МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт Информационных Технологий
Кафедра Вычислительной техники

ОТЧЕТ О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ №6

«Синтаксический анализатор»

по дисциплине
«Теория формальных языков**»**

Выполнил студент группы ИКБО-42-23 *Туляшева А.Т.*

Принял старший преподаватель *Боронников А.С.*

Практическая работа выполнена «__»_____2024 г.

«Зачтено» «__»_____2024 г.

Москва 2024

СОДЕРЖАНИЕ

ПОСТАНОВКА ЗАДАЧИ

Условие задачи: на выбранном языке программирования реализовать синтаксический анализатор, согласно варианту КР.

Согласно 9 варианту курсовой работы грамматика языка включает следующие синтаксические конструкции:

<операции_группы_отношения>::= <> | = | < | <= | > | >=

<операции_группы_сложения>::= + | - | or

<операции_группы_умножения>::= * | / | and

<унарная_операция>::= not

<программа> = {/ (<описание> | <оператор>) (: | переход строки) /} end

<описание>::= {<идентификатор> {, <идентификатор> } : <тип> ;}

<тип>::= % | ! | $

<оператор>::= <составной> | <присваивания> | <условный> | <фиксированного_цикла> | <условного_цикла> | <ввода> | <вывода>

<составной>::= «[» <оператор> { (: | перевод строки) <оператор> } «]»

<присваивания>::= <идентификатор> as <выражение>

<условный>::= if <выражение> then <оператор> [else <оператор>]

<фиксированного_цикла>::= for <присваивания> to <выражение> do <оператор>

<условного_цикла>::= while <выражение> do <оператор>

<ввода>::= read «(»<идентификатор> {, <идентификатор> } «)»

<вывода>::= write «(»<выражение> {, <выражение> } «)»

Многострочные комментарии в программе { … }

Выражения языка задаются правилами:

<выражение>::=<операнд>{<операции_группы_отношения><операнд>

<операнд>::=<слагаемое>{<операции_группы_сложения><слагаемое>}

<слагаемое>::=<множитель>{<операции_группы_умножения><множитель>}

<множитель>::= <идентификатор> | <число> | <логическая_константа> |<унарная_операция> <множитель> | «(»<выражение>«)»

<число>::= <целое> | <действительное>

<логическая_константа>::= true | false

Правила, определяющие идентификатор, букву и цифру:

<идентификатор>::= <буква> {<буква> | <цифра>}

<буква>::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T

<цифра>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Правила, определяющие целые числа:

<целое>::= <двоичное> | <восьмеричное> | <десятичное> |

<шестнадцатеричное>

<двоичное>::= {/ 0 | 1 /} (B | b)

<восьмеричное>::= {/ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 /} (O | o)

<десятичное>::= {/ <цифра> /} [D | d]

<шестнадцатеричное>::= <цифра> {<цифра> | A | B | C | D | E | F | a | b | c

Правила, описывающие действительные числа:

<действительное>::= <числовая_строка> <порядок> |

[<числовая_строка>] . <числовая_строка> [порядок]

<числовая_строка>::= {/n<цифра> /}

<порядок>::= (E | e)[+ | -] <числовая_строка>

Здесь для записи правил грамматики используется форма Бэкуса-Наура (БНФ). В записи БНФ левая и правая части порождения разделяются символом “::=”, нетерминалы заключены в угловые скобки, а терминалы – просто символы, используемые в языке.

РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

Будем считать, что лексический и синтаксической анализаторы взаимодействуют следующим образом. Если синтаксическому анализатору для анализа требуется очередная лексема, он запрашивает ее у лексического анализатора. Таким образом, разбор исходного текста программы идет под управлением подпрограммы синтаксического анализатора.

Разработку синтаксического анализатора проведем с помощью метода рекурсивного спуска (РС). В основе метода лежит тот факт, что каждому нетерминалу ставится в соответствие рекурсивная функция. Для того, чтобы в явном виде представить множество рекурсивных функций, перепишем грамматические правила следующим образом:

P → { D S }

D → I as TYPE ; D | ε

S → [S']

S' → IF E then S S'' | FOR A to E do S | WHILE E do S | READ (I) ; | WRITE (E) ; | I as E ; | S

S'' → else S | ε

E → E and R | E or R | R

R → R REL_OP A | A

A → A ADD_OP M | M

M → M MUL_OP U | U

U → (E) | not U | TRUE | FALSE | NUMBER | I

TYPE → % | ! | $

Где: P — программа (корневой блок), D — декларации, S — оператор (или последовательность операторов), S' — операторы, включая ветвления, циклы, ввод/вывод, и присваивание, S'' — ветвь else, E — выражение, R — выражение с операторами отношений, A — аддитивное выражение, M — мультипликативное выражение, U — унарное выражение или базовые элементы, TYPE — тип данных (% для целых, ! для вещественных, $ для булевых).

Грамматические правила разделены на более мелкие части, что облегчает понимание структуры программы. Исходный код синтаксического анализатора приведен в листинге 1.

СЕМАНТИЧЕСКИЙ АНАЛИЗАТОР

Некоторые особенности модельного языка не могут быть описаны контекстно-свободной грамматикой. Основные этапы семантического анализа:

канирование программы: на этом этапе происходит разбор исходного кода программы на токены, которые соответствуют грамматическим правилам (например, DIM, IDENTIFIER, ASSIGN, NUMBER, и т.д.).

роверка деклараций и типов данных: каждое объявление переменной должно содержать тип, например: “ x : !;”. При этом x должно быть идентификатором, а ! — одним из допустимых типов данных. Для этого нужно собрать все объявления в структуру данных (например, таблицу символов), чтобы проверять, что переменные использованы в программе правильно, с учетом их типов.

роверка присваиваний: присваивание должно быть совместимо с типом переменной. Например: “x as 10;”. Если x объявлена как %, то присваивание значения типа % или выражения, которое вычисляется в %, будет корректным.

роверка использования переменных: необходимо проверять, что переменные используются после их объявления. Например: “y as x + 10;”. Для этого можно поддерживать список всех объявленных переменных в ходе анализа программы и проверять, что каждая переменная используется только после ее объявления.

роверка выражений и совместимости типов: все выражения должны быть проверены с точки зрения типов операндов.

КОД ПРОГРАММЫ

Для реализации синтаксического анализатора был написал класс Parser.

Листинг 1 – синтаксический анализатор

```
class Parser:
    def __init__(self, token_list):
        self.token_list = token_list  # список токенов
        self.index = 0  # текущая позиция
        self.registered_variables = set()
        self.parsing_successful = True  # флаг успешности анализа
    def execute(self):  # запуск анализа блока программы
        try:
            self.root_block()
        except Exception as e:
            self.parsing_successful = False
            print(f"Ошибка синтаксического анализа: {e}")
    def active_token(self):  # текущий токен
        if self.index < len(self.token_list):
            return self.token_list[self.index]
        return None
    def consume(self, token_kind):  # для потребления токена определенного типа
        token = self.active_token()  # текущий
        if token and token[0] == token_kind:
            self.index += 1
        else:
            self.report_error(token_kind)
    def report_error(self, expected):  # обработка ошибок
        token = self.active_token()
        current_position = self.index + 1  # текущая позиция
        self.parsing_successful = False
        if token:
            print(f"Ошибка в позиции {current_position}: ожидалось
'{expected}', но найдено '{token[0]}' ('{token[1]}').")
        else:
            print(f"Ошибка в позиции {current_position}: ожидалось
'{expected}', но достигнут конец ввода.")

    def root_block(self):  # анализ основного блока
        self.consume('LBRACE')  # {
        while self.active_token() and self.active_token()[0] != 'RBRACE':  # }
            self.statement_or_declaration()  # анализ выражения или объявления
        self.consume('RBRACE')
    def statement_or_declaration(self):  # обработка выражений или объявлений
        while self.active_token() and self.active_token()[0] == 'COMMENT':
            self.consume('COMMENT')  # пропуск комментариев

        if self.active_token()[0] == 'IDENTIFIER':
            identifiers = self.gather_identifiers()
            if self.active_token() and self.active_token()[0] == 'ASSIGN':
                self.consume('ASSIGN')
                if self.active_token()[0] in ('IDENTIFIER', 'NUMBER', 'ADD_OP',
'MUL_OP', 'REL_OP'):
                    self.assignment_logic(identifiers[0])
                elif self.active_token()[0] in ('INT', 'REAL', 'BOOL'):
                    declared_type = self.active_token()[0]
                    self.consume(declared_type)
                    for ident in identifiers:
                        self.include_variable(ident)
                    self.handle_semicolon(True)
                else:
                    self.report_error("тип или выражение после 'as'")
            else:
                self.report_error("'as'")
        else:
            self.general_statement()
```

Продолжение листинга 1

```
        def gather_identifiers(self):   # сбор списка идентификаторов через
запятую
        ident_list = [self.active_token()[1]]
        self.consume('IDENTIFIER')
        while self.active_token() and self.active_token()[0] == 'COMMA':
            self.consume('COMMA')
            ident_list.append(self.active_token()[1])
            self.consume('IDENTIFIER')
        return ident_list  # список идентификаторов
    def general_statement(self):  # обработка общих операторов программы
        while self.active_token() and self.active_token()[0] == 'COMMENT':
            self.consume('COMMENT')
        token = self.active_token()
        if token[0] == 'IF':
            self.condition_structure()
        elif token[0] == 'FOR':
            self.iterative_structure()
        elif token[0] == 'WHILE':
            self.loop_structure()
        elif token[0] == 'READ':
            self.input_structure()
            self.handle_semicolon(True)
        elif token[0] == 'WRITE':
            self.output_structure()
            self.handle_semicolon(True)
        elif token[0] == 'LBRACE':
            self.composite_structure()
        elif token[0] == 'IDENTIFIER':
            self.assignment_logic()
            self.handle_semicolon(True)
        else:
            self.report_error("оператор или ключевое слово")
    def condition_structure(self):  # обработка if
        self.consume('IF')
        self.evaluate_expression()
        self.consume('THEN')
        self.general_statement()
        if self.active_token() and self.active_token()[0] == 'ELSE':
            self.consume('ELSE')
            self.general_statement()
        self.handle_semicolon()
    def iterative_structure(self):  # обработка for
        self.consume('FOR')
        self.assignment_logic()
        self.consume('TO')
        self.evaluate_expression()
        self.consume('DO')
        self.general_statement()

    def loop_structure(self):  # обработка while
        self.consume('WHILE')
        self.evaluate_expression()
        self.consume('DO')
        self.general_statement()

    def input_structure(self):  # обработка read
        self.consume('READ')
        self.consume('LPAREN')
        variables = self.gather_identifiers()
        self.verify_variables_exist(variables)
        self.consume('RPAREN')
```

Продолжение листинга 1

```
    def output_structure(self):  # обработка write
        self.consume('WRITE')
        self.consume('LPAREN')
        self.collect_expressions()
        self.consume('RPAREN')

    def composite_structure(self):  # для обработки составных блоков
        self.consume('LBRACE')
        while self.active_token() and self.active_token()[0] != 'RBRACE':
            self.general_statement()  # обработка оператора внутри блока
        self.consume('RBRACE')

    def assignment_logic(self, variable=None):  # для обработки присваивания
        if not variable:
            variable = self.active_token()[1]  # если переменная не передана,
берем тек. идентификатор
            self.consume('IDENTIFIER')
        self.verify_variables_exist([variable])
        self.consume('ASSIGN')
        self.evaluate_expression()

    def collect_expressions(self):  # для сбора выражений в списке
        self.evaluate_expression()  # первое выражение
        while self.active_token() and self.active_token()[0] == 'COMMA': # пока
идут ,
            self.consume('COMMA')
            self.evaluate_expression()
    def evaluate_expression(self):
        self.handle_logic_expression()

    def handle_logic_expression(self):
        self.handle_relational_expression()
        while self.active_token() and self.active_token()[0] in ('AND', 'OR'):
            self.consume(self.active_token()[0])
            self.handle_relational_expression()
    def handle_relational_expression(self):
        self.handle_additive_expression()
        while self.active_token() and self.active_token()[0] == 'REL_OP':
            self.consume('REL_OP')
            self.handle_additive_expression()

    def handle_additive_expression(self):
        self.handle_multiplicative_expression()
        while self.active_token() and self.active_token()[0] == 'ADD_OP':
            self.consume('ADD_OP')
            self.handle_multiplicative_expression()

    def handle_multiplicative_expression(self):  # для обработки выражений
        self.process_operand()  # обработка операнда
        while self.active_token() and self.active_token()[0] == 'MUL_OP':
            self.consume('MUL_OP')
            self.process_operand()

    def process_operand(self):  # для обработки операндов
        token = self.active_token()
        if token[0] == 'IDENTIFIER':
            self.verify_variables_exist([token[1]])
            self.consume('IDENTIFIER')
        elif token[0] == 'NUMBER':
            self.consume('NUMBER')
        elif token[0] in ('TRUE', 'FALSE'):
```

Продолжение листинга 1

```
            self.consume(token[0])
        elif token[0] == 'NOT':
            self.consume('NOT')
            self.process_operand()
        elif token[0] == 'LPAREN':
            self.consume('LPAREN')
            self.evaluate_expression()
            self.consume('RPAREN')
        else:
            self.report_error("идентификатор, число или выражение")

    def handle_semicolon(self, required=False):  # для обработки ;
        if self.active_token() and self.active_token()[0] == 'SEMICOLON':
            self.consume('SEMICOLON')
        elif required:
            self.report_error("';'")

    def verify_variables_exist(self, variables):   # проверка существования
переменной
        for variable in variables:
            if variable not in self.registered_variables:
                raise    SyntaxError(f"Семантическая    ошибка:    переменная
'{variable}' не объявлена.")

    def include_variable(self, variable):  # для регистрации переменной
        if variable in self.registered_variables:
            raise SyntaxError(f"Семантическая ошибка: переменная '{variable}'
уже объявлена.")
        self.registered_variables.add(variable)
```

ТЕСТИРОВАНИЕ

В отдельном текстовом файле представлен код программы, соответствующий синтаксису языка. На рисунках 1-3 были проведены тестирования.

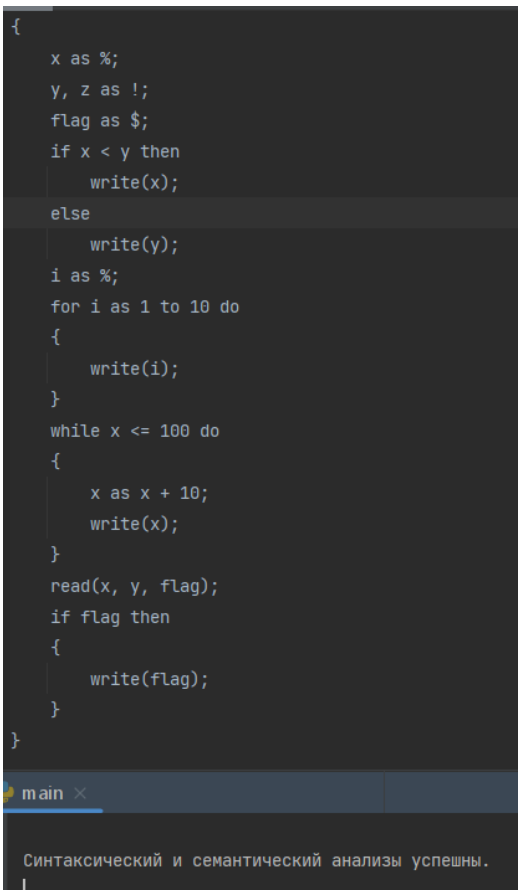

Рисунок 1 – Пример синтаксически корректной программы

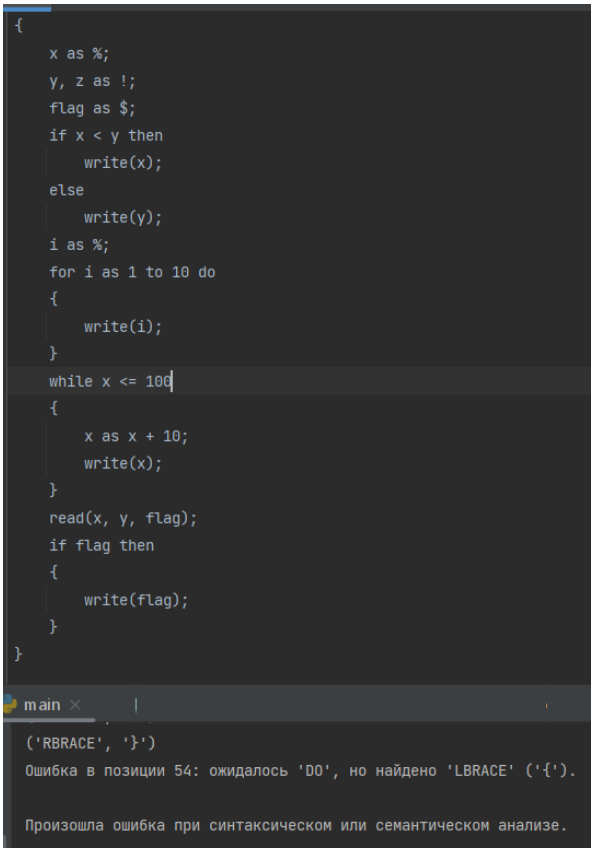

Рисунок 2 - Пример некорректной программы

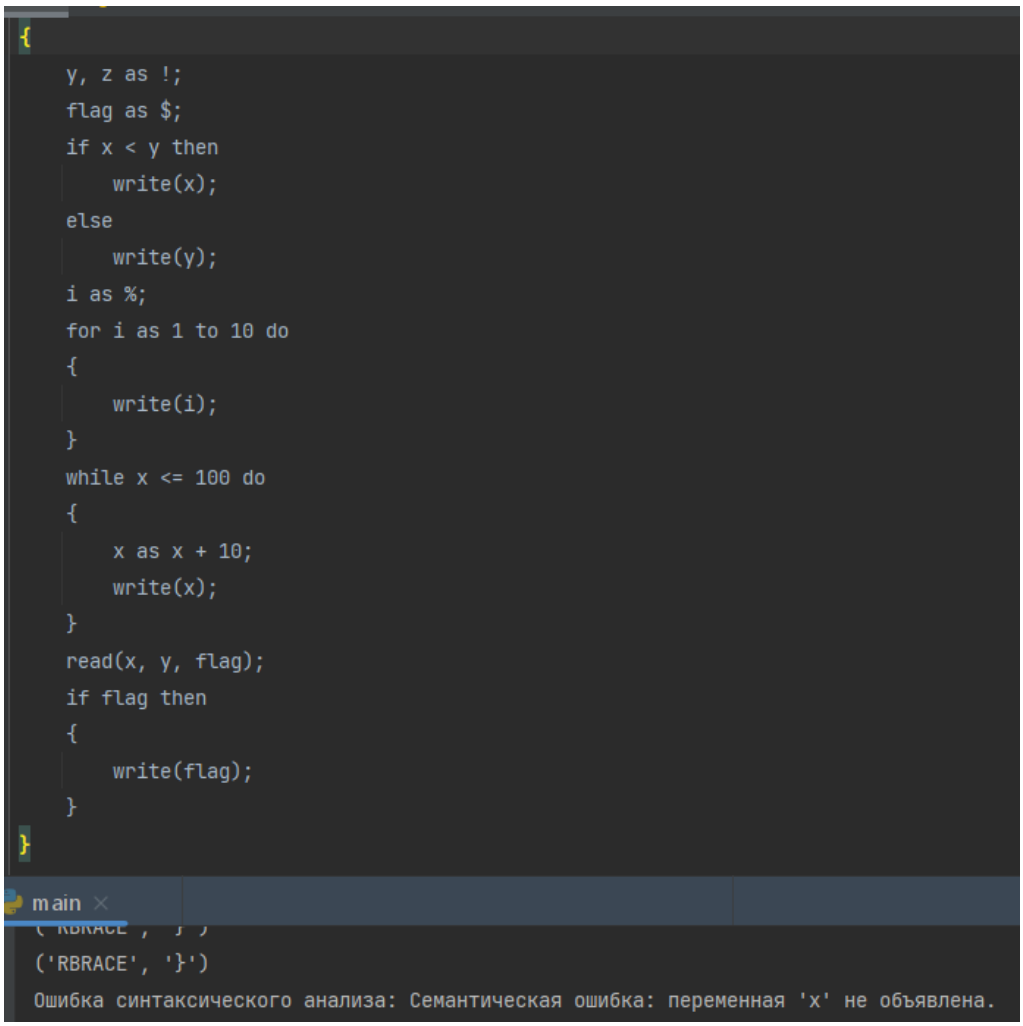

Рисунок 3 – Пример некорректной программы