Министерство образования Республики Беларусь
Учреждение образования
Белорусский государственный университет информатики и
Радиоэлектроники

Факультет информационных технологий и управления
Кафедра интеллектуальных информационных технологий

Отчет по лабораторной работе №5
по курсу “Языковые процессоры интеллектуальных систем”
Вариант 22

Выполнил:
Студент гр. 221701 Слобода Н. С.

Проверил: Соколович М. Г.

Минск 2025

1. Спецификация разработанного языка

1.1. Назначение и ключевые возможности

Язык vecmatLang ориентирован на работу с векторами и матрицами и поддерживает создание своих собственных структурных типов (class). Он включает неявное объявление переменных и блоков (INDENT, DEDENT), многоцелевое присваивание, , перегрузку подпрограмм по количеству аргументов, передачу параметров по значению и возвращаемому значению, а также полноценные управляющие конструкции (for, while, if-then-else, until). Целевой код — CPython (.pyc).

- Встроенные типы: int, float, vector, matrix, string, class, bool, void, unknown.
- Встроенные функции: read(), write(), len().
- Области видимости: глобальная, класс, метод/функция,.
- Подпрограммы: передача по значению и возвращаемому значению (return), объявление только в начале программы.
- Перегрузка: на основании имени и количества аргументов.
- Классы: конструктор по умолчанию, поля доступны извне для чтения, но не для перезаписи, методы (могут менять поля) и статичные функции (не имеют доступа к полям).

1.2. Синтаксис объявления переменных, подпрограмм и структурных типов (class)

В языке поддерживается неявное объявление переменных, это значит что переменная существует только после присваивания и ее тип определяется динамически (динамическая типизация). А при объявлении подрограммы достаточно указать ключевое слово ‘func’, название функции, список аргументов и любой блок кода. Также в языке есть возможность многоцелевого присваивания.

- Обычное присваивание:
    - a = 7
    - b = 3.67
    - v = [2, 4, 5]

- Многоцелевое присваивание:
    - a, b = 4, 5
    - c, d = f(6, 7)

- Объявление подпрограмм:
    - func a():
        return 6

- func s(a, b):
  ```
  t = a
  a = b + 2
  b = t - 2
  return b, a
  ```

Для объявления структурных типов используется ключевое слово “class”, затем название класса, список параметров инициализации, потом идет тело класса. В теле класса можно объявить его поля и методы (подпрограммы для класса, ключевое слово “method”), также есть возможность объявления функций, но в отличие от методов они могут работать без инициализации класса, однако не имеют доступа к полям класса. Поля класса могут изменяться только в теле класса и его методах.

- Объявление структурных типов (class):
  - class A():
    ```
    a = 5
    b = 2

    func f():
        return 9
    ```
  - class Point(a, b):
    ```
    x = a
    y = b

    method change(new_x, new_y):
        x = new_x
        y = new_y
    ```

1.3. Операции над данными

В языке присутствует большое количество операций над различными типами данных, в нем поддерживаются следующие типы и операции над ними:

- Логический тип: b
  - Или: b || b
  - И: b && b
  - Отрицание: !b

- Числовые данные: n (целые: int, дробные: float)
  - Арифметика: n + n, n - n, n * n, n / n
  - Сравнение: (n > n, n < n, n >= n, n <= n, n == n, n != n) возвращают b

- Вектора: v (могут быть только числовыми)
    - Арифметика: v + v, v - v, v * v возвращает n, n * v, v * n, v / n
    - Индексация: v[n] возвращает n

  Для сложение, вычитания и умножения векторов они должны быть одного размера.

- Матрицы: m (могут быть только числовыми)
    - Арифметика: m + m, m - m, m * m, m * v, v * m
    - Индексация:m[n] возвращает v

  Для сложение, вычитания матриц они должны быть одного размера. Для перемножения матриц и векторов они должны быть совместимы (m1 * m2 число столбцов m1 должно равняться числу строк m2)

- Строки: s
    - Конкатенация: s + s
    - Повторение строки: s * n

- Структуры: c
    - Инициализация: c(...)
    - Доступ к полю: c.id (id - идентификатор поля)

- Встроенные функции
    - Вывод: write(...)
    - Ввод: read(...)
    - Взятие длины: len(v), len(m)

1.4. Синтаксис управляющих конструкций

Данный язык поддерживает большинство стандартных управляющих конструкций (циклы, условное ветвление, операторы передачи управления).

- Циклы
    - for i in range(1, 5) # цикл с итерациями range обязателен
        b = b + i
    - while a < 10 # цикл работает пока условие истинно
        b = c - a
        a = a + 1
    - until |vec| = 1 # цикл работает пока условие ложно
        vec = vec / |vec|

В цикле с итерациями обязательно требуется range, эта конструкция генерирует значения для итеративной переменной (i) по правилу согласно переданным параметрам: range(a - начальное значение, b - конечное значение, c - шаг итерации), a, b, c типа int.

- Условное ветвление
    - if a > b then
        write(“a > b”)
      else
        write(“a <= b”)
    - if a == b then
        write(“a = b”)

Есть два варианта оператора условного ветвления if: if-then (если условие истинно - выполняется блок кода после “then”), if-then-else (и условие истинно - выполняется блок кода после “then”, иначе - блок кода после “else”).

- Операторы передачи управления
    - break - выходит из текущего цикла;
    - continue - пропускает текущую итерацию цикла и переходит к следующей;
    - return - завершает выполнение функции и передает управление вызываемой программе, может возвращать значение.

2. Файл грамматики разработанного языка

```
/*
Файл грамматики для vecmatLang
vecmatLang - Язык для работы с векторами и матрицами
с поддержкой создания своих собственных структурных типов
основные особенности:
* неявное объявление переменных
* множественное присваивание
* объявление функций только в начале программы
* перегрузка подпрограмм
* неявный блочный оператор (INDENT, DEDENT)
* объявление функций только в начале программы
* операторы цикла: for, while, until
* условный оператор if-then-else
* встроенные функции: read(), write(), len()
* векторное скалярное произведение и нахождеие нормы
* умножение матриц и нахождение нормы
* арифметические операции: +, -, *, /, ||
 */
grammar vecmatlang;

// Lexer (Основные токены языка)

// Невидимые токены
WS: [ ]+ -> skip; // Токен пробелов (игнорируется)
NEWLINE: '\r'? '\n'; // Токен, обозначающий переход на новую строку
TAB: '\t' -> skip; // токен табуляции (игнорируется)

// Токены неявного блочного оператора
/*
INDENT и DEDENT будут обрабатываться кастомным лексером indentation_lexer.py
только декларация для парсера
*/
INDENT: '<<INDENT>>';
DEDENT: '<<DEDENT>>';

// Токены ключевых слов языка
IF: 'if';
THEN: 'then';
ELSE: 'else';
FOR: 'for';
IN: 'in';
WHILE: 'while';
UNTIL: 'until';
FUNC: 'func';
RETURN: 'return';
WRITE: 'write';
READ: 'read';
RANGE: 'range';
CLASS: 'class';
METHOD: 'method';
CONTINUE: 'continue';
BREAK: 'break';
LEN: 'len';

// Токены логических операторов
AND: '&&'; // логическое И
OR: '||'; // логическое ИЛИ
NOT: '!'; // логическое отрицание
```

```
// Токены типов
/*
Нужны для реализации явного преобразования типов
такие токены существуют не для всех встроенных типов языка
а только для основных: целое, дробное, вектор, матрица
 */
INT_TYPE: 'int';
FLOAT_TYPE: 'float';
VECTOR: 'vector';
MATRIX: 'matrix';

// Токены литералов
/*
показывают как различные атомарные конструкции представлены в языке
идентификатор: ghost56, целое: 67, дробное: 78.7, строка: "fgfhghg"
 */
ID: [a-zA-Z_][a-zA-Z0-9_]*;
INT: [0-9]+;
FLOAT: [0-9]+ '.' [0-9]* | '.' [0-9]+;
STRING: '"' (~["\\\r\n] | '\\' ["\\])* '"';

COMMENT: '#' ~[\r\n]* -> skip; // Токен комментария в языке (игнорируется)


// Parser (Основные правила языка КС-грамматика)

// стартовое правило (корень синтаксического дерева)
/*
Программа может начинаться с пустых строк (NEWLINE*)
затем могут быть объявления классов (classDecl*)
после чего могут быть объявления функций (functionDecl*)
затем возможно идут различные statement блоки
в конце обязательно должен быть знак конца файла (EOF)
 */
program: NEWLINE* classDecl* functionDecl* statement* EOF;

// правило обьявления функции
/*
Начинается с ключевого слова "func" (FUNC)
после которого идет идентификатор функции (ID)
возможно, список параметров в скобках (parameterList?)
двоеточие, как знак окончания сигнатуры
далее минимум один переход на новую строку (NEWLINE+)
в конце блок кода (block)
 */
functionDecl: FUNC ID '(' parameterList? ')' ':' NEWLINE+ block;
parameterList: ID (',' ID)*; // правило, определяющие список параметров

// Правило построения блока кода
/*
Использует токены неявного блочного оператора (INDENT, DEDENT)
между ними должен быть минимум один statement блок
 */
block: INDENT statement+ DEDENT;
```

```
// правило объявления класса
/*
Начинается с ключевого слова "class" (CLASS)
после которого идет идентификатор класса (ID)
возможно, список параметров в скобках (parameterList?)
двоеточие, как знак окончания сигнатуры
далее минимум один переход на новую строку (NEWLINE+)
в конце тело класса (classBody)
 */
classDecl: CLASS ID '(' parameterList? ')' ':' NEWLINE+ classBody;

// Правило описывающее тело класса
/*
Использует токены неявного блочного оператора (INDENT, DEDENT)
между ними должен быть минимум один statement блок либо объявления метода или функции
 */
classBody: INDENT (statement | methodDecl | functionDecl)+ DEDENT;

// правило объявления метода
/*
Начинается с ключевого слова "method" (METHOD)
после которого идет идентификатор метода (ID)
возможно, список параметров в скобках (parameterList?)
двоеточие, как знак окончания сигнатуры
далее минимум один переход на новую строку (NEWLINE+)
в конце блок кода (block)
 */
methodDecl: METHOD ID '(' parameterList? ')' ':' NEWLINE+ block;

// Токены обращенич к полям и методам класса
fieldAppeal: ID '.' ID;
methodAppeal: ID '.' ID '(' argumentList? ')';

// правило описывающее все главные конструкции языка (основные потомки корня в синтаксическом дереве)
statement
    : assignment NEWLINE? // присваивание
    | ifStatement // условный оператор if-then-else
    | forStatement // оператор цикла for
    | whileStatement // оператор цикла while
    | untilStatement // оператор цикла until
    | expression NEWLINE? // любое выражение
    | returnStatement NEWLINE? // возрат значения из функции
    | writeStatement NEWLINE?  // функция вывода write
    | readStatement NEWLINE? // функция ввода read
    | CONTINUE NEWLINE? // оператор управления continue
    | BREAK NEWLINE? // оператор управления break
    | NEWLINE // пустая строка
    ;

// правило описывающее переменную
var
    : ID // обычная переменная
    | fieldAppeal // переменная как поле класса
    | ID '[' expression ']' // индексированная переменная
    ;

// правило присваивания
assignment
    : singleAssignment // одиночное присваивание
    | multipleAssignment // многоцелевое присваивание
    ;
```

```
singleAssignment: var '=' expression; // правило построения одиночного присваивания

// правило построения многоцелевого присваивания
/*
Перед оператором присваивания может быть перечисление переменных, минимум 2
после оператора присваивания может быть либо вызов функции (primaryExpression), либо список минимум из 2 выражений
 */
multipleAssignment: var (',' var)+ '=' (primaryExpression | expression (',' expression)+);

// правило условного оператора if-then-else
/*
В начале ключевое слово "if" (IF)
затем любое логическое выражение (expression)
После этого ключевое слово "then" (THEN)
далее минимум один переход на новую строку (NEWLINE+)
после всего блок кода (block)
в конце возможно дополнительное ветвление ((ELSE NEWLINE+ block)?)
 */
ifStatement:
    IF expression THEN NEWLINE+
    block
    (ELSE NEWLINE+ block)?;

// Правило оператора цикла
/*
В начале ключевое слово "for" (FOR)
затем идетификатор переменной итератора цикла (ID)
после этого ключевое слово "in" (IN)
далее вызов функции range с одним, двумя или тремя аргументами
в дальнейшем минимум один переход на новую строку (NEWLINE+)
в конце блок кода (block)
 */
forStatement:
    FOR ID IN RANGE '(' expression (',' expression)? (',' expression)? ')' NEWLINE+
    block;

// Правило оператора цикла
/*
В начале ключевое слово "while" (WHILE)
затем любое логическое выражение (expression)
далее минимум один переход на новую строку (NEWLINE+)
в конце блок кода (block)
 */
whileStatement:
    WHILE expression NEWLINE+
    block;

// Правило оператора цикла
/*
В начале ключевое слово "until" (UNTIL)
затем любое логическое выражение (expression)
далее минимум один переход на новую строку (NEWLINE+)
в конце блок кода (block)
 */
untilStatement:
    UNTIL expression NEWLINE+
    block;

returnStatement: RETURN argumentList?; // Правило конструкции возврата значения из функции
writeStatement: WRITE '(' argumentList? ')'; // Правило вызова фцнкции вывода
readStatement: READ '(' argumentList? ')'; // Правило вызова функции ввода
```

```
// Правило описывающее все возможные выражения в языке
expression
    : primaryExpression                                          #primaryExpr // первичное (не рекурсивное) выражение
    | '|' expression '|'                                         #normExpr // выражение нормы/модуля
    | '-' expression                                             #unaryMinusExpr // выражение унарного минуса
    |'(' expression ')'                                          #parenExpr // выражение в скобках
    | expression '[' expression ']'                              #indexExpr // выражение индексации
    | expression ('*' | '/') expression                          #mulDivExpr // выражение умножения/деления
    | expression ('+' | '-') expression                          #addSubExpr // выражения сложения/вычитания
    | expression ('>' | '<' | '>=' | '<=' | '==' | '!=') expression #comparisonExpr // выражения сравнения
    | NOT expression                                             #notExpr // выражение логического отрицания
    | expression (AND | OR) expression                           #binlogicExpr // выражения логического И/ИЛИ
    ;

// правило типов (используется для вызова функций преобразования типов)
type
    : INT_TYPE // целое число
    | FLOAT_TYPE // число с плавающей точкой
    | VECTOR // вектор
    | MATRIX // матрица
    ;

// правило описывающее все первичные (не рекурсивные) выражения в языке
primaryExpression
    : ID '(' argumentList? ')'                  #funcCallExpr // вызов функции
    | var                                       #varExpr // переменная
    | methodAppeal                              #methodExpr // вызов метода класса
    | literal                                   #literalExpr // литерал
    | type '(' argumentList? ')'                #typeExpr // вызов функции преобразования типов
    | LEN '(' argumentList? ')'                 #lenExpr // вызов функции взятия длины
    | readStatement                             #readExpr // вызов функции ввода
    ;

argumentList: expression (',' expression)*; // правило описывающее список аргументов

// правило построения литералов
literal
    :INT                                                    #intLiteral // литерал целого числа
    | FLOAT                                                 #floatLiteral // литерал числа с плавающей точкой
    | '[' argumentList ']'                                  #vectorLiteral // векторный литерал
    | '[' '[' argumentList ']' (',' '[' argumentList ']')* ']' #matrixLiteral // матричный литерал
    | STRING                                                #stringLiteral // литерал строки
    ;
```

3. Описание дополнительно разработанных классов

С помощью средств ANTLR4 были сгенерированы некоторый базовый перечень классов для построения упрощенного компилятора языка (vecmatlangLexer, vecmatlangParser, vecmatlangListener). Также были разработаны отдельные классы для синтаксического анализатора (IndentationLexer, SyntaxErrorListener), построения таблицы символов (Type, Symbol, FunctionSymbol, ClassSymbol), семантического анализатора (SemanticAnalyzer), компилятора (PythonCompiler, Compiler).

3.1. Классы для синтаксического анализа

- **IndentationLexer(indentation_lexer.py)**
  Используется для программной обработки неявного блочного оператора и генерации INDENT, DEDENT токенов. Наследуется от vecmatlangLexer и имеет следующие поля и методы

- *indent_stack* - список для хранения уровней отступов, используется как stack при добавлении элемента повышается уровень отступа, при удалении - снижается.
- *tokens_queue* - список токенов, используется как структура данных очередь.
- *at_start_of_line* - флаг нахождения в начале осмысленной части строки кода (после всех отступов).
- *pending_indent_check* - флаг ожидания проверки отступа.
- *nextToken(self)* - переопределенный метод для получения следующего токена и генерации INDENT, DEDENT токенов.
- *_calculate_indent(self, token)* - метод для подсчета отступа на основании позиции токена, используется методом *nextToken.*
- *_create_token(self, token_type, text, line, column)* - метод для создания токена типа *token_type* с текстом *text* и расположением *line, column*, используется методом *nextToken.*

- **SyntaxErrorListener(syntaxerror_listener.py)**
  Используется для лучшей обработки и сбора синтаксических ошибок. Наследуется от antlr4.error.ErrorListener.ErrorListener и имеет следующие поля и методы

  - *errors* - список для хранения ошибок.
  - *tokens_queue* - список токенов, используется как структура данных очередь.
  - *error_count* - хранит текущее число ошибок.
  - *syntaxError(self, recognizer, offendingSymbol, line, column, msg, e)* - переопределение метода генерации синтаксической ошибки.
  - *print_errors(self)* - метод для вывода статистики слушателя ошибок.

Сама логика синтаксического анализа описана в syntax_analyzer.py. Но без создания отдельного класса. Там есть функция для чтения файлов примеров *read_vml(file_path),* сбора и получения всех вышеописанных классов *get_analyze_tools(code, error_listener),* самого синтаксического анализа файла исходного кода vecmatLang *analyze(file_path).*

3.1. Классы для составления таблицы символов

- **Type(symbols.py)**
  Класс для определения основных встроенных типов языка. Наследуется от enum.Enum и имеет следующие поля и методы:

  - *INT, FLOAT, BOOL, VECTOR, MATRIX, VOID, CLASS, STRING, UNKNOWN* - значения для этого Enum класса.

- *from_token(token_type: int)* - статический метод для получения типа по числовому представлению токена *token_type*.

- **Symbol(symbols.py)**
  Базовый класс для хранения символа в таблице символов. Имеет следующие поля:

  - *name* - идентификатор символа.
  - *type* - тип символа.
  - *line* - номер строки объявления этого символа.
  - *column* - номер колонки объявления этого символа.
  - *is_initialized* - флаг инициализирована ли переменная.
  - *class_ref* - ссылка на класс (его идентификатор), если символ является объектом некоторого класса.

- **FunctionSymbol(symbols.py)**
  Класс для хранения символа подпрограммы в таблице символов. Наследуется от Symbol. Имеет следующие поля и методы:

  - *params* - список параметров подпрограммы.
  - *is_method* - флаг является ли этот символ подпрограммы методом класса.
  - *overloads* - список существующих перегрузок этой подпрограммы.
  - *column* - номер колонки объявления этого символа.
  - *add_overload(self, params)* - метод для добавления новой перегрузки подпрограммы по списку параметров *params*.
  - *find_matching_overload(self, arg_count)* - метод для нахождения подходящей перегрузки по требуемому количеству аргументов *arg_count*.
  - *get_all_overloads_info(self)* - метод для получения информации о всех существующих перегрузках функции.

- **ClassSymbol(symbols.py)**
  Класс для хранения символа класса в таблице символов. Наследуется от Symbol. Имеет следующие поля и методы:

  - *fields* - список полей класса.
  - *methods* - список методов класса.
  - *static_funcs* - список функций класса.
  - *constructor_params* - список параметров инициализации класса.
  - *add_constructor_param(self, name)* - метод для добавления требуемых классом параметров инициализации.
  - *add_field(self, name, field_type, line, column)* - метод для добавления поля класса в виде символа.

- *add_method(self, name, return_type, params)* - метод для добавления метода класса в виде символа подпрограммы с истинным значением флага является ли это методом.
- *add_static_func(self, name, return_type, params)* - метод для добавления метода класса в виде символа подпрограммы.

3.2. Класс для семантического анализа

- **SemanticAnalyzer(semantic_analyzer.py)**
  Основной класс, реализующий все семантические проверки, также он генерирует семантические ошибки и предупреждения, для случаев когда семантика не нарушена, но есть риск возникновения ошибки во время исполнения. Наследуется от vecmatlangListener. Имеет следующие поля и методы:

  - *errors* - список синтаксических ошибок.
  - *warnings* - список предупреждений.
  - *symbol_table* - таблица символов.
  - *current_scope* - список областей видимостей, используется как структура данных стек.
  - *current_class* - символ текущего класса (при нахождении в теле класса).
  - *current function* - символ текущей подпрограммы (при нахождении в кодовом блоке функции).
  - *различные флаги для упрощения логики…*
  - *for_loop_vars* - хранит переменные итераторы для циклов с итерациями.
  - *различные методы, вызываемые главными методами обхода дерева…*
  - *enterFunctionDecl* **|** *exitFunctionDecl(self, ctx)* - метод для обработки объявления функции и ее кода (вход и выход из контекста).
  - *enterClassDecl* **|** *exitClassDecl(self, ctx)* - метод для обработки объявления класса и его кода (вход и выход из контекста).
  - *enterMethodDecl* **|** *exitMethodDecl(self, ctx)* - метод для обработки объявления метода и его кода (вход и выход из контекста).
  - *enterSingleAssignment* **|** *enterMultipleAssignment(self, ctx)* - метод для обработки одиночного **|** многоцелевого присваивания (только вход, поскольку не создается новая область видимости).
  - *различные методы для анализа выражений и совместимости типов переменных в них…*

- *enterVarStatement* **|** *enterReturnStatement(self, ctx)* - метод для обработки переменных **|** обработки возврата значения (только вход, поскольку не создается новая область видимости).
- enterForStatement **|** *exitForStatement(self, ctx)* - метод для обработки цикла с итерациями (вход и выход из контекста).

Для запуска семантического анализа файла с исходным кодом vecmatLang используется функция *analyze(file_path)* из этого же файла.

3.2. Классы для реализации компилятора

- **PythonCompiler(python_compiler.py)**
  Класс, реализующий промежуточную компиляцию vecmatLang в Python. Наследуется от vecmatlangListener. Имеет следующие поля и методы:

  - *output* - список строк сгенерированного кода.
  - *indent_level* - текущий уровень отступа.
  - *current_class* - имя текущего обрабатываемого класса.
  - *current_function* - имя текущей обрабатываемой функции.
  - *различные флаги для упрощения логики…*
  - *local_vars* - множество локальных переменных текущего контекста.
  - *class_fields* - множество полей текущего класса.
  - *function_params* - список параметров текущей функции.
  - *loop_stack* - стек для отслеживания вложенности циклов.
  - *symbol_table* - таблица символов.
  - *function_counter* - словарь для отслеживания перегруженных функций.
  - enterProgram | exitProgram(self, ctx) - метод для инициализации Python модуля и завершения генерации (вход и выход из контекста программы).
  - enterFunctionDecl | exitFunctionDecl(self, ctx) - метод для обработки объявления функции и ее кода (вход и выход из контекста).
  - enterClassDecl | exitClassDecl(self, ctx) - метод для обработки объявления класса и его кода (вход и выход из контекста).
  - enterMethodDecl | exitMethodDecl(self, ctx) - метод для обработки объявления метода и его кода (вход и выход из контекста).
  - enterSingleAssignment | enterMultipleAssignment(self, ctx) - метод для обработки одиночного и многоцелевого присваивания (только вход).
  - *различные методы, вызываемые главными методами обхода дерева…*
  - enterIfStatement | exitIfStatement(self, ctx) - метод для обработки условного оператора (вход и выход из контекста).

- enterForStatement | exitForStatement(self, ctx) - метод для обработки цикла for (вход и выход из контекста).
- enterWhileStatement | exitWhileStatement(self, ctx) - метод для обработки цикла while (вход и выход из контекста).
- enterUntilStatement | exitUntilStatement(self, ctx) - метод для обработки цикла until (вход и выход из контекста).

- **Compiler(compiler.py)**
  Основной класс компилятора vecmatLang, координирующий весь процесс компиляции от исходного кода до CPython. Имеет следующие поля и методы:

  - *semantic_analyzer* - экземпляр семантического анализатора.
  - *python_compiler* - экземпляр Python компилятора.
  - *compile(self, source_code, output_file=None)* - основной метод компиляции.
  - *compile_file(self, input_file, output_file=None)* - метод компиляции файла VML. Читает файл и использует метод compile.

4. Перечень генерируемых ошибок

4.1. Синтаксические ошибки

Синтаксические ошибки генерируются в случае когда исходный код не соответствует грамматике языка, которая описана в файле грамматики. По умолчанию они генерируются ANTLR, но для повышение информативности и удобства чтения они обрабатываются и собираются в SyntaxErrorListener(syntaxerror_listener.py). Общий формат вывода информации об ошибке:

- Номер ошибки (например “Ошибка #1:”)
- Расположение точки возникновения ошибки в коде (например “Строка: 4, Колонка: 3”)
- Сообщение от ANTLR
- Символ, на котором возникла ошибка

Пример вывода ошибки в консоль

```
Ошибка #1:
  Строка: 1, Колонка: 6
  Сообщение: extraneous input ',' expecting {<EOF>, '(', '[', '|', '-', NEWLINE, 'if', 'for', 'while', 'until', 'return', 'write',
'read', 'continue', 'break', 'len', '!', 'int', 'float', 'vector', 'matrix', ID, INT, FLOAT, STRING}
  Символ: ','
```

4.1. Семантические ошибки

Семантические ошибки генерируются в случае когда исходный код синтаксически корректен, но нарушает внутреннюю внутреннюю логику языка (несоответствие типов, неправильное количество аргументов при вызове функции). Данные ошибки генерируются и обрабатываются в SemanticAnalyzer(semantic_analyzer.py). Общий формат вывода информации об ошибке:

- Расположение точки возникновения ошибки в коде (например “Ошибка в строке 4, колонке 3:”)
- Некоторое сообщение об ошибке

Пример вывода ошибки в консоль

```
Ошибка в строке 4, колонке 4: Функция 'g' ожидает 2 аргументов, получено 1
```

Также при семантическом анализе генерируются предупреждения, ситуация, когда нет прямого нарушения логики языка, однако при исполнении, после компиляции есть риск возникновения ошибки, формат вывода информации практически идентичен.

Список генерируемых семантических ошибок:

- ***Использование названия существующей переменной для объявления функции*** (формат сообщения: Имя {id - идентификатор переменной} уже используется для другой сущности)
- ***Повторное объявление уже объявленного класса*** (формат сообщения: Класс {class - идентификатор класса} уже объявлен)
- ***Использование названия поля как параметр метода*** (формат сообщения: Параметр метода {param - идентификатор параметра} перекрывает поле класса с тем же именем)
- ***Перезапись поля класса вне класса*** (формат сообщения: Поле {field - название поля} объекта {obj - идентификатор проинициализированного класса} изменяется вне класса)
- ***Использование названия поля как параметр метода*** (формат сообщения: Параметр метода {param - идентификатор параметра} перекрывает поле класса с тем же именем)
- ***Обращение к полю без инициализации*** (формат сообщения: Объект {obj - идентификатор проинициализированного класса} не найден
- ***Попытка обращения к полю со стороны не-класса*** (формат сообщения: {obj - идентификатор проинициализированного класса} не является классом)

- ***Обращение к несуществующему полю*** (формат сообщения: Поле {field - название поля} не найдено в классе {class - идентификатор класса})
- ***Несоответствие возврата функции присваивающим переменным*** (формат сообщения: Функция {func - идентификатор функции} возвращает {returns - ожидаемое количество возвращаемых значений} значений, но присваивается {vars - количество присваивающих переменных} переменным)
- ***Применение унарного минуса не к числу, вектору или матрице*** (формат сообщения: Унарный минус применяется только к числам, векторам и матрицам)
- ***Применение унарного минуса не к числу, вектору или матрице*** (формат сообщения: Унарный минус применяется только к числам, векторам и матрицам)
- ***Не целый индекс*** (формат сообщения: Индекс должен быть int, получен {type - тип переданного значения})
- ***Индексация не по вектору или матрице*** (формат сообщения: Индексация применяется только к vector и matrix)
- ***Применение логического отрицания не к логическому типу*** (формат сообщения: Оператор '!' применяется только к bool)
- ***Применение унарного минуса не к числу, вектору или матрице*** (формат сообщения: Унарный минус применяется только к числам, векторам и матрицам)
- ***Использование параметра конструктора в методе*** (формат сообщения: Параметр конструктора {var - идентификатор параметра инициализации} недоступен в методах класса)
- ***Использование переменной до инициализации*** (формат сообщения: Идентификатор {var - идентификатор переменной} не найден)
- ***Не числовой вектор*** (формат сообщения: УВектор может содержать только числа)
- ***Вызов len не для вектора или матрицы*** (формат сообщения: Функция len() применяется только к vector и matrix)
- ***Вызов метода у конструктора класса*** (формат сообщения: Нельзя вызывать методы у конструктора класса)
- ***Несоответствие количества переданных аргументов количеству ожидаемых при вызове функции***(формат сообщения: Функция {func - идентификатор функции} ожидает {args - ожидаемое количество аргументов} аргументов, получено {arg_count - количество переданных аргументов})
- ***Разная длина строк у матрицы*** (формат сообщения: Строки матрицы имеют разную длину)
- ***Несоответствие типов операндов операции***(формат сообщения: Операция {op - операция} не поддерживается между {left - тип левого операнда} и {right - тип правого операнда})

5. Примеры работы компилятора

Поскольку .pyc хранит байт-код Python в бинарном виде, то для получения его удобно читаемого варианта лучше использовать дизассемблирование. Оно реализовано в compiled\disassembler.py.

Пример дизассемблированного кода из example1.vml

```
Disassembly of main:
 73           RESUME                   0

 74           LOAD_CONST               1 (3)
              STORE_FAST               0 (i1)

 75           LOAD_CONST               2 (2.86)
              STORE_FAST               1 (f1)

 76           LOAD_FAST_LOAD_FAST      1 (i1, f1)
              BINARY_OP               10 (-)
              STORE_FAST               2 (n1)

 77           LOAD_GLOBAL              0 (np)
              LOAD_ATTR                2 (array)
              PUSH_NULL
              BUILD_LIST               0
              LOAD_CONST               3 ((1, 2, 3))
              LIST_EXTEND              1
              LOAD_GLOBAL              0 (np)
              LOAD_ATTR                4 (float64)
              LOAD_CONST               4 (('dtype',))
              CALL_KW                  2
              STORE_FAST               3 (v1)

 78           LOAD_GLOBAL              0 (np)
              LOAD_ATTR                2 (array)
              PUSH_NULL
              BUILD_LIST               0
              LOAD_CONST               5 ((4, 5, 6))
              LIST_EXTEND              1
              LOAD_GLOBAL              0 (np)
              LOAD_ATTR                4 (float64)
              LOAD_CONST               4 (('dtype',))
              CALL_KW                  2
              STORE_FAST               4 (v2)

 79           LOAD_FAST                3 (v1)
              LOAD_GLOBAL              7 (_vml_vector_mult + NULL)
              LOAD_FAST_LOAD_FAST     66 (v2, n1)
              CALL                     2
```

Пример дизассемблированного кода из example2.vml

```
Disassembly of main:
 73           RESUME                   0

 74           LOAD_CONST               1 (<code object factorial at 0x0000021AB9F088B0
              MAKE_FUNCTION
              STORE_FAST               0 (factorial)

 80           LOAD_GLOBAL              1 (int + NULL)
              LOAD_GLOBAL              3 (read + NULL)
              LOAD_CONST               2 ('Enter num:')
              CALL                     1
              CALL                     1
              STORE_FAST               1 (num)

 81           LOAD_GLOBAL              5 (write + NULL)
              LOAD_CONST               3 ('Factorial of ')
              LOAD_FAST                1 (num)
              LOAD_CONST               4 (' equals ')
              LOAD_FAST                0 (factorial)
              PUSH_NULL
              LOAD_FAST                1 (num)
              CALL                     1
              CALL                     4
              POP_TOP

 82           LOAD_GLOBAL              6 (np)
              LOAD_ATTR                8 (array)
              PUSH_NULL
              BUILD_LIST               0
              LOAD_CONST               5 ((1, 2, 3))
              LIST_EXTEND              1
              LOAD_GLOBAL              6 (np)
              LOAD_ATTR               10 (float64)
              LOAD_CONST               6 (('dtype',))
              CALL_KW                  2
              STORE_FAST               2 (vec)

 83           LOAD_GLOBAL             13 (_vml_norm + NULL)
              LOAD_FAST                2 (vec)
              CALL                     1
```

Пример дизассемблированного кода из example3.vml

```
Disassembly of main:
 73           RESUME                   0

 74           LOAD_CONST               1 (<code object swap_2arg at 0x0000021AB9F056F0,
              MAKE_FUNCTION
              STORE_FAST               0 (swap_2arg)

 77           LOAD_CONST               2 (<code object swap_3arg at 0x0000021AA1B06410,
              MAKE_FUNCTION
              STORE_FAST               1 (swap_3arg)

 80           LOAD_GLOBAL              0 (np)
              LOAD_ATTR                2 (array)
              PUSH_NULL
              BUILD_LIST               0
              LOAD_CONST               3 ((1, 2, 3))
              LIST_EXTEND              1
              LOAD_GLOBAL              0 (np)
              LOAD_ATTR                4 (float64)
              LOAD_CONST               4 (('dtype',))
              CALL_KW                  2
              STORE_FAST               2 (x)

 81           LOAD_GLOBAL              0 (np)
              LOAD_ATTR                2 (array)
              PUSH_NULL
              LOAD_CONST               5 (4)
              LOAD_CONST               6 (5)
              BUILD_LIST               2
              LOAD_GLOBAL              0 (np)
              LOAD_ATTR                4 (float64)
              LOAD_CONST               4 (('dtype',))
              CALL_KW                  2
              STORE_FAST               3 (y)

 82           LOAD_FAST                0 (swap_2arg)
              PUSH_NULL
              LOAD_FAST_LOAD_FAST     35 (x, y)
              CALL                     2
              UNPACK_SEQUENCE          2
```

Пример дизассемблированного кода из example4.vml

```
Disassembly of main:
 73           RESUME                   0

 74           LOAD_BUILD_CLASS
              PUSH_NULL
              LOAD_CONST               1 (<code object VectorPair at 0x0000021AA1ACD6B0,
              MAKE_FUNCTION
              LOAD_CONST               2 ('VectorPair')
              CALL                     2
              STORE_FAST               0 (VectorPair)

 83           LOAD_FAST                0 (VectorPair)
              PUSH_NULL
              LOAD_GLOBAL              0 (np)
              LOAD_ATTR                2 (array)
              PUSH_NULL
              BUILD_LIST               0
              LOAD_CONST               3 ((1, 2, 3))
              LIST_EXTEND              1
              LOAD_GLOBAL              0 (np)
              LOAD_ATTR                4 (float64)
              LOAD_CONST               4 (('dtype',))
              CALL_KW                  2
              LOAD_GLOBAL              0 (np)
              LOAD_ATTR                2 (array)
              PUSH_NULL
              BUILD_LIST               0
              LOAD_CONST               5 ((3, 2, 1))
              LIST_EXTEND              1
              LOAD_GLOBAL              0 (np)
              LOAD_ATTR                4 (float64)
              LOAD_CONST               4 (('dtype',))
              CALL_KW                  2
              CALL                     2
              STORE_FAST               1 (vec_pair)

 84           LOAD_GLOBAL              7 (write + NULL)
              LOAD_CONST               6 ('Cos-similarity of vectors: ')
              LOAD_FAST                1 (vec_pair)
              LOAD_ATTR                8 (vec1)
```