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

Отчет по лабораторной работе №1

по курсу «Языки программирования»

Выполнил студент группы P4119	_____	М.С. Маршалов
	(подпись)	

Руководитель	_____	Ю. Д. Кореньков
	(подпись)	

Санкт-Петербург

2025

Цель

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.

Задачи

Порядок выполнения:

1. Изучить выбранное средство синтаксического анализа

1.2 Средство должно поддерживать программный интерфейс, совместимый с используемым ЯП

1.3 Средство должно параметризоваться спецификацией, описывающей синтаксическую структуру разбираемого языка

1.4 Средство может функционировать посредством кодогенерации и/или подключения необходимых для его работы дополнительных библиотек

1.5 Средство может быть реализовано с нуля, в этом случае оно должно использовать обобщённый алгоритм, управляемый спецификацией

2. Изучить синтаксис разбираемого по варианту языка и записать спецификацию для средства синтаксического анализа, включающую следующие конструкции:

2.1 Подпрограммы со списком аргументов и возвращаемым значением

2.2 Операции контроля потока управления – простые ветвления if-else и циклы или аналоги

2.3 В зависимости от варианта – определения переменных

2.4 Целочисленные, строковые и односимвольные литералы

2.5 Выражения численной, битовой и логической арифметики

2.6 Выражения над одномерными массивами

2.7 Выражения вызова функции

3. Реализовать модуль, использующий средство синтаксического анализа для разбора языка по варианту

3.1 Программный интерфейс модуля должен принимать строку с текстом и возвращать структуру,

3.2 описывающую соответствующее дерево разбора и коллекцию сообщений ошибке

3.3 Результат работы модуля – дерево разбора – должно содержать иерархическое

3.4 представление для всех синтаксических конструкций, включая выражения, логически

3.5 представляющие собой иерархически организованные данные, даже если на уровне средства

3.6 синтаксического анализа для их разбора было использовано линейное представление

4 Реализовать тестовую программу для демонстрации работоспособности созданного модуля

4.2 Через аргументы командной строки программа должна принимать имя входного файла для чтения и анализа, имя выходного файла записи для дерева, описывающего синтаксическую структуру разобранного текста

4.3 Сообщения об ошибке должны выводиться тестовой программной (не модулем, отвечающим за анализ!) в стандартный поток вывода ошибок

Ход работы

При выполнении работы использовалась библиотека ANTLR4. Была реализована следующая грамматика, представленная в Листинге 1. Потребовалось переписать ее под ANTLR4, результат представлен в Листинге 2. С помощью написанной программы был получен результат разбора грамматики на Рисунке 1.

```
int main() {
    printf("Hello, World!\n");
    return 0;
}
================================================
=== Parse Tree ===
source
  sourceItem
    funcDef
      funcSignature
        typeRef
          builtinType
            INT: 'int'
        IDENTIFIER: 'main'
        LPAREN: '('
        argDefList
        RPAREN: ')'
      block
        LBRACE: '{'
        statement
          expressionStatement
            expr
              expr
                IDENTIFIER: 'printf'
              LPAREN: '('
              exprList
                expr
                  literal
                    STRING: '"Hello, World!\n"'
              RPAREN: ')'
            SEMI: ';'
        statement
          returnStatement
            RETURN: 'return'
            expr
              literal
                DEC: '0'
            SEMI: ';'
        RBRACE: '}'
```

Рисунок 1 – Результат разбора

Листинг 1 – Исходная грамматика

```
source: sourceItem*;
typeRef: {
|builtin: 'bool'|'byte'|'int'|'uint'|'long'|'ulong'|'char'|'string';
|custom: identifier;
|array: typeRef '[' (',')* ']';
};
funcSignature: typeRef? identifier '(' list<argDef> ')' {
argDef: typeRef? identifier;
};
sourceItem: {
|funcDef: funcSignature (statement.block|';');
};
statement: {
|var: typeRef list<identifier ('=' expr)?> ';'; // for static typing
|if: 'if' '(' expr ')' statement ('else' statement)?;
|block: '{' statement* '}';
|while: 'while' '(' expr ')' statement;
|do: 'do' block 'while' '(' 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 ')';

|call: expr '(' list<expr> ')';

|indexer: expr '[' list<expr> ']';

|place: identifier;

|literal: bool|str|char|hex|bits|dec;

};

Листинг 2 – Грамматика для ANTLR4

```
grammar Language;



source: sourceItem* EOF;


typeRef
    : builtinType # BuiltinTypeRef
    | IDENTIFIER  # CustomTypeRef
    | typeRef LBRACK (COMMA)* RBRACK # ArrayTypeRef
    ;

builtinType: BOOL | BYTE | INT | UINT | LONG | ULONG | CHAR | STRING_TYPE;


funcSignature: typeRef? IDENTIFIER LPAREN argDefList RPAREN;
argDefList: (argDef (COMMA argDef)*)?;
argDef: typeRef? IDENTIFIER;


sourceItem: funcDef;
funcDef: funcSignature (block | SEMI);


statement
    : varStatement
    | ifStatement
```

```
    | block
    | whileStatement
    | doStatement
    | breakStatement
    | returnStatement
    | expressionStatement
    | emptyStatement
    ;

varStatement: typeRef identifierList SEMI;
identifierList: IDENTIFIER (ASSIGN expr)? (COMMA IDENTIFIER (ASSIGN expr)?)*;

ifStatement: IF LPAREN expr RPAREN statement (ELSE statement)?;
block: LBRACE statement* RBRACE;
whileStatement: WHILE LPAREN expr RPAREN statement;
doStatement: DO block WHILE LPAREN expr RPAREN SEMI;
breakStatement: BREAK SEMI;
returnStatement: RETURN expr? SEMI;  // Добавлено правило для return
expressionStatement: expr SEMI;
emptyStatement: SEMI;

expr
    : expr binOp expr               # binaryExpr
```

```
    | unOp expr                    # unaryExpr
    | LPAREN expr RPAREN             # bracesExpr
    | expr LPAREN exprList RPAREN    # callExpr
    | expr LBRACK exprList RBRACK    # indexerExpr
    | IDENTIFIER                     # placeExpr
    | literal                    # literalExpr
    ;

exprList: (expr (COMMA expr)*)?;

literal
    : BOOL_LITERAL
    | STRING
    | CHAR_LITERAL
    | HEX
    | BITS
    | DEC
    ;

binOp: MUL | MINUS | PLUS | DIV | ASSIGN | EQ | NEQ | LT | GT | LTE | GTE | AND | OR;
unOp: MINUS | NOT | TILDE;

// Lexer rules
```

```
// Keywords
BOOL: 'bool';
BYTE: 'byte';
INT: 'int';
UINT: 'uint';
LONG: 'long';
ULONG: 'ulong';
CHAR: 'char';
STRING_TYPE: 'string';
IF: 'if';
ELSE: 'else';
WHILE: 'while';
DO: 'do';
BREAK: 'break';
RETURN: 'return';  // Добавлен return

// Literals
BOOL_LITERAL: 'true' | 'false';
IDENTIFIER: [a-zA-Z_][a-zA-Z_0-9]*;
STRING: '"' (~["\\] | '\\' .)* '"';
CHAR_LITERAL: '\'' .? '\'';
HEX: '0' [xX] [0-9A-Fa-f]+;
BITS: '0' [bB] [01]+;
```

```
DEC: [0-9]+;

// Operators
PLUS: '+';
MINUS: '-';
MUL: '*';
DIV: '/';
ASSIGN: '=';
EQ: '==';
NEQ: '!=';
LT: '<';
GT: '>';
LTE: '<=';
GTE: '>=';
AND: '&&';
OR: '||';
NOT: '!';
TILDE: '~';

// Punctuation
LPAREN: '(';
RPAREN: ')';
LBRACE: '{';
RBRACE: '}';
```

```
LBRACK: '[';

RBRACK: ']';

SEMI: ';';

COMMA: ',';


// Whitespace and comments

WS: [ \t\r\n]+ -> skip;

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

MULTILINE_COMMENT: '/*' .*? '*/' -> skip;
```