический разбор методом рекурсивного спуска и выдающая сообщения о синтаксических ошибках в случае их наличия.

# Листинг

main.py

```
from parser import Parser

if __name__ == "__main__":
    filename = 'program.txt'
```

```
    parser = Parser.read_program_from_file(filename)
    parser.print_lexems()
    parser.analyse()
```

parser.py

```
class ParserSyntaxError(Exception):
    def __init__(self, *args):
        if args:
               self.message = args[0]
        else:
               self.message = None

    def __str__(self):
        if self.message:
               return f'ParserSyntaxError: {self.message}'
        else:
               return f'ParserSyntaxError was raised'

class Parser:
    def __init__(self, string_of_lexems: str):
        self.lexems = string_of_lexems.split()
        self.lexems.append('_')
        self.number_of_current_lexem = 0
        self.length_of_list = len(self.lexems)
        self.current_lexem = self.lexems[self.number_of_current_lexem]
        self.print_name_of_functions = True
        self.print_lexem = True

    def read_program_from_file(filename: str):
        with open(filename) as file:
               data = file.read()
               return Parser(string_of_lexems = data)

    def next(self):
        self.number_of_current_lexem += 1

        self.current_lexem = self.lexems[self.number_of_current_lexem]
        if self.print_lexem:
               print(f'Currently examining lexem  "{self.current_lexem}"')

    def print_lexems(self):
        print(f'Lexems: {self.lexems}')

    def __is_identificator(self, lexem):
        return lexem == 'x' or lexem == 'y' or lexem == 'z'

    def __is_operation_of_relation(self, lexem):
        return lexem == '==' or lexem == '>' or lexem == '>=' or lexem == '<>' or
lexem == '<' or lexem == '<='

    def __is_operation_of_addition(self, lexem):
        return lexem == '+' or lexem == '-'

    def __is_operation_of_multiplication(self, lexem):
```

```
        return lexem == '*' or lexem == '/'

    def __is_constant(self, lexem):
        #print(f'in range? {lexem} {lexem in range(10)}')
        return lexem in ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')

    def analyse(self):
        self.program()
        if self.current_lexem != '_':
               raise ParserSyntaxError('EOF expected')
        print(f'Your program is syntactically correct')

    def program(self):
        if self.print_name_of_functions:
               print(f'In program')
        self.block()

    def block(self):
        if self.print_name_of_functions:
               print(f'In block')
        if self.lexems[self.number_of_current_lexem] == '{':
               self.next()
               self.list_of_operators()
               if self.lexems[self.number_of_current_lexem] == '}':
                      self.next()
               else:
                      raise ParserSyntaxError('} expected in the end of the block')
        else:
               raise ParserSyntaxError('{ expected')

    def list_of_operators(self):
        if self.print_name_of_functions:
               print(f'In list_of_operators')
        if self.__is_identificator(self.current_lexem) or self.current_lexem ==
'{':
               self.operator()
               if self.current_lexem == ';':
                      self.tail()
        else:
               raise ParserSyntaxError('Operator expected')

    def operator(self):
        if self.print_name_of_functions:
               print(f'In operator')
        if self.current_lexem == '{':
               self.next()
               self.list_of_operators()
               if self.current_lexem == '}':
                      self.next()
               else:
                      raise ParserSyntaxError('} expected at the end of block')
        elif self.__is_identificator(self.current_lexem):
               self.identificator()
               if  self.current_lexem == '=':
```

```
                self.next()
                self.expression()
            else:
                raise ParserSyntaxError('= expected after identificator')
        else:
            raise ParserSyntaxError('Operator expected')

    def tail(self):
        if self.print_name_of_functions:
            print(f'In tail')
        if self.current_lexem == ';':
            self.next()
            self.operator()
            if self.current_lexem == ';':
                self.tail()
        else:
            raise ParserSyntaxError('Tail expected')

    def expression(self):
        if self.print_name_of_functions:
            print(f'In expression')
        self.arithmetic_expression()
        if self.__is_operation_of_relation(self.current_lexem):
            self.operation_of_relation()
            self.arithmetic_expression()

    def arithmetic_expression(self):
        if self.print_name_of_functions:
            print(f'In arithmetic_expression')
        self.term()
        if self.__is_operation_of_addition(self.current_lexem):
            self.arithmetic_expression_()

    def arithmetic_expression_(self):
        if self.print_name_of_functions:
            print(f'In arithmetic_expression_')
        self.operation_of_addition()
        self.term()
        if self.__is_operation_of_addition(self.current_lexem):
            self.arithmetic_expression_()

    def term(self):
        if self.print_name_of_functions:
            print(f'In term')
        self.factor()
        if self.__is_operation_of_multiplication(self.current_lexem):
            self.term_()

    def term_(self):
        if self.print_name_of_functions:
            print(f'In term_')
        self.operation_of_multiplication()
        self.factor()
        if self.__is_operation_of_multiplication(self.current_lexem):
```

```
            self.term_()

    def factor(self):
        if self.print_name_of_functions:
            print(f'In factor')
        if self.__is_identificator(self.current_lexem):
            self.identificator()
        elif self.__is_constant(self.current_lexem):
            self.constant()
        elif self.current_lexem == '(':
            self.next()
            self.arithmetic_expression()
            if self.current_lexem == ')':
                self.next()
            else:
                raise ParserSyntaxError(') expected')
        else:
            raise ParserSyntaxError('Identificator, constant or
arithmetic_expression in brackets expected')


    def operation_of_relation(self):
        if self.print_name_of_functions:
            print(f'In operation_of_relation')
        if self.__is_operation_of_relation(self.current_lexem):
            self.next()
        else:
            raise ParserSyntaxError('Operation of relation expected')

    def operation_of_addition(self):
        if self.print_name_of_functions:
            print(f'In operation_of_addition')
        if self.__is_operation_of_addition(self.current_lexem):
            self.next()
        else:
            raise ParserSyntaxError('Operation of addition expected')

    def operation_of_multiplication(self):
        if self.print_name_of_functions:
            print(f'In operation_of_multiplication')
        if self.__is_operation_of_multiplication(self.current_lexem):
            self.next()
        else:
            raise ParserSyntaxError('Operation of multiplication expected')

    def constant(self):
        if self.print_name_of_functions:
            print(f'In constant')
        if self.__is_constant(self.current_lexem):
            self.next()
        else:
            raise ParserSyntaxError('Constant expected')

    def identificator(self):
```

```
if self.print_name_of_functions:
       print(f'In identificator')
if self.__is_identificator(self.current_lexem):
       self.next()
else:
       raise ParserSyntaxError('Identificator expected')
```