Национальный исследовательский университет ИТМО
Мегафакультет компьютерных технологий и управления
Факультет программной инженерии и компьютерной техники

Отчет по практическому заданию №1

по курсу «Языки программирования»

Вариант: 3

Выполнил студент группы P4119 _____ В.Ю. Суровикин
(подпись)

Руководитель _____ Ю. Д. Кореньков
(подпись)

05 ноября 2025 г.

Санкт-Петербург
2025

Оглавление

1. Цель работы 2

2. План работы 2

3. Ход работы 2

3.1. Разработка синтаксического анализатора 2

3.2. Разработка тестовой программы 9

3.3. Тестирование разработанной грамматики 13

4. Вывод 19

1. Цель работы

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.

2. План работы

1. Изучить выбранное средство синтаксического анализа.
2. Изучить синтаксис разбираемого по варианту языка и записать спецификацию для средства синтаксического анализа.
3. Реализовать модуль, использующий средство синтаксического анализа для разбора языка по варианту.
4. Реализовать тестовую программу для демонстрации работоспособности созданного модуля.

3. Ход работы

3.1. Разработка синтаксического анализатора

Для разработки синтаксического анализатора была выбрана библиотека antlr4, на ней уже есть готовые анализаторы для C++ и других языков, что поможет при проектировании своего анализатора.

Выполним реализацию общей части 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
```

В antlr4 эти элементы будут описаны следующим образом:

Листинг 3.1. Базовые элементы грамматики

```
fragment DIGIT      : [0-9];
fragment HEXDIGIT   : [0-9A-Fa-f];
fragment BIN_DIGIT  : [01];

// numeric/string/char tokens
HEX  : '0' [xX] HEXDIGIT+ ;
BITS : '0' [bB] BIN_DIGIT+ ;
DEC  : DIGIT+ ;

// string with simple escape handling
STRING
    : '"' ( ~["\\\r\n] | '\\' . )* '"'
    ;

// character literal: 'a' or escaped '\n'
CHAR
    : '\'' ( ~['\\\r\n] | '\\' . ) '\''
    ;

// Combined token for bool literal
BOOL : TRUE | FALSE ;

// Identifiers: start with letter or underscore, then letters/digits/underscore
identifier
    : Identifier
    ;

Identifier
    : [a-zA-Z_] [a-zA-Z_0-9]*
    ;
```

Как можно заметить, в antlr4 грамматика описывается способом, похожим на описание формы Бэкуса-Наура, однако после этого компилируется в лексер и парсер на C++. В приведенном фрагменте отсутствует описание списка элементов т.к. он будет реализовываться для каждого конкретного случая отдельно.

Далее будет приведена реализация блоков для варианта 3. Разработка выполнялась снизу вверх, сначала expr, затем statement, а потом уже верхнеуровневое описание sourceItem.

Ниже приведен блок синтаксической модели для expr:

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

Аналогичным образом выглядит этот же блок в tree-sitter:

Листинг 3.2. Блок expr в antlr4

```
// Expressions with precedence (lowest first)
expr
    : expr OR expr                              # orExpr
    | expr AND expr                             # andExpr
    | expr (EQ | NEQ | LT | GT | LE | GE) expr  # compareExpr
    | expr (ADD | SUB) expr                     # addExpr
    | expr (MUL | DIV | MOD) expr               # mulExpr
    | expr ASSIGN expr                          # assignExpr
    | (NOT | SUB) expr                           # unaryExpr  // unary minus and not
    | primary                                   # primaryExpr
    ;
```

Каждый из полей ссылается на другое правило. Primary объединяет вызов функции, выражение в скобках, напрямую ссылается на идентификатор, объявленный раньше или использует literal:

Листинг 3.3. Поле literal в antlr4

```
// Primary expression and call/index chaining
primary
    : atom ( '(' argExprList? ')' )*            // call(s)
    ;

argExprList
    : expr (',' expr)*
    ;

atom
    : '(' expr ')'              // parens
    | identifier                // variable/place
    | literal                   // literal
    ;

// Literals
literal
    : BOOL
    | STRING
    | CHAR
    | HEX
    | BITS
    | DEC
    ;
```

Приоритет правил задается их расположением: чем ниже правило, тем ниже его приоритет. Это можно увидеть в файле парсера:

Листинг 3.4. Enum приоритетов операций

```
enum {
  RuleSource = 0, RuleSourceItem = 1, RuleFuncDef = 2, RuleFuncSignature = 3,
  RuleArgList = 4, RuleArgDef = 5, RuleTypeRef = 6, RuleTypeArgList = 7,
  RuleStatement = 8, RuleIdentifierList = 9, RuleExpr = 10, RulePrimary = 11,
  RuleArgExprList = 12, RuleAtom = 13, RuleLiteral = 14, RuleIdentifier = 15,
  RuleBuiltinType = 16
};
```

Таким образом были реализованы все блоки Expressions, далее будет рассмотрен следующий уровень, а именно Statement, основной «строительный блок» функции.

Ниже приведена синтаксическая модель, данная во втором варианте, которую необходимо реализовать:

```
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 ';';
};
```

В грамматике antlr4 он будет иметь следующий вид:

Листинг 3.5. Блок statement в грамматике tree-sitter.

```
// Statements
statement
    : DIM identifierList AS typeRef ';'?                # varDecl
    | IF expr THEN statement* (ELSE statement*)? END IF # ifStmt
    | WHILE expr statement* WEND                        # whileStmt
    | DO statement* LOOP (WHILE | UNTIL) expr           # doLoopStmt
    | BREAK ';'?                                        # breakStmt
    | expr ';'                                          # exprStmt
    ;

// identifier list like: a, b, c
identifierList
    : identifier (',' identifier)*
    ;
```

Таким образом был полностью обработан блок statement, для завершения грамматики осталось лишь разработать объявление функций:

```
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')?;
};
```

Здесь нет никаких особенностей, все пишется ровно так, как и написано, лишь с особенностями перевода на antlr4, код приведен ниже:

Листинг 3.6. Конструкции объявления функций и параметров

```
source
    : sourceItem* EOF
    ;

sourceItem
    : funcDef
    ;

funcDef
    : FUNCTION funcSignature statement* END FUNCTION
    ;

funcSignature
    : identifier '(' argList? ')' (AS typeRef)?
    ;

argList
    : argDef (',' argDef)*
    ;

argDef
    : identifier (AS typeRef)?
    ;

typeRef
    : builtinType                       # builtinTypeRef
    | identifier                        # customTypeRef
    | typeRef '(' typeArgList? ')'      # arrayTypeRef
    ;

typeArgList
    : typeRef (',' typeRef)*
    ;
```

Приведенная грамматика генерирует группу файлов. ANTLR-генератор (LangParser) создает дерево разбора (Parse Tree), которое напрямую отражает синтаксис, но не всегда удобно для дальнейшего анализа.

Чтобы получить более абстрактное представление программы, был реализован модуль ASTBuilder, выполняющий следующие задачи:

- Сворачивание избыточных уровней дерева

— например, несколько уровней expr → addExpr → mulExpr объединяются в одну иерархию.

- Создание обобщенных узлов

— например, для if, while, do loop создаются узлы IfStmt, WhileStmt, DoLoopStmt, в которые помещаются тело блока (Then/Else/Loop) и выражение-условие (CompareExpr).

- Обработка выражений присваивания

— выражения вида z = x + y * x + y разворачиваются в древовидную структуру

Далее будет создана тестовая программа, которая будет принимать грамматику и выводить дерево разбора, вместе с ошибками, возникшими в ходе разбора переданного файла.

3.2. Разработка тестовой программы

Тестовая программа также реализована на C++ как и парсер.

Первый этап – получение аргументов командной строки:

Листинг 3.7. Код для получения данных из командной строки

```
int main(int argc, const char *argv[]) {
    if (argc < 3) {
        std::cerr << "Использование: " << argv[0] << " <входной_файл> <выходной_файл>\n";
        return 1;
    }

    const std::string inputFile = argv[1];
    const std::string outputFile = argv[2];

    // Чтение входного файла
    std::ifstream stream(inputFile);
    if (!stream.is_open()) {
        std::cerr << "Ошибка: не удалось открыть файл " << inputFile << "\n";
        return 1;
    }
```

Код, приведенный выше может принимать грамматику в виде непосредственно пути к файлу грамматики, а также принимать путь файла, куда будет записано дерево.

Далее полученную грамматику передаем парсеру для дальнейшего построения дерева.

Листинг 3.8. Код для создания дерева

```
ANTLRInputStream input(stream);
LangLexer lexer(&input);
CommonTokenStream tokens(&lexer);
LangParser parser(&tokens);

// Подключаем сборщик ошибок
ErrorCollector errorCollector;
parser.removeErrorListeners();
parser.addErrorListener(&errorCollector);

// Запуск парсера
LangParser::SourceContext *tree = parser.source();

// Проверяем ошибки
auto errors = errorCollector.getErrors();
if (!errors.empty()) {
    std::cerr << "Ошибки парсера:\n";
    for (const auto &err : errors) {
        std::cerr << "  " << err.toString() << "\n";
    }
    return 1;
} else {
    std::cerr << "Парсинг завершен без ошибок.\n";
}

// Строим AST
ASTBuilder builder;
antlrcpp::Any result = builder.visitSource(tree);

// Извлекаем дерево
std::shared_ptr<ASTNode> ast;
try {
    ast = std::any_cast<std::shared_ptr<ASTNode>>(result);
} catch (const std::bad_any_cast &e) {
    std::cerr << "Ошибка при построении AST: " << e.what() << "\n
    return 1;
}
```

Последняя часть кода записывает в файл дерево в удобочитаемой форме.

Листинг 3.9. Код записи дерева в выходной файл

```
    // Проверяем, пустое ли дерево
    if (!ast) {
        std::cerr << "AST пуст.\n";
        return 1;
    }

    // Записываем дерево в выходной файл
    std::ofstream outFile(outputFile);
    if (!outFile.is_open()) {
        std::cerr << "Ошибка: не удалось открыть выходной файл " << outputFile << "\n";
        return 1;
    }

    printAST(ast, outFile);
    std::cerr << "AST записан в файл: " << outputFile << "\n";

    return 0;
}
```

Сама функция, которая отвечает за отрисовку дерева:

Листинг 3.10. Код функции вывода дерева

```
void printAST(const std::shared_ptr<ASTNode> &node, std::ostream &out, int indent = 0) {
    if (!node) return;
    out << std::string(indent, ' ') << node->name;
    if (!node->value.empty()) out << " (" << node->value << ")";
    out << "\n";
    for (const auto &child : node->children) {
        printAST(child, out, indent + 2);
    }
}
```

Ниже представлены два важных фрагмента: узел AST-дерева и структура, которая выдает в тестовой программе дерево и коллекцию ошибок.

Листинг 3.11. Код узла дерев

```
struct ASTNode {
    std::string name;   // тип узла
    std::string value;  // значение
    std::vector<std::shared_ptr<ASTNode>> children;

    ASTNode(const std::string& n) : name(n) {}
    ASTNode(const std::string& n, const std::string& v) : name(n), value(v) {}

    void addChild(std::shared_ptr<ASTNode> child) {
        if (child) children.push_back(child);
    }
};

using ASTNodePtr = std::shared_ptr<ASTNode>;
```

Листинг 3.12. Код структуры

```
struct ParseResult {
    ASTNodePtr root;              // nullptr при неуспешном построении AST
    std::vector<std::string> errors;
};
```

После этого можно переходить непосредственно к тестированию.

3.3. Тестирование разработанной грамматики

Для тестирования грамматики будем подавать входной файл, охватывающий большую часть грамматики, и смотреть на результат. Если он соответствует ожиданиям, значит все работает корректно.

Подадим следующий файл:

Листинг 3.13. Файл, подающийся в тестовую программу

```
function add(a as int, b as int) as int
    dim result as int
    if a > b then
        dim temp as int
        result = a + b;
    else
        dim temp1 as int
        result = a - b;
    end if
end function

function main()
    dim x as int
    dim y as int
    dim z as int

    x = 5;
    y = 10;
    z = x + y * x - y;
```

```
    while z > 0
        z = z - 1;
    wend

    do
        x = x + 1;
        if x > 10 then
            break;
        end if
    loop until x > 15
end function
```

Запустим код и получим файл разбора.

```
Source
  FuncDef (add)
    FuncSignature (add)
      ArgList
        ArgDef (a)
          BuiltinType (int)
        ArgDef (b)
          BuiltinType (int)
      BuiltinType (int)
    VarDecl
      IdentifierList
        Identifier (result)
      BuiltinType (int)
    IfStmt
      CompareExpr
        Identifier (a)
        Identifier (b)
      Then
        VarDecl
          IdentifierList
            Identifier (temp)
          BuiltinType (int)
        AssignExpr
          Identifier (result)
          AddExpr
            Identifier (a)
            Identifier (b)
      Else
        VarDecl
          IdentifierList
            Identifier (temp1)
          BuiltinType (int)
        AssignExpr
          Identifier (result)
          SubExpr
            Identifier (a)
            Identifier (b)
  FuncDef (main)
    FuncSignature (main)
    VarDecl
      IdentifierList
        Identifier (x)
      BuiltinType (int)
    VarDecl
      IdentifierList
        Identifier (y)
      BuiltinType (int)
    VarDecl
      IdentifierList
        Identifier (z)
      BuiltinType (int)
    AssignExpr
      Identifier (x)
```

```
      Literal (5)
    AssignExpr
      Identifier (y)
      Literal (10)
    AssignExpr
      Identifier (z)
      MulExpr
        AddExpr
          Identifier (x)
          Identifier (y)
        SubExpr
          Identifier (x)
          Identifier (y)
    WhileStmt
      CompareExpr
        Identifier (z)
        Literal (0)
      AssignExpr
        Identifier (z)
        SubExpr
          Identifier (z)
          Literal (1)
    DoLoopStmt
      AssignExpr
        Identifier (x)
        AddExpr
          Identifier (x)
          Literal (1)
      IfStmt
        CompareExpr
          Identifier (x)
          Literal (10)
        Then
          BreakStmt
      CompareExpr (UNTIL)
        Identifier (x)
        Literal (15)
```

Этот порядок полностью соответствует правильному, что свидетельствует о корректности разбора различных унарных и бинарных выражений, а также остальных конструкций.

Далее проведем тестирование на том же файле, но внесем в него предварительно пару ошибок.

```
Ошибки парсера:
  PARSER error at 4:13 - missing ';' at 'temp' (offending: "temp")
  PARSER error at 4:18 - mismatched input 'as' expecting ';' (offending: "as")
  PARSER error at 9:4 - missing ';' at 'end' (offending: "end")
  PARSER error at 20:5 - mismatched input ')' expecting {BOOL, HEX, BITS, DEC, STRING, CHAR, Identifier, '-', 'not', '('} (offending: ")")
  PARSER error at 22:4 - missing ';' at 'while' (offending: "while")
```

Здесь видно, что коллекция ошибок выводится корректно.

Таким образом было проведено полное тестирование разработанной грамматики.

4. Вывод

В ходе выполнения лабораторной работы был спроектирован и реализован синтаксический анализатор для заданного языка программирования с использованием библиотеки antlr4. В процессе работы была изучена структура и принципы построения грамматик на данном инструменте, реализованы все необходимые элементы языка: выражения, операторы, конструкции ветвления и циклов, а также механизм объявления функций и переменных. Полученная грамматика была протестирована на комплексном примере, демонстрирующем корректную работу парсера для всех предусмотренных структур и выражений, включая вложенные и сложные синтаксические случаи.

Созданная тестовая программа на C++ позволила строить синтаксическое дерево по полученному файлу на целевом языке и выводить его в удобочитаемом виде. Проведённые испытания подтвердили корректность работы как самого анализатора, так и алгоритма визуализации дерева разбора.

Таким образом, цель лабораторной работы достигнута. Создан работоспособный модуль синтаксического анализа, формирующий корректное синтаксическое дерево по исходному коду заданного языка и обеспечивающий возможность дальнейшего анализа структуры программы.

5. Исходные файлы

Исходные файлы проекта: репозиторий [Электронный ресурс]. — GitHub. — Режим доступа: https://github.com/Vsevomes/ProgLang_lab1 (дата обращения: 03.11.2025).