МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ И.С. ТУРГЕНЕВА»

Кафедра информационных систем и цифровых технологий

ОТЧЕТ
по лабораторной работе № 4
на тему: «Метод рекурсивного спуска»
по дисциплине: «Теория автоматов и формальных языков»

Выполнили: Кожухова О.А., Шорин В.Д.
Институт приборостроения, автоматизации и информационных технологий
Направление: 09.03.04 «Программная инженерия»
Группа: 71ПГ
Проверили: Гордиенко А.П., Чижов А.В.

Отметка о зачете:

Дата: «____» _____ 2021 г.

Орел, 2021 г.

Задание на лабораторную работу:

Написать грамматику для распознавания программы. Реализовать полученную грамматику методом рекурсивного спуска.

Выполнение работы:

Грамматика:

GOAL → SECTION
SECTION → DECLARATION IMPLEMENTATION
DECLARATION → var_term VAR_DECL_INSTR ; VAR_DECLARATION
// каждое объявление переменной/ных
VAR_DECLARATION → VAR_DECL_INSTR ; VAR_DECLARATION
→ E
// очередная переменная/ список переменных одного типа
VAR_DECL_INSTR → id ID_DECL : type_term
ID_DECL → , id ID_DECL
→ E
IMPLEMENTATION → begin_term LIST_INSTRUCTION end_term .
LIST_INSTRUCTION → ASSIGNMENT_INSTRUCTION ; LIST_INSTRUCTION
→ READ_INSTR ; LIST_INSTRUCTION
→ WRITE_INSTR ; LIST_INSTRUCTION
→ BRANCH_INSTR ; LIST_INSTRUCTION
→ E
ASSIGNMENT_INSTRUCTION → id := EXPR
EXPR → TERM EXPR1
EXPR1 → + TERM EXPR1
→ - TERM EXPR1
→ E
TERM → FACTOR TERM1
TERM1 → * FACTOR TERM1
→ / FACTOR TERM1
→ E
FACTOR → (EXPR)
→ round_term (EXPR)
→ num
→ float(realnum)
→ id
READ_INSTR → read_term (id)
WRITE_INSTR → write_term (id)
→ write_term (“ string_term “)
→ write_term (“ id “)
BRANCH_INSTR → if_term CONDITION CONSEQUENCE
CONDITION → BOOL_EXPR CONDITION1
→ EXPR BOOL_OP EXPR
CONDITION1 → or_term BOOL_EXPR CONDITION1
→ E
BOOL_EXPR → BOOL_TERM BOOL_EXPR1
BOOL_EXPR1 → and_term BOOL_TERM BOOL_EXPR1
→ E
BOOL_TERM → (CONDITION)
→ (EXPR BOOL_OP EXPR)
BOOL_OP → =
→ <
→ >
CONSEQUENCE → then_term CONSEQUENCE1 ELSE
CONSEQUENCE1 → ASSIGNMENT_INSTRUCTION
→ READ_INSTR
→ WRITE_INSTR
→ BRANCH_INSTR

→ begin LIST_INSTRUCTION end
ELSE → else_term CONSEQUENCE1
→ E

Программа:

```
import ply.lex as lex
import tokrules
import ply.yacc as yacc

lexer = lex.lex(module=tokrules)
token = lexer

def goal():
    return section()

def section():
    return declaration() and implementation_instr()

# region declaration
def declaration():
    global token
    if token.type == "VAR":
        token = lexer.token()
        if var_decl_instr() and token.type == ';':
            token = lexer.token()
            return var_declaration()
        else:
            return False
    else:
        return False

def var_declaration():
    global token
    # if token.type == 'ID':
    if token is not None and token.type == 'ID':
        if var_decl_instr() \
                and token is not None \
                and token.type == ';':
            token = lexer.token()
            return var_declaration()
        else:
            return False
    else:
        return True  # var_declaration -> E

def var_decl_instr():
    global token
    if token.type == 'ID':
        token = lexer.token()
        if id_decl():
            if token.type == ':':
                token = lexer.token()
                if token.type in tokrules.term_types:
                    token = lexer.token()
                    return True
                else:
                    return False
            else:
                return False
        else:
            return False
```

```
    else:
        return False

def id_decl():
    global token
    if token.type == ',':
        token = lexer.token()
        if token.type == 'ID':
            token = lexer.token()
            return id_decl()
        else:
            return False
    else:
        return True  # ->E

# endregion

# region implementation
def implementation_instr():
    global token
    if token.type == 'BEGIN' \
            and list_instructions() \
            and token.type == 'END':
        return True
    else:
        return False

def list_instructions():
    global token
    token = lexer.token()
    if token.type == 'ID':
        if assignment_instruction() \
                and token.type == ';' \
                and list_instructions():
            return True
        else:
            return False
    elif token.type == 'READ':
        if read_instruction() \
                and token.type == ';' \
                and list_instructions():
            return True
        else:
            return False
    elif token.type == 'WRITE':
        if write_instruction() \
                and token.type == ';' \
                and list_instructions():
            return True
        else:
            return False
    elif token.type == 'IF':
        if branch_instruction() \
                and token.type == ';' \
                and list_instructions():
            return True
        else:
            return False
    else:
        return True  # ->E

# region Арифметические выражения
def assignment_instruction():
```

```
    global token
    if lexer.token().type == 'ASSIGNMENT':
        token = lexer.token()
        return expr()
    else:
        return False

def expr():
    global token
    if term() and expr1():
        return True
    else:
        return False

def expr1():
    global token
    if token.type == '+':
        token = lexer.token()
        if term() and expr1():
            return True
        else:
            return False
    elif token.type == '-':
        token = lexer.token()
        if term() and expr1():
            return True
        else:
            return False
    else:
        return True  # ->E

def term():
    global token
    if factor() and term1():
        return True
    else:
        return False

def term1():
    global token
    if token.type == '*':
        token = lexer.token()
        if factor() and term1():
            return True
        else:
            return False
    elif token.type == '/':
        token = lexer.token()
        if factor() and term1():
            return True
        else:
            return False
    else:
        return True  # ->E

def factor():
    global token
    if token.type == 'NUMBER':
        result = True
    elif token.type == 'FLOAT':
        result = True
    elif token.type == 'ID':
        result = True
```

```
    elif token.type == 'ROUND':
        if lexer.token().type == '(':
            token = lexer.token()
            result = expr() and token.type == ')'
        else:
            result = False
    elif token.type == '(':
        token = lexer.token()
        result = expr() and token.type == ')'
    else:
        result = False
    token = lexer.token()
    return result

# endregion

# region Read_Write instructions
def read_instruction():
    global token
    token = lexer.token()
    if token.type == '(':
        token = lexer.token()
        if token.type == 'ID':
            result = lexer.token().type == ')'
            token = lexer.token()
        else:
            result = False
    else:
        result = False
    return result

def write_instruction():
    global token
    token = lexer.token()
    if token.type == '(':
        token = lexer.token()
        if token.type == 'ID':
            result = lexer.token().type == ')'
        elif token.type == ',':
            token = lexer.token()
            if token.type == 'ID':
                result = (lexer.token().type == ',') and (lexer.token().type == ')')
            elif token.type == 'STRING_TERM':
                result = (lexer.token().type == ',') and (lexer.token().type == ')')
            else:
                result = False
        else:
            result = False
        token = lexer.token()
    else:
        result = False
    return result

# endregion

# region Branch
def branch_instruction():
    global token
    token = lexer.token()
    if condition() and consequence():
        return True
    else:
        return False
```

```
# region condition
def condition():
    global token
    if token.type == '(':
        return boolexpr() and condition1()
    else:
        return expr() and boolop() and expr()

def condition1():
    global token
    if token.type == 'OR':
        token = lexer.token()
        if boolexpr() and condition1():
            return True
        else:
            return False
    else:
        return True  # ->E

def boolexpr():
    if boolterm() and boolexpr1():
        return True
    else:
        return False

def boolexpr1():
    global token
    if token.type == 'AND':
        token = lexer.token()
        if boolterm() and boolexpr1():
            return True
        else:
            return False
    else:
        return True  # ->E

def boolterm():
    global token
    if token.type == '(':
        token = lexer.token()
        if token.type == '(':
            result = condition()
        else:
            result = expr() and boolop() and expr() and token.type == ')'
        token = lexer.token()
    else:
        result = False
    return result

def boolop():
    global token
    if token.type == '=':
        result = True
    elif token.type == '<':
        result = True
    elif token.type == '>':
        result = True
    else:
        result = False
    token = lexer.token()
    return result
```

```
# endregion

# region consequence
def consequence():
    global token
    if token.type == 'THEN':
        token = lexer.token()
        return consequence1() and else_branch()
    else:
        return False

def consequence1():
    global token
    if token.type == 'ID':
        return assignment_instruction()
    elif token.type == 'READ':
        return read_instruction()
    elif token.type == 'WRITE':
        return write_instruction()
    elif token.type == 'IF':
        return branch_instruction()
    elif token.type == 'BEGIN':
        if list_instructions() and token.type == 'END':
            token = lexer.token()
            return True
        else:
            return False
    else:
        return False

def else_branch():
    global token
    if token.type == 'ELSE':
        token = lexer.token()
        return consequence1()
    else:
        return True  # ->E

# endregion
# endregion

# endregion

def myAnalyzer():
    data = 'var ' \
           'a, e: integer;' \
           'r: real;' \
           's: string;' \
           'c: char;' \
           'b: boolean; ' \
           'begin ' \
           'a:= 5;' \
           'r:= a * c + 1 / 2 - 3 + round(a);' \
           'if (a < b) and (c > d) or (a = d) then ' \
           '    begin' \
           '    a:= 2;' \
           '    end ' \
           ' else ' \
           '    begin ' \
           '    c:= a;' \
           '    end;' \
           'write(a);' \
           'read(s);' \
```

```
        'end' \
\
      # data = ','
   lexer.input(data)

   global token
   token = lexer.token()

   # print(token.type)
   if goal():
      print("Программа соответствует грамматике")
   else:
      print("Программа не соответствует грамматике")

   # while True:
   #     tok = lexer.token()
   #     if not tok:
   #         break
   #     print(tok)

# Press the green button in the gutter to run the script.
if __name__ == '__main__':
   myAnalyzer()
```

Реализация грамматики:

```
term_types = ['INTEGER', 'REAL', 'STRING', 'CHAR', 'BOOLEAN']

literals = ['+', '-', '*', '/',
            '=', '<', '>',
            ';', ':', ',',
            '(', ')',
            "'", '\"',
            ]

reserved = {
   'var': 'VAR',
   'if': 'IF',
   'then': 'THEN',
   'else': 'ELSE',
   'begin': 'BEGIN',
   'end': 'END',
   'round': 'ROUND',
   'read': 'READ',
   'write': 'WRITE',
   'or': 'OR',
   'and': 'AND',
   'integer': 'INTEGER',
   'real': 'REAL',
   'string': 'STRING',
   'char': 'CHAR',
   'boolean': 'BOOLEAN',
}

tokens = [
            'ID',
            'FLOAT',
            'NUMBER',
            'ASSIGNMENT',
            'STRING_TERM',
            'WHITESPACE',
            'OTHER',
```

```
            # 'LOGICAL_AND',
            # 'LOGICAL_OR',
            # 'OPERATION_PLUS',
            # 'OPERATION_MINUS',
            # 'OPERATION_MULTIPLICATION',
            # 'OPERATION_DIVISION',
            # 'EQUAL_LESS',
            # 'EQUAL_MORE',
            # 'EQUAL',
            # 'COMMAND_SEPARATOR',
            # 'COLON',
            # 'OPEN_BRACKET',
            # 'CLOSE_BRACKET',
        ] + list(reserved.values())

def t_ID(t):
    r'[a-zA-Z]+\w*'
    t.type = reserved.get(t.value, 'ID')
    return t

def t_FLOAT(t):
    r'\-?\d+[\.|\,]+\d+'
    # t.value = float(t.value)
    return t

def t_NUMBER(t):
    r'\-?\d+'
    t.value = int(t.value)
    return t

def t_WHITESPACE(t):
    r'\s'
    pass

t_ASSIGNMENT = r':='
t_STRING_TERM = r'\w+'

def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)
```