ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Национальный исследовательский университет ИТМО»

Факультет «Программной Инженерии и Компьютерных Технологий»

Отчет о лабораторной работе №1

«Задание 1»

по дисциплине

«Языки Программирования»

Вариант 3

Выполнил:

Студент группы P4119

Эр Мерт.

Проверил:

Доцент факультета ПИиКТ, к.т.н.

Кореньков Ю. Д.

Задание:

Использовать средство синтаксического анализа по выбору, реализовать модуль для разбора текста в соответствии с языком по варианту. Реализовать построение по исходному файлу с текстом синтаксического дерева с узлами, соответствующими элементам синтаксической модели языка. Вывести полученное дерево в файл в формате, поддерживающем просмотр графического представления.

Общая часть синтаксической модели для всех вариантов:

```
identifier: "[a-zA-Z_][a-zA-Z_0-9]*"; // идентификатор

str: "\"[^\"\\]*(?:\\.[^\"\\]*)*\""; // строка, окруженная двойными кавычками
char: "'[^']'";                      // одиночный символ в одинарных кавычках
hex: "0[xX][0-9A-Fa-f]+";            // шестнадцатеричный литерал
bits: "0[bB][01]+";                  // битовый литерал
dec: "[0-9]+";                       // десятичный литерал
bool: 'true'|'false';                // булевский литерал

list<item>: (item (',' item)*)?; // список элементов, разделённых запятыми
```

Задание по варианту:

```
source: sourceItem*;

typeRef: {
      |builtin: 'bool'|'byte'|'int'|'uint'|'long'|'ulong'|'char'|'string';
      |custom: identifier;
      |array: typeRef '(' (',')* ')';
};

funcSignature: identifier '(' list<argDef> ')' ('as' typeRef)? {
      argDef: identifier ('as' typeRef)?;
};

sourceItem: {
      |funcDef: 'function' funcSignature (statement* 'end' 'function')?;
};

statement: {
      |var: 'dim' list<identifier> 'as' typeRef;// for static typing
      |if: 'if' expr 'then' statement* ('else' statement*)? 'end' 'if';
      |while: 'while' expr statement* 'wend';
      |do: 'do' statement* 'loop' ('while'|'until') expr;
      |break: 'break';
      |expression: expr ';';
};

expr: { // присваивание через '='
      |binary: expr binOp expr; // где binOp - символ бинарного оператора
      |unary: unOp expr; // где unOp - символ унарного оператора
      |braces: '(' expr ')';
      |callOrIndexer: expr '(' list<expr> ')';
      |place: identifier;
      |literal: bool|str|char|hex|bits|dec;
};
```

Лексер ANTLR4

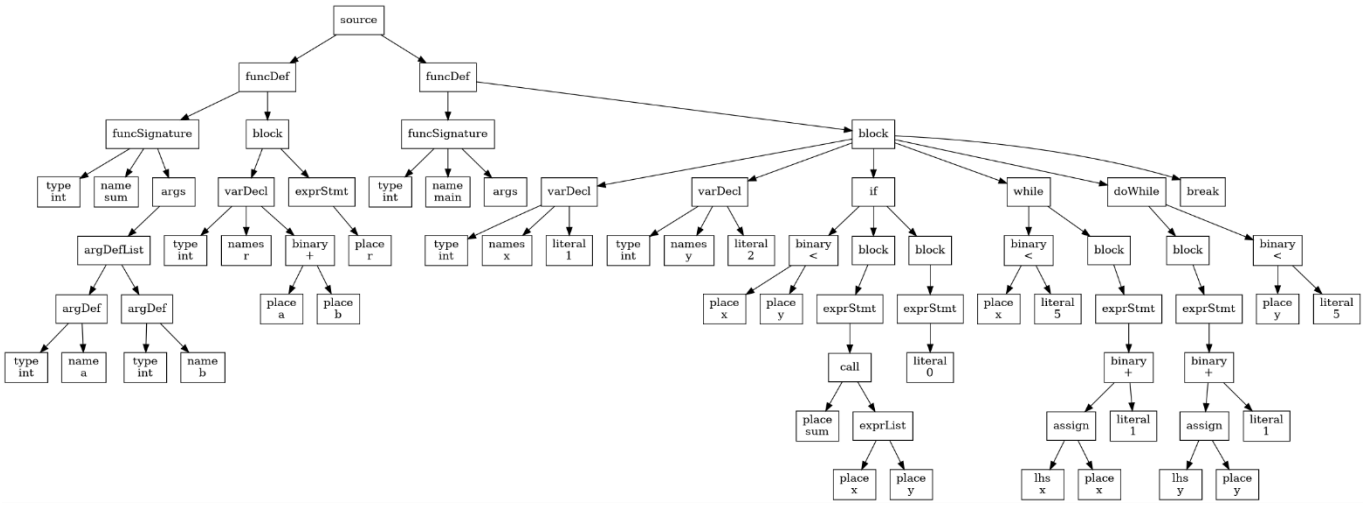
```
lexer grammar v3Lexer;

@header { package org.gen; }

// Пробелы
WS : [ \t\r\n]+ -> skip ;

// Управляющие конструкции
IF      : 'if'      ;
THEN    : 'then'    ;
ELSE    : 'else'    ;
WEND    : 'wend'    ;
END     : 'end'     ;
WHILE   : 'while'   ;
DO      : 'do'      ;
LOOP    : 'loop'    ;
UNTIL   : 'until'   ;
BREAK   : 'break'   ;

// Служебные слова
FUNCTION : 'function' ;
DIM      : 'dim' ;
AS       : 'as' ;

// Типы
TYPE    : 'bool'
        | 'byte'
        | 'uint'
        | 'int'
        | 'ulong'
        | 'long'
        | 'char'
        | 'string'
        ;
```

```
// Операторы
PLUS                : '+'   ;
MINUS               : '-'   ;
MUL                 : '*'   ;
DIV                 : '/'   ;
MOD                 : '%'   ;
LSHIFT              : '<<'  ;
RSHIFT              : '>>'  ;
EQUALS              : '=='  ;
NOT_EQUALS          : '!='  ;
LESS_OR_EQUALS      : '<='  ;
GREATER_OR_EQUALS   : '>='  ;
LESS                : '<'   ;
GREATER             : '>'   ;
ASSIGN              : '='   ;
LOGICAL_OR          : '||'  ;
LOGICAL_AND         : '&&'  ;
EXCLAMATION         : '!'   ;

// Разделители
L_PAREN         : '('    ;
R_PAREN         : ')'    ;
COMMA           : ','    ;
SEMICOLON       : ';'    ;

STR  : '"' ( '\\' . | ~["\\] )* '"' ;
CHAR : '\'' ~'\'' '\''              ;
HEX  : '0'[xX][0-9A-Fa-f]+          ;
BITS : '0'[bB][01]+                 ;
DEC  : [0-9]+                       ;
BOOL : 'true' | 'false'             ;

IDENTIFIER : [a-zA-Z_][a-zA-Z_0-9]* ;

COMMENT : '\'' ~[\r\n]* -> skip ;
```

Парсер ANTLR4

```
parser grammar v3Parser;

options { tokenVocab = v3Lexer; }

@header { package org.gen; }

source : sourceItem* ;

argDefList : (argDef (COMMA argDef)*)? ;
identifierList : (IDENTIFIER (COMMA IDENTIFIER)*)? ;
exprList       : expr (COMMA expr)* ;

typeRef : TYPE                            #BuiltIn
        | IDENTIFIER                      #Custom
        | typeRef L_PAREN (COMMA)* R_PAREN  #Array
        ;

funcSignature : IDENTIFIER L_PAREN argDefList R_PAREN (AS typeRef)? ;

argDef : IDENTIFIER (AS typeRef)? ;

sourceItem : funcDef ;

funcDef : FUNCTION funcSignature (statement* END FUNCTION)? ;

statement : DIM variableDeclList AS typeRef                      #Var
          | IF expr THEN statement* (ELSE statement*)? END IF    #If
          | WHILE expr statement* WEND                           #While
          | DO statement* LOOP (WHILE|UNTIL) expr                #Do
          | BREAK                                                #Break
          | expr SEMICOLON                                       #Expression
          ;

variableDeclList : variableDecl (COMMA variableDecl)* ;
variableDecl     : IDENTIFIER (L_PAREN DEC R_PAREN)? ;

expr           : assignmentExpr ;

assignmentExpr : logicOrExpr (ASSIGN assignmentExpr)? ;
logicOrExpr    : logicAndExpr (LOGICAL_OR logicAndExpr)* ;
logicAndExpr   : equalityExpr (LOGICAL_AND equalityExpr)* ;
equalityExpr   : relationalExpr ((EQUALS | NOT_EQUALS) relationalExpr)* ;
relationalExpr : additiveExpr ((LESS | LESS_OR_EQUALS | GREATER | GREATER_OR_EQUALS) additiveExpr)* ;
additiveExpr   : multiplicativeExpr ((PLUS | MINUS) multiplicativeExpr)* ;
multiplicativeExpr : unaryExpr ((MUL | DIV | MOD) unaryExpr)* ;

unaryExpr      : (PLUS | MINUS | EXCLAMATION) unaryExpr
               | postfixExpr
               ;

postfixExpr    : primaryExpr (L_PAREN exprList? R_PAREN)* ;

primaryExpr    : IDENTIFIER
               | literal
               | L_PAREN expr R_PAREN
               ;

literal        : BOOL | STR | CHAR | HEX | BITS | DEC ;
```

Вывод:



Описание работы

Программа принимает на вход исходный текст и выполняет следующие этапы:

1. Лексический анализ: Превращение текста в поток токенов.
2. Синтаксический анализ: Построение `ParseTree`.
3. AST Transformation: Очистка дерева от синтаксического "шума" (скобки, точки с запятой) и создание структуры `SourceNode` -> `FunctionNode` -> `StatementNode`.

Описание структур данных

В основе лежит иерархия классов в пакете `org.parser.ast.node`:

- `SourceNode`: Корень, содержащий список функций.
- `FunctionNode`: Хранит сигнатуру (имя, параметры, тип возврата) и тело (список `StatementNode`).
- `BinaryExprNode`: Хранит левый и правый операнды и перечисление `BinaryOp` (ASSIGN, PLUS, MUL и др.).
- `LiteralExprNode`: Хранит вид литерала (`LiteralKind`) и его строковое значение.

Дополнительная обработка

Для формирования AST потребовалась значительная обработка в `AstBuilder`:

1. Схлопывание приоритетов: В грамматике выражения разделены на уровни (additive, multiplicative), но в AST они превращаются в плоские узлы `BinaryExprNode`.
2. Обработка условий: В методе `visitIf` происходит разделение дочерних элементов на ветки `then` и `else` путем отслеживания терминального узла `ELSE`.
3. Типизация: Текстовые определения типов (`int`, `bool`) преобразуются в строго типизированные узлы `BuiltInTypeNode`.

5. Аспекты реализации

Пример реализации обхода выражений в `AstBuilder`:

```java
// additiveExpr : multiplicativeExpr ((PLUS | MINUS) multiplicativeExpr)* ;
@Override
public AstNode visitAdditiveExpr(v3Parser.AdditiveExprContext ctx) {
    ExprNode result = (ExprNode) visit(ctx.multiplicativeExpr(i: 0));
    for (int i = 1; i < ctx.multiplicativeExpr().size(); i++) {
        String opText = ctx.getChild(2 * i - 1).getText();
        BinaryOp op = opText.equals("+") ? BinaryOp.PLUS : BinaryOp.MINUS;
        ExprNode right = (ExprNode) visit(ctx.multiplicativeExpr(i));
        result = new BinaryExprNode(result, op, right);
    }
    return result;
}
```

Данный подход позволяет рекурсивно строить дерево выражений с учетом левой ассоциативности.

Вывод:

Реализован компиляторный модуль на Java с использованием ANTLR4, который разбирает исходный текст в синтаксическое дерево (AST) согласно заданной грамматике. Модуль отделяет логику парсинга от обработки ошибок, передавая список проблем тестовой программе для вывода в System.err. Для визуализации реализован обход дерева с экспортом в формат Mermaid Markdown, что позволяет генерировать графическое представление структуры кода.