Министерство образования и науки РФ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

Лабораторная работа 1

вариант 3

по дисциплине

«Системное программное обеспечение»

Выполнил:

Кустарев Иван Палович 309681

Преподаватель:

Кореньков Юрий Дмитриевич

Санкт-Петербург, 2025

Задачи:

Реализовать построение графа потока управления посредством анализа дерева разбора для набора входных файлов. Выполнить анализ собранной информации и сформировать набор файлов с графическим представлением для результатов анализа.

Цели:

Порядок выполнения:

1. Изучить выбранное средство синтаксического анализа
    a. Средство должно поддерживать программный интерфейс, совместимый с языком Си
    b. Средство должно параметризоваться спецификацией, описывающей синтаксическую структуру разбираемого языка
    c. Средство может функционировать посредством кодогенерации и/или подключения необходимых для его работы дополнительных библиотек
    d. 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:
    a. Подпрограммы со списком аргументов и возвращаемым значением
    b. Операции контроля потока управления – простые ветвления if-else и циклы или аналоги
    c. В зависимости от варианта – определения переменных
    d. Целочисленные, строковые и односимвольные литералы
    e. Выражения численной, битовой и логической арифметики
    f. Выражения над одномерными массивами
    g. Выражения вызова функции
3. Реализовать модуль, использующий средство синтаксического анализа для разбора языка по варианту
    a. Программный интерфейс модуля должен принимать строку с текстом и возвращать структуру, описывающую соответствующее дерево разбора и коллекцию сообщений ошибке
    b. Результат работы модуля – дерево разбора – должно содержать иерархическое представление для всех синтаксических конструкций, включая выражения, логически представляющие собой иерархически организованные данные, даже если на уровне средства синтаксического анализа для их разбора было использовано линейное представление
4. Реализовать тестовую программу для демонстрации работоспособности созданного модуля
    a. 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
    b. Сообщения об ошибке должны выводиться тестовой программной (не модулем, отвечающим за анализ!) в стандартный поток вывода ошибок
5. Результаты тестирования представить в виде отчета, в который включить:
    a. В части 3 привести описание структур данных, представляющих результат разбора текста (3а)
    b. В части 4 описать, какая дополнительная обработка потребовалась для результата разбора, предоставляемого средством синтаксического анализа, чтобы сформировать результат работы созданного модуля
    c. В части 5 привести примеры исходных анализируемых текстов для всех синтаксических конструкций разбираемого языка и соответствующие результаты разбора

Выполнение:

Файл конфигураций flex:

```
%{
#include "error.h"
#include "parser.tab.h"
#include <string.h>
%}

%option yylineno

%%

[0-9]+ {
   yylval.node = createNode("DEC", NULL, yytext);
   return DEC;
}
\"[^\"\\]*(?:\\.[^\"\\]*)*\" {
   char buffer[1024];
   strncpy(buffer, yytext+1, strlen(yytext)-2);
   buffer[strlen(buffer)-1] = 0;
   yylval.node = createNode("STR", NULL, buffer);
   return STR;
}
'[^']' {
   char buffer[1];
   strncpy(buffer, yytext+1, 1);
   yylval.node = createNode("CHAR", NULL, buffer);
   return CHAR;
}

0[xX][0-9A-Fa-f]+ {
   char* buffer = malloc(256);
   sprintf(buffer, "%ld", strtol(yytext + 2, NULL, 16));
   yylval.node = createNode("HEX", NULL, buffer);
   return HEX;
}
0[bB][01]+ {
   char* buffer = malloc(256);
   sprintf(buffer, "%ld", strtol(yytext + 2, NULL, 2));
   yylval.node = createNode("BIN", NULL, buffer);
   return BIN;
}
"byte" {
         yylval.node = createNode("TYPEDEF", NULL, yytext);
         return TYPEDEF;
      }
"int" {
        yylval.node = createNode("TYPEDEF", NULL, yytext);
        return TYPEDEF;
     }
"uint" {
         yylval.node = createNode("TYPEDEF", NULL, yytext);
         return TYPEDEF;
```

```
    }
"long" {
        yylval.node = createNode("TYPEDEF", NULL, yytext);
        return TYPEDEF;
    }
"bool" {
        yylval.node = createNode("TYPEDEF", NULL, yytext);
        return TYPEDEF;
    }
"ulong" {
         yylval.node = createNode("TYPEDEF", NULL, yytext);
         return TYPEDEF;
     }
"char" {
        yylval.node = createNode("TYPEDEF", NULL, yytext);
        return TYPEDEF;
    }
"string"  {
          yylval.node = createNode("TYPEDEF", NULL, yytext);
          return TYPEDEF;
       }
"dim"                          {
 yylval.node = createNode("DIM", NULL, "");
 return DIM;
 }
"if"                          {
 yylval.node = createNode("IF", NULL, "");
 return IF;
 }
"as"                          {
 yylval.node = createNode("AS", NULL, "");
 return AS;
 }
"function"                          {
 yylval.node = createNode("FUNCTION", NULL, "");
 return FUNCTION;
 }
"end"                          {
 yylval.node = createNode("END", NULL, "");
 return END;
 }
"else"                          {
 yylval.node = createNode("ELSE", NULL, "");
 return ELSE;
 }
"then"                          {
 yylval.node = createNode("THEN", NULL, "");
 return THEN;
 }
"do"                          {
 yylval.node = createNode("DO", NULL, "");
 return DO;
 }
"loop"                          {
 yylval.node = createNode("LOOP", NULL, "");
 return LOOP;
```

```
 }
"while"                         {
 yylval.node = createNode("WHILE", NULL, "");
 return WHILE;
 }
"wend"                          {
 yylval.node = createNode("WEND", NULL, "");
 return WEND;
 }
"until"                         {
 yylval.node = createNode("UNTIL", NULL, "");
 return UNTIL;
 }
"break"                         {
 yylval.node = createNode("BREAK", NULL, "");
 return BREAK;
 }
"true"                          {
 yylval.node = createNode("TRUE", NULL, "");
 return TRUE;
 }
"false"                         {
 yylval.node = createNode("FALSE", NULL, "");
 return FALSE;
 }
"%"                             {
 return PERCENT;
 }
";"                            {
 return SEMICOLON;
 }
","                            {
 yylval.node = createNode("COMMA", NULL, "");
 return COMMA;
 }
"!="                            {
 return NOTEQUAL;
 }
"=="                            {
 return EQUALITY;
 }
"!"                            {
 return NOT;
 }
"\|\|"                          {
 return OR;
 }
"&&"                            {
 return AND;
 }
"+"                            {
 return PLUS;
 }
"-"                            {
 return MINUS;
 }
```

```
"++"                    {
 return INCREMENT;
 }
"--"                    {
 return DECREMENT;
 }
"<="                    {
 return LESSTHANEQ;
 }
">="                    {
 return GREATERTHANEQ;
 }
"<"                     {
 return LESSTHAN;
 }
">"                     {
 return GREATERTHAN;
 }
"*"                     {
 return MUL;
 }
"("                     {
 return LPAREN;
 }
")"                     {
 return RPAREN;
 }
"="                     {
 return SET;
 }
[a-zA-Z_][a-zA-Z_0-9]* {
   yylval.node = createNode("IDENTIFIER", NULL, yytext);
   return IDENTIFIER;
}
\/\/.*                    { /* Пропустить комментарии */ }
\n                      { /* Пропустить переводы строк */ }
[ \t]                   { /* Пропустить пробелы и табуляцию */ }
.                       {
                          // Обработка нераспознанных символов
                          fprintf(stderr, "Нераспознанный символ: %s\n", yytext);
                        }
%%


int yywrap() {
   return 1;
}
```

Для парсинга использовался bison. Конфигурации:

```
%{
#include "parser.tab.h"
#include "error.h"

#define YYDEBUG 1
%}
%define parse.error verbose
%locations

%union {
   TreeNode* node;
}

%token <node> PLUS MINUS MUL SLASH PERCENT SET NOTEQUAL EQUALITY
%token <node> LESSTHAN GREATERTHAN LESSTHANEQ GREATERTHANEQ
%token <node> AND OR NOT
%token <node> DECREMENT INCREMENT
%token <node> FUNCTION
%token <node> AS
%token <node> ARRAY
%token <node> DEF END BEGIN_BLOCK
%token <node> WEND
%token <node> IDENTIFIER
%token <node> STR
%token <node> COMMA
%token <node> CHAR
%token <node> BIN HEX DEC
%token <node> TRUE FALSE
%token <node> IF ELSE WHILE UNTIL DO BREAK
%token <node> THEN
%token <node> LOOP
%token <node> SEMICOLON
%token <node> LPAREN RPAREN
%token <node> TYPEDEF
%token <node> DIM

%left SET

%left AND OR

%left EQUALITY NOTEQUAL

%left LESSTHAN GREATERTHAN LESSTHANEQ GREATERTHANEQ

%left PLUS MINUS

%left MUL SLASH PERCENT

%left INCREMENT DECREMENT

%type <node> typeRef
%type <node> funcSignature
```

```
%type <node> argDef
%type <node> sourceItem
%type <node> listSourceItem
%type <node> statement
%type <node> var
%type <node> if
%type <node> break
%type <node> builtin
%type <node> custom
%type <node> array
%type <node> source
%type <node> listArgDef
%type <node> optionalTypeRef
%type <node> literal
%type <node> place
%type <node> expr
%type <node> listExpr
%type <node> callOrIndexer
%type <node> braces
%type <node> unary
%type <node> binary
%type <node> listVar
%type <node> else
%type <node> listStatement
%type <node> while
%type <node> do
%type <node> whileOrUntil
%type <node> arrayCommas
%type <node> funcDef

%%
/* SourceItem */

source: listSourceItem      {{TreeNode* elements[] = {$1};$$ = createNode("source",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}};

sourceItem: funcDef        {{TreeNode* elements[] = {$1};$$ = createNode("sourceItem",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}};

listSourceItem:            {{$$ = NULL;}}
   | sourceItem listSourceItem     {{TreeNode* elements[] = {$1, $2}; $$ = createNode("listSourceItem",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}};



/* FuncSignature */

funcDef: FUNCTION funcSignature listStatement END FUNCTION {{TreeNode* elements[] = {$2, $3}; $$ =
createNode("funcDef", mallocChildNodes(*(&elements + 1) - elements, elements), "");}};

funcSignature: IDENTIFIER LPAREN listArgDef RPAREN optionalTypeRef {{TreeNode* elements[] = {$3, $5};$$
= createNode("funcSignature", mallocChildNodes(*(&elements + 1) - elements, elements), $1->value);}};

listArgDef:                {{$$ = NULL;}}
   | argDef listArgDef     {{TreeNode* elements[] = {$1, $2};$$ = createNode("listArgDef",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}; //чтобы не было listArgDef с двумя
```

```
argDef
   | argDef COMMA listArgDef      {{TreeNode* elements[] = {$1, $3};$$ = createNode("listArgDef",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}};

argDef: IDENTIFIER optionalTypeRef  {{TreeNode* elements[] = {$1, $2};$$ = createNode("argDef",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}};

optionalTypeRef:          {{ $$ = NULL; }}
   | AS typeRef          {{$$ = $2;}};



/* TypeRef */

typeRef: builtin          {{$$ = $1;}}
   | custom             {{$$ = $1;}}
   | array              {{$$ = $1;}};

builtin: TYPEDEF          {{$$ = $1;}};



/* Statement */

statement: var            {{$$ =  $1;}}
   | if                {{$$ =  $1;}}
   | while              {{$$ =  $1;}}
   | do                {{$$ =  $1;}}
   | break              {{$$ =  $1;}}
   | expr SEMICOLON       {{$$ = $1;}};

listStatement:            {{$$ = NULL;}}
   | statement listStatement   {{TreeNode* elements[] = {$1, $2};$$ = createNode("listStatement",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}};



custom: IDENTIFIER {{$$ = $1;}};

array: typeRef LPAREN arrayCommas RPAREN {{TreeNode* elements[] = {$1, $3};$$ = createNode("array",
mallocChildNodes(*(&elements + 1) - elements, elements), $2->value);}};

arrayCommas:              {{$$ = NULL;}}
   | COMMA arrayCommas     {{TreeNode* elements[] = {$1, $2};$$ = createNode("arrayCommas",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}};



/* IF ELSE */

if: IF expr THEN listStatement else END IF {{TreeNode* elements[] = {$2, $4, $5};$$ = createNode("if",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}};

else: ELSE listStatement    {{TreeNode* elements[] = {$2};$$ = createNode("else",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   |                    {{$$ = NULL;}};
```

```
while: WHILE expr listStatement WEND {{TreeNode* elements[] = {$2, $3};$$ = createNode("while",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}

do: DO listStatement LOOP whileOrUntil expr {{TreeNode* elements[] = {$2, $4, $5};$$ = createNode("do",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}

whileOrUntil: WHILE        {{$$ = $1;}}
   | UNTIL                 {{$$ = $1;}}

break: BREAK {{;$$ = createNode("break", NULL, "");}};

expr: unary                {{$$ = $1;}}
   | binary                {{$$ = $1;}}
   | braces                {{$$ = $1;}}
   | callOrIndexer         {{$$ = $1;}}
   | place                 {{$$ = $1;}}
   | literal               {{$$ = $1;}};

binary: expr SET expr     {{TreeNode* elements[] = {$1, $3};$$ = createNode("SET",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}};
   | expr PLUS expr        {{TreeNode* elements[] = {$1, $3};$$ = createNode("PLUS",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   | expr MINUS expr       {{TreeNode* elements[] = {$1, $3};$$ = createNode("MINUS",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   | expr MUL expr         {{TreeNode* elements[] = {$1, $3};$$ = createNode("MUL",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   | expr SLASH expr       {{TreeNode* elements[] = {$1, $3};$$ = createNode("SLASH",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   | expr PERCENT expr     {{TreeNode* elements[] = {$1, $3};$$ = createNode("PERCENT",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   | expr EQUALITY expr {{TreeNode* elements[] = {$1, $3};$$ = createNode("EQUALITY",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   | expr NOTEQUAL expr    {{TreeNode* elements[] = {$1, $3};$$ = createNode("NOTEQUAL",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   | expr LESSTHAN expr    {{TreeNode* elements[] = {$1, $3};$$ = createNode("LESSTHAN",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   | expr GREATERTHAN expr {{TreeNode* elements[] = {$1, $3};$$ = createNode("GREATERTHAN",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   | expr LESSTHANEQ expr  {{TreeNode* elements[] = {$1, $3};$$ = createNode("LESSTHANEQ",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   | expr GREATERTHANEQ expr   {{TreeNode* elements[] = {$1, $3};$$ = createNode("GREATERTHANEQ",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   | expr AND expr         {{TreeNode* elements[] = {$1, $3};$$ = createNode("AND",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   | expr OR expr          {{TreeNode* elements[] = {$1, $3};$$ = createNode("OR",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}};

unary: INCREMENT expr           {{TreeNode* elements[] = {$2};$$ = createNode("INCREMENT",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   | DECREMENT expr          {{TreeNode* elements[] = {$2};$$ = createNode("DECREMENT",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   | NOT expr            {{TreeNode* elements[] = {$2};$$ = createNode("NOT", mallocChildNodes(*(&elements
+ 1) - elements, elements), "");}};

braces: LPAREN expr RPAREN  {{TreeNode* elements[] = {$2};$$ = createNode("braces",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}};
```

```
callOrIndexer: expr LPAREN listExpr RPAREN  {{TreeNode* elements[] = {$1, $3};$$ =
createNode("callOrIndexer", mallocChildNodes(*(&elements + 1) - elements, elements), "");}};

listExpr:                {{$$ = NULL;}}
   | expr listExpr       {{TreeNode* elements[] = {$1, $2};$$ = createNode("listExpr",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
   | expr COMMA listExpr   {{TreeNode* elements[] = {$1, $3};$$ = createNode("listExpr",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}

place: IDENTIFIER         {{$$ = $1;}};

literal: TRUE             {{$$ = $1;}}
   | FALSE                {{$$ = $1;}}
   | STR                  {{$$ = $1;}}
   | CHAR                 {{$$ = $1;}}
   | HEX                  {{$$ = $1;}}
   | BIN                  {{$$ = $1;}}
   | DEC                  {{$$ = $1;}};



/* VAR */

listVar:  {{$$ = NULL;}}
   | IDENTIFIER listVar {{TreeNode* elements[] = {$1, $2};$$ = createNode("listVar",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}; //чтобы не было listVar с двумя
identifier
   | IDENTIFIER COMMA listVar {{TreeNode* elements[] = {$1, $3};$$ = createNode("listVar",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}};

var: DIM listVar AS typeRef {{TreeNode* elements[] = {$2, $4};$$ = createNode("var",
mallocChildNodes(*(&elements + 1) - elements, elements), "");}}
%%
```

Для запуска используется Makefile:

```
SOURCE_PATH=source

generate-resources: $(SOURCE_PATH)/lexems.l $(SOURCE_PATH)/parser.y
	flex $(SOURCE_PATH)/lexems.l
	bison -d -t $(SOURCE_PATH)/parser.y
	echo '#include "node.h"' | cat - parser.tab.h > temp && mv temp parser.tab.h
```

Файл заголовков, содержащий структуры использованные для создания дерева операций:

```
#ifndef SPO_LAB1_NODE_H
#define SPO_LAB1_NODE_H

#include "parser.h"
```

```
typedef struct TreeNode TreeNode;
typedef struct ChildNodes ChildNodes;

extern TreeNode **allNodes;
extern int allNodesCount;

struct ChildNodes {
   int size;
   TreeNode **childNodes;
};

struct TreeNode {
   char *type;
   TreeNode **childNodes;
   int childrenNumber;
   char *value;
   int id;
};

void printTree(TreeNode **allNodes, int allNodesCount, FILE *output_file);

TreeNode *createNode(char *type, ChildNodes *childNodes, char *value);

ChildNodes *mallocChildNodes(int size, TreeNode **nodesArg);

#endif //SPO_LAB1_NODE_H
```

Файл заголовков содержащий структуры, использованные для сохранения результатов разбора:

```
#ifndef SPO_LAB1_PARSER_H
#define SPO_LAB1_PARSER_H

#include <stdlib.h>
#include <stdio.h>
#include "node.h"

typedef struct ChildNodes ChildNodes;
typedef struct TreeNode TreeNode;
typedef struct ChildNodes ChildNodes;
typedef struct ParseResult ParseResult;

extern TreeNode **allNodes;
extern int allNodesCount;
extern char** errors;
extern int errorsCount;

extern int yyparse();

extern FILE *yyin;

struct ParseResult {
```

```
    int size;
    TreeNode **nodes;
    char** errors;
    int errorsCount;
};

ParseResult* parse(FILE *file);
void freeMem(ParseResult *parseResult);

#endif //SPO_LAB1_PARSER_H
```

При запуске программы необходимо передать два аргумента - файл с данными и файл для вывода.

Результаты:

В результате работы модуля в файл заполняется структура, в формате flowchart, представляющая структуру дерева разбора. Пример выходного файла:

```
flowchart TB
node498([Type: source]) --> node497([Type: listSourceItem])
node497([Type: listSourceItem]) --> node23([Type: sourceItem])
node23([Type: sourceItem]) --> node22([Type: funcDef])
node22([Type: funcDef]) --> node16([Type: funcSignature, Value: sum])
node16([Type: funcSignature, Value: sum]) --> node12([Type: listArgDef])
node12([Type: listArgDef]) --> node6([Type: argDef])
node6([Type: argDef]) --> node2([Type: IDENTIFIER, Value: arg1])
node6([Type: argDef]) --> node4([Type: TYPEDEF, Value: int])
node12([Type: listArgDef]) --> node11([Type: listArgDef])
node11([Type: listArgDef]) --> node10([Type: argDef])
node10([Type: argDef]) --> node7([Type: IDENTIFIER, Value: arg2])
node10([Type: argDef]) --> node9([Type: TYPEDEF, Value: int])
node16([Type: funcSignature, Value: sum]) --> node14([Type: TYPEDEF, Value: int])
node22([Type: funcDef]) --> node20([Type: listStatement])
node20([Type: listStatement]) --> node18([Type: SUM])
node18([Type: SUM]) --> node15([Type: IDENTIFIER, Value: arg1])
node18([Type: SUM]) --> node17([Type: IDENTIFIER, Value: arg2])
node497([Type: listSourceItem]) --> node496([Type: listSourceItem])
node496([Type: listSourceItem]) --> node47([Type: sourceItem])
node47([Type: sourceItem]) --> node46([Type: funcDef])
node46([Type: funcDef]) --> node40([Type: funcSignature, Value: sub])
node40([Type: funcSignature, Value: sub]) --> node36([Type: listArgDef])
node36([Type: listArgDef]) --> node30([Type: argDef])
node30([Type: argDef]) --> node26([Type: IDENTIFIER, Value: arg1])
node30([Type: argDef]) --> node28([Type: TYPEDEF, Value: int])
node36([Type: listArgDef]) --> node35([Type: listArgDef])
node35([Type: listArgDef]) --> node34([Type: argDef])
node34([Type: argDef]) --> node31([Type: IDENTIFIER, Value: arg2])
```

node34([Type: argDef]) --> node33([Type: TYPEDEF, Value: int])
node40([Type: funcSignature, Value: sub]) --> node38([Type: TYPEDEF, Value: int])
node46([Type: funcDef]) --> node44([Type: listStatement])
node44([Type: listStatement]) --> node42([Type: SUB])
node42([Type: SUB]) --> node39([Type: IDENTIFIER, Value: arg1])
node42([Type: SUB]) --> node41([Type: IDENTIFIER, Value: arg2])
node496([Type: listSourceItem]) --> node495([Type: listSourceItem])
node495([Type: listSourceItem]) --> node71([Type: sourceItem])
node71([Type: sourceItem]) --> node70([Type: funcDef])
node70([Type: funcDef]) --> node64([Type: funcSignature, Value: mul])
node64([Type: funcSignature, Value: mul]) --> node60([Type: listArgDef])
node60([Type: listArgDef]) --> node54([Type: argDef])
node54([Type: argDef]) --> node50([Type: IDENTIFIER, Value: arg1])
node54([Type: argDef]) --> node52([Type: TYPEDEF, Value: int])
node60([Type: listArgDef]) --> node59([Type: listArgDef])
node59([Type: listArgDef]) --> node58([Type: argDef])
node58([Type: argDef]) --> node55([Type: IDENTIFIER, Value: arg2])
node58([Type: argDef]) --> node57([Type: TYPEDEF, Value: int])
node64([Type: funcSignature, Value: mul]) --> node62([Type: TYPEDEF, Value: int])
node70([Type: funcDef]) --> node68([Type: listStatement])
node68([Type: listStatement]) --> node66([Type: MUL])
node66([Type: MUL]) --> node63([Type: IDENTIFIER, Value: arg1])
node66([Type: MUL]) --> node65([Type: IDENTIFIER, Value: arg2])
node495([Type: listSourceItem]) --> node494([Type: listSourceItem])
node494([Type: listSourceItem]) --> node95([Type: sourceItem])
node95([Type: sourceItem]) --> node94([Type: funcDef])
node94([Type: funcDef]) --> node88([Type: funcSignature, Value: div])
node88([Type: funcSignature, Value: div]) --> node84([Type: listArgDef])
node84([Type: listArgDef]) --> node78([Type: argDef])
node78([Type: argDef]) --> node74([Type: IDENTIFIER, Value: arg1])
node78([Type: argDef]) --> node76([Type: TYPEDEF, Value: int])
node84([Type: listArgDef]) --> node83([Type: listArgDef])
node83([Type: listArgDef]) --> node82([Type: argDef])
node82([Type: argDef]) --> node79([Type: IDENTIFIER, Value: arg2])
node82([Type: argDef]) --> node81([Type: TYPEDEF, Value: int])
node88([Type: funcSignature, Value: div]) --> node86([Type: TYPEDEF, Value: int])

node94([Type: funcDef]) --> node92([Type: listStatement])
node92([Type: listStatement]) --> node90([Type: DIV])
node90([Type: DIV]) --> node87([Type: IDENTIFIER, Value: arg1])
node90([Type: DIV]) --> node89([Type: IDENTIFIER, Value: arg2])
node494([Type: listSourceItem]) --> node493([Type: listSourceItem])
node493([Type: listSourceItem]) --> node177([Type: sourceItem])
node177([Type: sourceItem]) --> node176([Type: funcDef])
node176([Type: funcDef]) --> node101([Type: funcSignature, Value: read_num])
node101([Type: funcSignature, Value: read_num]) --> node99([Type: TYPEDEF, Value: int])
node176([Type: funcDef]) --> node174([Type: listStatement])
node174([Type: listStatement]) --> node107([Type: var])
node107([Type: var]) --> node104([Type: listVar])
node104([Type: listVar]) --> node102([Type: IDENTIFIER, Value: in])
node107([Type: var]) --> node105([Type: TYPEDEF, Value: int])
node174([Type: listStatement]) --> node173([Type: listStatement])
node173([Type: listStatement]) --> node113([Type: var])
node113([Type: var]) --> node110([Type: listVar])
node110([Type: listVar]) --> node108([Type: IDENTIFIER, Value: res])
node113([Type: var]) --> node111([Type: TYPEDEF, Value: int])

node173([Type: listStatement]) --> node172([Type: listStatement])
node172([Type: listStatement]) --> node119([Type: var])
node119([Type: var]) --> node116([Type: listVar])
node116([Type: listVar]) --> node114([Type: IDENTIFIER, Value: i])
node119([Type: var]) --> node117([Type: TYPEDEF, Value: int])
node172([Type: listStatement]) --> node171([Type: listStatement])
node171([Type: listStatement]) --> node121([Type: SET])
node121([Type: SET]) --> node118([Type: IDENTIFIER, Value: res])
node121([Type: SET]) --> node120([Type: DEC, Value: 0])
node171([Type: listStatement]) --> node170([Type: listStatement])
node170([Type: listStatement]) --> node124([Type: SET])
node124([Type: SET]) --> node122([Type: IDENTIFIER, Value: i])
node124([Type: SET]) --> node123([Type: DEC, Value: 1])
node170([Type: listStatement]) --> node169([Type: listStatement])
node169([Type: listStatement]) --> node165([Type: while])
node165([Type: while]) --> node126([Type: BOOL, Value: true])
node165([Type: while]) --> node164([Type: listStatement])
node164([Type: listStatement]) --> node130([Type: SET])
node130([Type: SET]) --> node127([Type: IDENTIFIER, Value: in])
node130([Type: SET]) --> node129([Type: callOrIndexer])
node129([Type: callOrIndexer]) --> node128([Type: IDENTIFIER, Value: stdin])
node164([Type: listStatement]) --> node163([Type: listStatement])
node163([Type: listStatement]) --> node144([Type: if])
node144([Type: if]) --> node135([Type: EQUALITY])
node135([Type: EQUALITY]) --> node132([Type: IDENTIFIER, Value: in])
node135([Type: EQUALITY]) --> node133([Type: DEC, Value: 13])
node144([Type: if]) --> node142([Type: listStatement])
node142([Type: listStatement]) --> node137([Type: callOrIndexer])
node137([Type: callOrIndexer]) --> node136([Type: IDENTIFIER, Value: stdin])
node142([Type: listStatement]) --> node141([Type: listStatement])
node141([Type: listStatement]) --> node139([Type: break])
node163([Type: listStatement]) --> node162([Type: listStatement])
node162([Type: listStatement]) --> node154([Type: SET])
node154([Type: SET]) --> node145([Type: IDENTIFIER, Value: res])
node154([Type: SET]) --> node153([Type: SUM])
node153([Type: SUM]) --> node148([Type: MUL])
node148([Type: MUL]) --> node146([Type: IDENTIFIER, Value: res])
node148([Type: MUL]) --> node147([Type: DEC, Value: 10])
node153([Type: SUM]) --> node152([Type: braces])
node152([Type: braces]) --> node151([Type: SUB])
node151([Type: SUB]) --> node149([Type: IDENTIFIER, Value: in])
node151([Type: SUB]) --> node150([Type: DEC, Value: 48])
node162([Type: listStatement]) --> node161([Type: listStatement])
node161([Type: listStatement]) --> node159([Type: SET])
node159([Type: SET]) --> node155([Type: IDENTIFIER, Value: i])
node159([Type: SET]) --> node158([Type: SUM])
node158([Type: SUM]) --> node156([Type: IDENTIFIER, Value: i])
node158([Type: SUM]) --> node157([Type: DEC, Value: 1])

node169([Type: listStatement]) --> node168([Type: listStatement])
node168([Type: listStatement]) --> node166([Type: IDENTIFIER, Value: res])
node493([Type: listSourceItem]) --> node492([Type: listSourceItem])
node492([Type: listSourceItem]) --> node236([Type: sourceItem])
node236([Type: sourceItem]) --> node235([Type: funcDef])
node235([Type: funcDef]) --> node188([Type: funcSignature, Value: number_length])
node188([Type: funcSignature, Value: number_length]) --> node184([Type: listArgDef])

node184([Type: listArgDef]) --> node183([Type: argDef])
node183([Type: argDef]) --> node180([Type: IDENTIFIER, Value: num])
node183([Type: argDef]) --> node182([Type: TYPEDEF, Value: int])
node188([Type: funcSignature, Value: number_length]) --> node186([Type: TYPEDEF, Value: int])
node235([Type: funcDef]) --> node233([Type: listStatement])
node233([Type: listStatement]) --> node194([Type: var])
node194([Type: var]) --> node191([Type: listVar])
node191([Type: listVar]) --> node189([Type: IDENTIFIER, Value: rest])
node194([Type: var]) --> node192([Type: TYPEDEF, Value: int])
node233([Type: listStatement]) --> node232([Type: listStatement])
node232([Type: listStatement]) --> node200([Type: var])
node200([Type: var]) --> node197([Type: listVar])
node197([Type: listVar]) --> node195([Type: IDENTIFIER, Value: count])
node200([Type: var]) --> node198([Type: TYPEDEF, Value: int])
node232([Type: listStatement]) --> node231([Type: listStatement])
node231([Type: listStatement]) --> node202([Type: SET])
node202([Type: SET]) --> node199([Type: IDENTIFIER, Value: count])
node202([Type: SET]) --> node201([Type: DEC, Value: 1])
node231([Type: listStatement]) --> node230([Type: listStatement])
node230([Type: listStatement]) --> node207([Type: SET])
node207([Type: SET]) --> node203([Type: IDENTIFIER, Value: rest])
node207([Type: SET]) --> node206([Type: DIV])
node206([Type: DIV]) --> node204([Type: IDENTIFIER, Value: num])
node206([Type: DIV]) --> node205([Type: DEC, Value: 10])
node230([Type: listStatement]) --> node229([Type: listStatement])
node229([Type: listStatement]) --> node225([Type: while])
node225([Type: while]) --> node212([Type: NOTEQUAL])
node212([Type: NOTEQUAL]) --> node209([Type: IDENTIFIER, Value: rest])
node212([Type: NOTEQUAL]) --> node210([Type: DEC, Value: 0])
node225([Type: while]) --> node224([Type: listStatement])
node224([Type: listStatement]) --> node216([Type: SET])
node216([Type: SET]) --> node211([Type: IDENTIFIER, Value: rest])
node216([Type: SET]) --> node215([Type: DIV])
node215([Type: DIV]) --> node213([Type: IDENTIFIER, Value: rest])
node215([Type: DIV]) --> node214([Type: DEC, Value: 10])
node224([Type: listStatement]) --> node223([Type: listStatement])
node223([Type: listStatement]) --> node221([Type: SET])
node221([Type: SET]) --> node217([Type: IDENTIFIER, Value: count])
node221([Type: SET]) --> node220([Type: SUM])
node220([Type: SUM]) --> node218([Type: IDENTIFIER, Value: count])
node220([Type: SUM]) --> node219([Type: DEC, Value: 1])
node229([Type: listStatement]) --> node228([Type: listStatement])
node228([Type: listStatement]) --> node226([Type: IDENTIFIER, Value: count])
node492([Type: listSourceItem]) --> node491([Type: listSourceItem])
node491([Type: listSourceItem]) --> node347([Type: sourceItem])
node347([Type: sourceItem]) --> node346([Type: funcDef])
node346([Type: funcDef]) --> node247([Type: funcSignature, Value: print_num])
node247([Type: funcSignature, Value: print_num]) --> node243([Type: listArgDef])
node243([Type: listArgDef]) --> node242([Type: argDef])
node242([Type: argDef]) --> node239([Type: IDENTIFIER, Value: num])
node242([Type: argDef]) --> node241([Type: TYPEDEF, Value: int])
node247([Type: funcSignature, Value: print_num]) --> node245([Type: TYPEDEF, Value: int])
node346([Type: funcDef]) --> node344([Type: listStatement])
node344([Type: listStatement]) --> node253([Type: var])
node253([Type: var]) --> node250([Type: listVar])
node250([Type: listVar]) --> node248([Type: IDENTIFIER, Value: rest])

node253([Type: var]) --> node251([Type: TYPEDEF, Value: int])

node344([Type: listStatement]) --> node343([Type: listStatement])
node343([Type: listStatement]) --> node259([Type: var])
node259([Type: var]) --> node256([Type: listVar])
node256([Type: listVar]) --> node254([Type: IDENTIFIER, Value: count])
node259([Type: var]) --> node257([Type: TYPEDEF, Value: int])
node343([Type: listStatement]) --> node342([Type: listStatement])
node342([Type: listStatement]) --> node265([Type: var])
node265([Type: var]) --> node262([Type: listVar])
node262([Type: listVar]) --> node260([Type: IDENTIFIER, Value: shift])
node265([Type: var]) --> node263([Type: TYPEDEF, Value: int])
node342([Type: listStatement]) --> node341([Type: listStatement])
node341([Type: listStatement]) --> node267([Type: SET])
node267([Type: SET]) --> node264([Type: IDENTIFIER, Value: rest])
node267([Type: SET]) --> node266([Type: IDENTIFIER, Value: num])
node341([Type: listStatement]) --> node340([Type: listStatement])
node340([Type: listStatement]) --> node275([Type: SET])
node275([Type: SET]) --> node268([Type: IDENTIFIER, Value: count])
node275([Type: SET]) --> node274([Type: SUB])
node274([Type: SUB]) --> node272([Type: callOrIndexer])
node272([Type: callOrIndexer]) --> node269([Type: IDENTIFIER, Value: number_length])
node272([Type: callOrIndexer]) --> node271([Type: listExpr])
node271([Type: listExpr]) --> node270([Type: IDENTIFIER, Value: num])
node274([Type: SUB]) --> node273([Type: DEC, Value: 1])
node340([Type: listStatement]) --> node339([Type: listStatement])
node339([Type: listStatement]) --> node278([Type: SET])
node278([Type: SET]) --> node276([Type: IDENTIFIER, Value: shift])
node278([Type: SET]) --> node277([Type: DEC, Value: 1])
node339([Type: listStatement]) --> node338([Type: listStatement])
node338([Type: listStatement]) --> node296([Type: while])
node296([Type: while]) --> node283([Type: NOTEQUAL])
node283([Type: NOTEQUAL]) --> node280([Type: IDENTIFIER, Value: count])
node283([Type: NOTEQUAL]) --> node281([Type: DEC, Value: 0])
node296([Type: while]) --> node295([Type: listStatement])
node295([Type: listStatement]) --> node287([Type: SET])
node287([Type: SET]) --> node282([Type: IDENTIFIER, Value: shift])
node287([Type: SET]) --> node286([Type: MUL])
node286([Type: MUL]) --> node284([Type: IDENTIFIER, Value: shift])
node286([Type: MUL]) --> node285([Type: DEC, Value: 10])
node295([Type: listStatement]) --> node294([Type: listStatement])
node294([Type: listStatement]) --> node292([Type: SET])
node292([Type: SET]) --> node288([Type: IDENTIFIER, Value: count])
node292([Type: SET]) --> node291([Type: SUB])
node291([Type: SUB]) --> node289([Type: IDENTIFIER, Value: count])
node291([Type: SUB]) --> node290([Type: DEC, Value: 1])
node338([Type: listStatement]) --> node337([Type: listStatement])
node337([Type: listStatement]) --> node323([Type: while])
node323([Type: while]) --> node301([Type: NOTEQUAL])
node301([Type: NOTEQUAL]) --> node298([Type: IDENTIFIER, Value: shift])
node301([Type: NOTEQUAL]) --> node299([Type: DEC, Value: 0])
node323([Type: while]) --> node322([Type: listStatement])
node322([Type: listStatement]) --> node308([Type: callOrIndexer])
node308([Type: callOrIndexer]) --> node300([Type: IDENTIFIER, Value: stdout])
node308([Type: callOrIndexer]) --> node307([Type: listExpr])
node307([Type: listExpr]) --> node306([Type: SUM])

node306([Type: SUM]) --> node304([Type: DIV])
node304([Type: DIV]) --> node302([Type: IDENTIFIER, Value: rest])
node304([Type: DIV]) --> node303([Type: IDENTIFIER, Value: shift])
node306([Type: SUM]) --> node305([Type: DEC, Value: 48])
node322([Type: listStatement]) --> node321([Type: listStatement])
node321([Type: listStatement]) --> node313([Type: SET])
node313([Type: SET]) --> node309([Type: IDENTIFIER, Value: rest])
node313([Type: SET]) --> node312([Type: PERCENT])
node312([Type: PERCENT]) --> node310([Type: IDENTIFIER, Value: rest])
node312([Type: PERCENT]) --> node311([Type: IDENTIFIER, Value: shift])
node321([Type: listStatement]) --> node320([Type: listStatement])
node320([Type: listStatement]) --> node318([Type: SET])

node318([Type: SET]) --> node314([Type: IDENTIFIER, Value: shift])
node318([Type: SET]) --> node317([Type: DIV])
node317([Type: DIV]) --> node315([Type: IDENTIFIER, Value: shift])
node317([Type: DIV]) --> node316([Type: DEC, Value: 10])
node337([Type: listStatement]) --> node336([Type: listStatement])
node336([Type: listStatement]) --> node327([Type: callOrIndexer])
node327([Type: callOrIndexer]) --> node324([Type: IDENTIFIER, Value: stdout])
node327([Type: callOrIndexer]) --> node326([Type: listExpr])
node326([Type: listExpr]) --> node325([Type: DEC, Value: 10])
node336([Type: listStatement]) --> node335([Type: listStatement])
node335([Type: listStatement]) --> node331([Type: callOrIndexer])
node331([Type: callOrIndexer]) --> node328([Type: IDENTIFIER, Value: stdout])
node331([Type: callOrIndexer]) --> node330([Type: listExpr])
node330([Type: listExpr]) --> node329([Type: DEC, Value: 13])
node335([Type: listStatement]) --> node334([Type: listStatement])
node334([Type: listStatement]) --> node332([Type: DEC, Value: 0])
node491([Type: listSourceItem]) --> node490([Type: listSourceItem])
node490([Type: listSourceItem]) --> node489([Type: sourceItem])
node489([Type: sourceItem]) --> node488([Type: funcDef])
node488([Type: funcDef]) --> node351([Type: funcSignature, Value: main])
node488([Type: funcDef]) --> node486([Type: listStatement])
node486([Type: listStatement]) --> node357([Type: var])
node357([Type: var]) --> node354([Type: listVar])
node354([Type: listVar]) --> node352([Type: IDENTIFIER, Value: a])
node357([Type: var]) --> node355([Type: TYPEDEF, Value: int])
node486([Type: listStatement]) --> node485([Type: listStatement])
node485([Type: listStatement]) --> node363([Type: var])
node363([Type: var]) --> node360([Type: listVar])
node360([Type: listVar]) --> node358([Type: IDENTIFIER, Value: b])
node363([Type: var]) --> node361([Type: TYPEDEF, Value: int])
node485([Type: listStatement]) --> node484([Type: listStatement])
node484([Type: listStatement]) --> node369([Type: var])
node369([Type: var]) --> node366([Type: listVar])
node366([Type: listVar]) --> node364([Type: IDENTIFIER, Value: op])
node369([Type: var]) --> node367([Type: TYPEDEF, Value: int])
node484([Type: listStatement]) --> node483([Type: listStatement])
node483([Type: listStatement]) --> node375([Type: var])
node375([Type: var]) --> node372([Type: listVar])
node372([Type: listVar]) --> node370([Type: IDENTIFIER, Value: res])
node375([Type: var]) --> node373([Type: TYPEDEF, Value: int])
node483([Type: listStatement]) --> node482([Type: listStatement])
node482([Type: listStatement]) --> node480([Type: while])
node480([Type: while]) --> node376([Type: BOOL, Value: true])

node480([Type: while]) --> node479([Type: listStatement])
node479([Type: listStatement]) --> node380([Type: SET])
node380([Type: SET]) --> node377([Type: IDENTIFIER, Value: a])
node380([Type: SET]) --> node379([Type: callOrIndexer])
node379([Type: callOrIndexer]) --> node378([Type: IDENTIFIER, Value: read_num])
node479([Type: listStatement]) --> node478([Type: listStatement])
node478([Type: listStatement]) --> node384([Type: SET])
node384([Type: SET]) --> node381([Type: IDENTIFIER, Value: op])
node384([Type: SET]) --> node383([Type: callOrIndexer])
node383([Type: callOrIndexer]) --> node382([Type: IDENTIFIER, Value: stdin])
node478([Type: listStatement]) --> node477([Type: listStatement])
node477([Type: listStatement]) --> node386([Type: callOrIndexer])
node386([Type: callOrIndexer]) --> node385([Type: IDENTIFIER, Value: stdin])
node477([Type: listStatement]) --> node476([Type: listStatement])
node476([Type: listStatement]) --> node388([Type: callOrIndexer])
node388([Type: callOrIndexer]) --> node387([Type: IDENTIFIER, Value: stdin])
node476([Type: listStatement]) --> node475([Type: listStatement])
node475([Type: listStatement]) --> node392([Type: SET])
node392([Type: SET]) --> node389([Type: IDENTIFIER, Value: b])
node392([Type: SET]) --> node391([Type: callOrIndexer])
node391([Type: callOrIndexer]) --> node390([Type: IDENTIFIER, Value: read_num])

node475([Type: listStatement]) --> node474([Type: listStatement])
node474([Type: listStatement]) --> node410([Type: if])
node410([Type: if]) --> node397([Type: EQUALITY])
node397([Type: EQUALITY]) --> node394([Type: IDENTIFIER, Value: op])
node397([Type: EQUALITY]) --> node395([Type: CHAR, Value: +])
node410([Type: if]) --> node408([Type: listStatement])
node408([Type: listStatement]) --> node406([Type: SET])
node406([Type: SET]) --> node398([Type: IDENTIFIER, Value: res])
node406([Type: SET]) --> node405([Type: callOrIndexer])
node405([Type: callOrIndexer]) --> node399([Type: IDENTIFIER, Value: sum])
node405([Type: callOrIndexer]) --> node404([Type: listExpr])
node404([Type: listExpr]) --> node400([Type: IDENTIFIER, Value: a])
node404([Type: listExpr]) --> node403([Type: listExpr])
node403([Type: listExpr]) --> node402([Type: IDENTIFIER, Value: b])
node474([Type: listStatement]) --> node473([Type: listStatement])
node473([Type: listStatement]) --> node428([Type: if])
node428([Type: if]) --> node415([Type: EQUALITY])
node415([Type: EQUALITY]) --> node412([Type: IDENTIFIER, Value: op])
node415([Type: EQUALITY]) --> node413([Type: CHAR, Value: -])
node428([Type: if]) --> node426([Type: listStatement])
node426([Type: listStatement]) --> node424([Type: SET])
node424([Type: SET]) --> node416([Type: IDENTIFIER, Value: res])
node424([Type: SET]) --> node423([Type: callOrIndexer])
node423([Type: callOrIndexer]) --> node417([Type: IDENTIFIER, Value: sub])
node423([Type: callOrIndexer]) --> node422([Type: listExpr])
node422([Type: listExpr]) --> node418([Type: IDENTIFIER, Value: a])
node422([Type: listExpr]) --> node421([Type: listExpr])
node421([Type: listExpr]) --> node420([Type: IDENTIFIER, Value: b])
node473([Type: listStatement]) --> node472([Type: listStatement])
node472([Type: listStatement]) --> node446([Type: if])
node446([Type: if]) --> node433([Type: EQUALITY])
node433([Type: EQUALITY]) --> node430([Type: IDENTIFIER, Value: op])
node433([Type: EQUALITY]) --> node431([Type: CHAR, Value: *])
node446([Type: if]) --> node444([Type: listStatement])

```
node444([Type: listStatement]) --> node442([Type: SET])
node442([Type: SET]) --> node434([Type: IDENTIFIER, Value: res])
node442([Type: SET]) --> node441([Type: callOrIndexer])
node441([Type: callOrIndexer]) --> node435([Type: IDENTIFIER, Value: mul])
node441([Type: callOrIndexer]) --> node440([Type: listExpr])
node440([Type: listExpr]) --> node436([Type: IDENTIFIER, Value: a])
node440([Type: listExpr]) --> node439([Type: listExpr])
node439([Type: listExpr]) --> node438([Type: IDENTIFIER, Value: b])
node472([Type: listStatement]) --> node471([Type: listStatement])
node471([Type: listStatement]) --> node464([Type: if])
node464([Type: if]) --> node451([Type: EQUALITY])
node451([Type: EQUALITY]) --> node448([Type: IDENTIFIER, Value: op])
node451([Type: EQUALITY]) --> node449([Type: CHAR, Value: /])
node464([Type: if]) --> node462([Type: listStatement])
node462([Type: listStatement]) --> node460([Type: SET])
node460([Type: SET]) --> node452([Type: IDENTIFIER, Value: res])
node460([Type: SET]) --> node459([Type: callOrIndexer])
node459([Type: callOrIndexer]) --> node453([Type: IDENTIFIER, Value: div])
node459([Type: callOrIndexer]) --> node458([Type: listExpr])
node458([Type: listExpr]) --> node454([Type: IDENTIFIER, Value: a])
node458([Type: listExpr]) --> node457([Type: listExpr])
node457([Type: listExpr]) --> node456([Type: IDENTIFIER, Value: b])
node471([Type: listStatement]) --> node470([Type: listStatement])
node470([Type: listStatement]) --> node468([Type: callOrIndexer])
node468([Type: callOrIndexer]) --> node465([Type: IDENTIFIER, Value: print_num])
node468([Type: callOrIndexer]) --> node467([Type: listExpr])
node467([Type: listExpr]) --> node466([Type: IDENTIFIER, Value: res])
node498([Type: source])
```

Дерево разбора можно построить с помощью этой утилиты: https://mermaid.live/

Вывод:

В результате выполнения лабораторной работы познакомился с основными принципами формирования дерева разбора. Изучил инструменты позволяющие реализовывать парсинг дерева. Разобрался с конфликтами, которые могут возникать в конфигурации подобных систем.