

|  |
| --- |
| **МИНОБРНАУКИ РОССИИ**  федеральное государственное бюджетное образовательное  учреждение высшего образования  **«Национальный исследовательский университет «МЭИ»**  **Институт информационных и вычислительных технологий** |

Специальность: Прикладная математика и информатика (01.03.02)

Кафедра: ПМИИ

Лабораторная работа №1

по дисциплине

«Основы построения трансляторов»

Выполнил: Шамриков А.И.

Группа: А-05-19

Вариант: 29

Москва

2022

**Постановка задачи**

Разработать алгоритм и реализовать программу для грамматического анализа методом рекурсивного спуска.

Для каждого индивидуального варианта задаётся набор операторов (подмножество языка Паскаль).

**Часть 1.**

Необходимо описать правила языка в форме БНФ. По данным правилам описать грамматику языка. Разработанную грамматику преобразовать к форме автоматной грамматики. Результаты показать преподавателю.

**Часть 2.**

По заданной грамматике построить ДКА, распознающий грамматику, и только её. Результаты показать преподавателю.

**Часть 3.**

Разработать алгоритм синтаксического анализа методом рекурсивного спуска.

Предусмотреть тесты для проверки распознавания всех лексем.

В каждом варианте должен быть реализован следующий функционал:

* оператор присваивания
* арифметические операции +  |- | \* | /
* арифметические выражения
* Типы данных: Integer, Boolean

**Решение:**

**Часть 1.**

<bool> ::= false | true

<int> ::= -<unsigned\_int> | <unsigned\_int>

<record> ::= record <var\_list> <var> end;

<unsigned\_int> ::= <digit> | <digit><unsigned\_int>

<while> ::= while (<bool\_expression>) do <statement\_part> begin | while (<bool\_expression>) <statement>

<bool\_expr> ::= (<bool\_expr> and <bool\_expr>) | (<bool\_expr> or <bool\_expr>) | not <bool-exp> | <simple\_bool\_expr>

<simple\_bool\_expr> ::= <identifier> <comparison\_operator> <identifier> | <bool>

<prog> ::= program <program\_name>; type <type\_part> var <var\_list> begin <statement\_part> end. | program <program\_name>; var <var\_list> begin <statement\_part> end. | program <program\_name>; var <var\_list> begin <statement\_part> end. | program <program\_name>; begin <statement\_part> end.

<program\_name> ::= <identifier>

<identifier> ::= <letter> | <letter><chars>

<chars> ::= <char>|<char><chars>

<char> ::= <letter>|<digit>

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | \_

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<type\_list> ::= <one\_type>; | <one\_type>; <type\_list>

<one\_type> ::= <type\_name> = <i\_b\_type>;

<type\_name> ::= <identifier> | <identifier>, <identifier\_list>

<var\_list> ::= <var>; | <var>; <var\_list>

<var> ::= <identifier\_list> : <i\_b\_type>

<i\_b\_type> ::= Integer | Boolean

<identifier\_list> ::= <identifier> | <identifier>, <identifier\_list>

<statement\_part> ::= <statement>; | <statement>; <statement\_part>

<statement> ::= <assignment> | <write> | <read> | <while> | <if> | begin <statement\_list> end | begin end

<assignment> ::= <identifier> := <expression>

<expression> ::= <identifier> | <expression><operation><expression> | (<expression>) | <int>

<operation> ::= + | - | \* | /

**Грамматика**

1. Алфавит терминальных символов:

**T** = { program, var, begin, end, read, write, if, then, else, while, to, do, integer, boolean, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, A, B, C, D, E, F, G, H, I, G, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, +, -, \_, \*, /, or, and, not, true, false, >=, =, =<, <, >, :=, ",", ".", ";", ":", "(", ")" }

1. Алфавит нетерминальных символов:

**N** = { <prog>, <var\_part>, <statement\_part>, <program\_name>,

<bool>, <var>, <var\_list>, <i\_b\_type>, <type\_list>, <one\_type>, <identifier\_list>,

<statement\_list>, <statement>, <assignment>,

<comparison\_operator>, <identifier>, <chars>, <char>,

<letter>, <digit>, <read>, <write>, <int>, <unsigned\_int>,

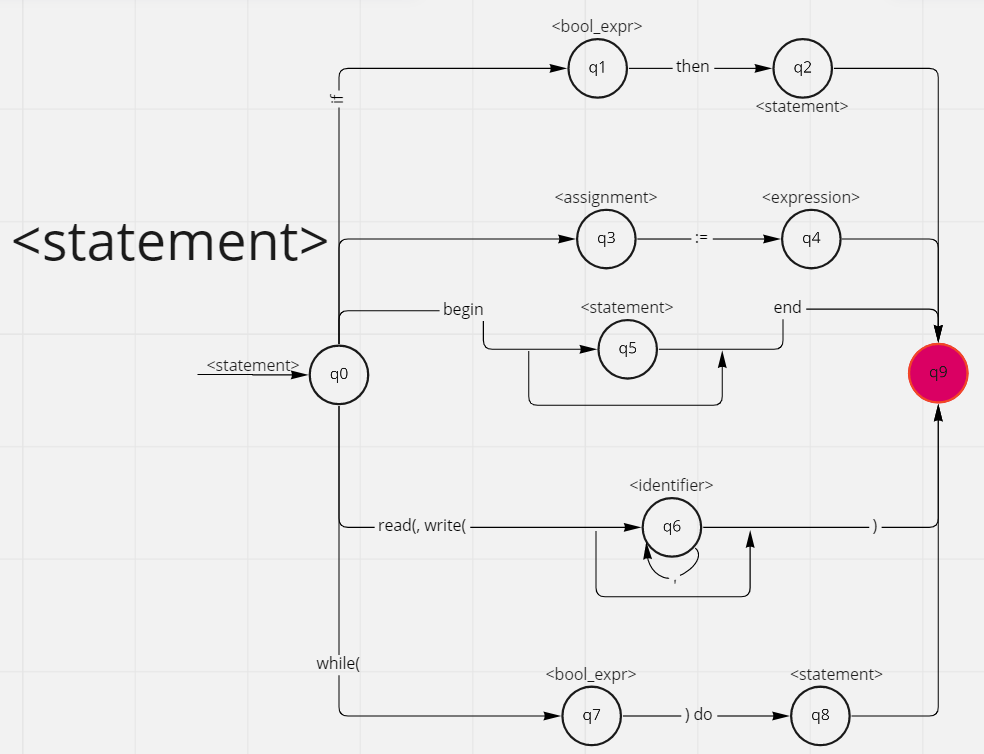
<operation>, <expression>, <while>, <if>, <bool\_expr>,

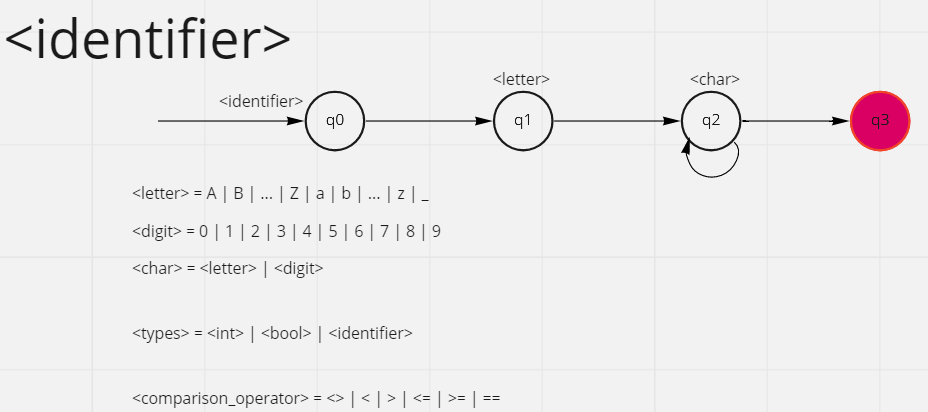
<simple\_bool\_expr> }

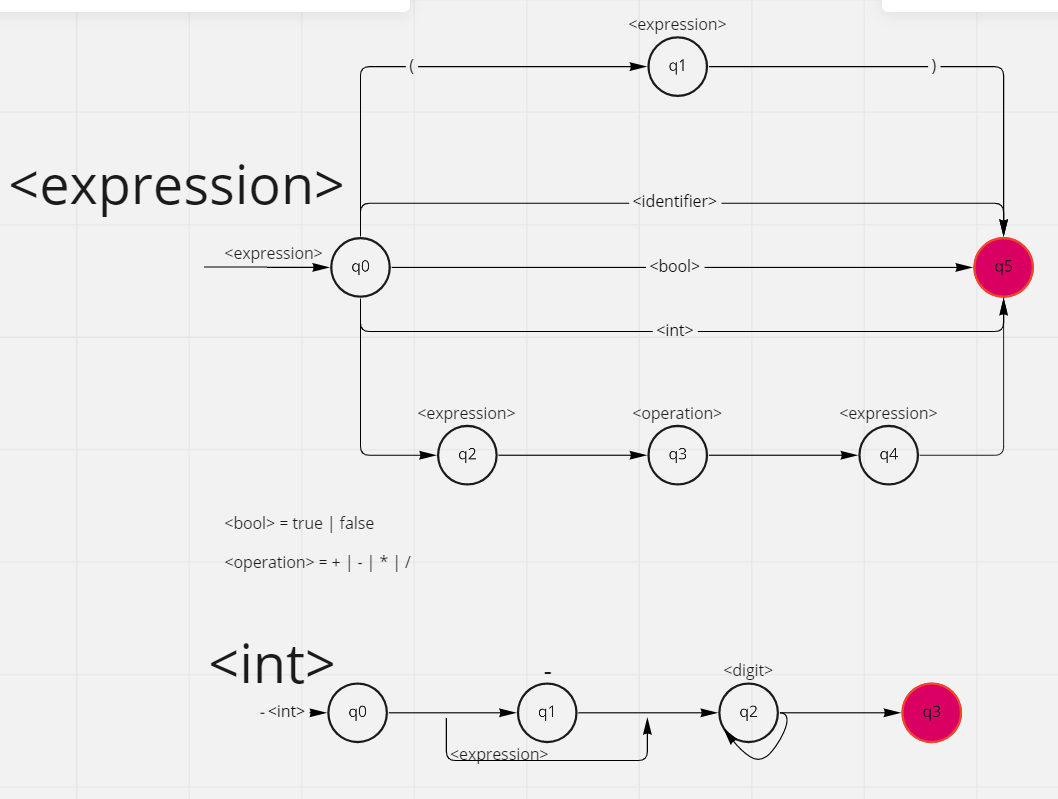
1. Начальный нетерминал

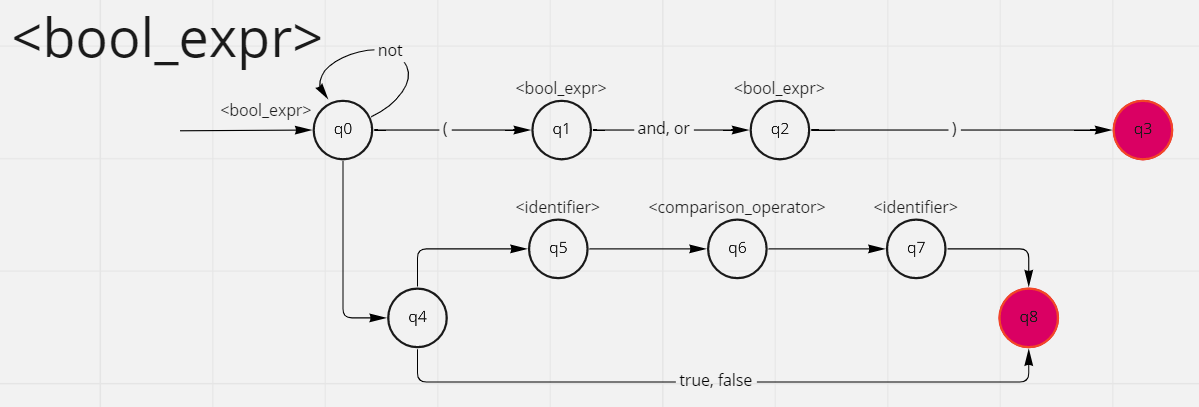
**Z** = { <prog> }

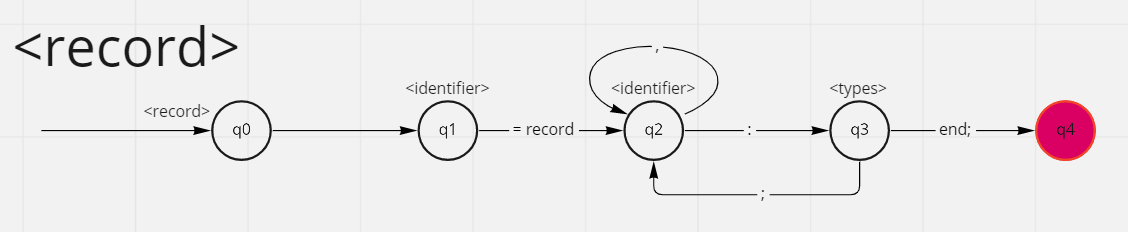
**Часть 2.**

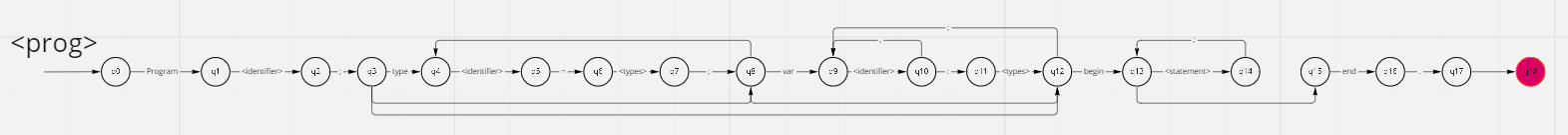


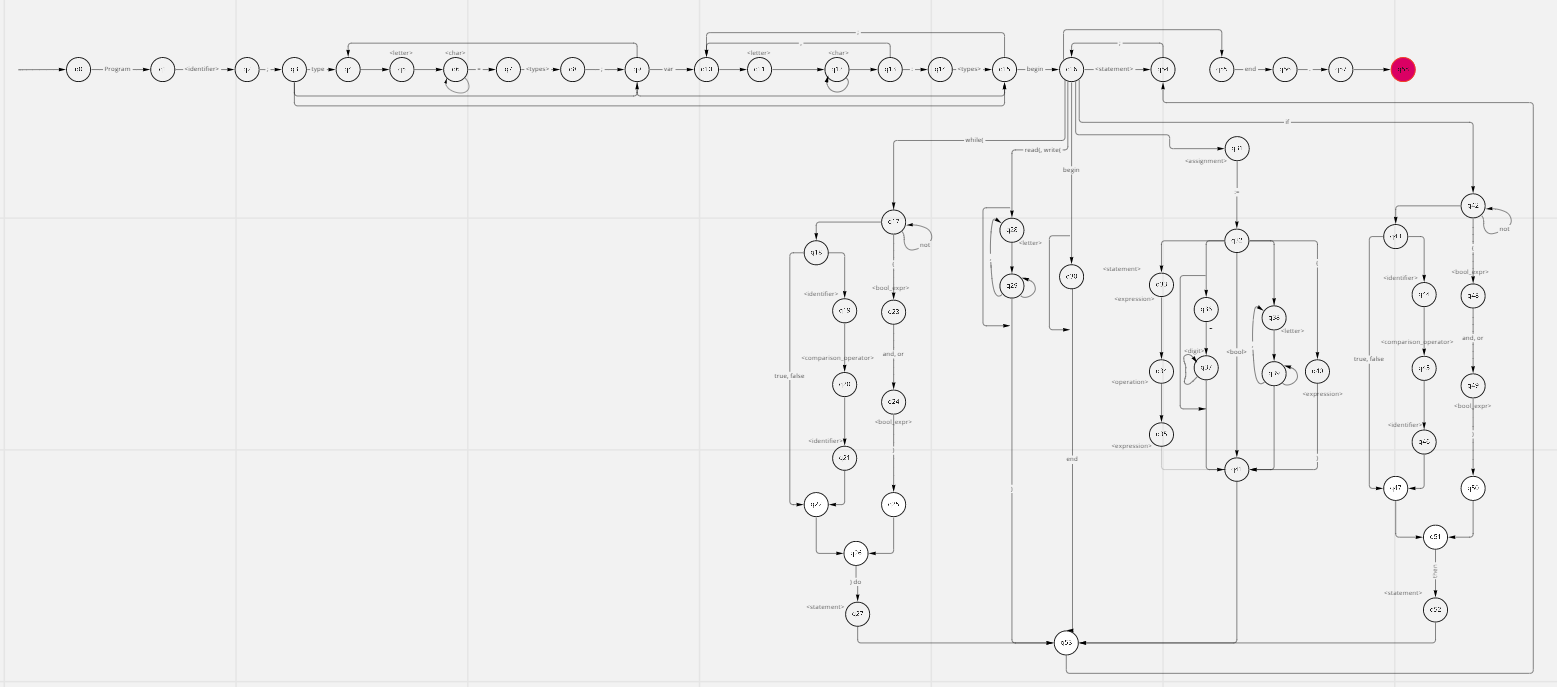












**Часть 3.**

Метод рекурсивногоспуска — это алгоритм нисходящего синтаксического анализа, реализуемый путём взаимного вызова процедур парсинга, где каждая процедура соответствует одному из правил контекстно-свободной грамматики или БНФ. Применения правил последовательно, слева-направо поглощают токены, полученные от лексического анализатора. Это один из самых простых алгоритмов парсинга, подходящий для полностью ручной реализации.

Сначала идет проверка на структуру программы. Для данного варианта – это Program <name>; begin end. || Program <name> var begin end. || Program <name> type begin end; || Program <name> type var begin end.

Для каждого блока программы проверяет корректность синтаксиса, опираясь на ключевые слова.

Код:

|  |
| --- |
| #include <iostream>  #include <fstream>  #include <windows.h>  #include <string>  #include <set>  #include <map>  #include <vector>  using namespace std;  class Parser {  public:  void CheckProgram(string text\_on\_pascal, map <int, vector<int> > mp) {  text = text\_on\_pascal;  result = prog();  if (!result) {  if (fix\_ind == 0) {  for (auto& item : mp)  {  if (find(item.second.begin(), item.second.end(), mainIndex) != item.second.end())  {  cout << endl << "\n\nОшибка в строке: " << item.first;  break;  }  }  }  else {  if (err\_r\_w) { fix\_ind = ind\_br; }  if (err\_last) { fix\_ind = ind\_last; }  for (auto& item : mp)  {  if (find(item.second.begin(), item.second.end(), fix\_ind) != item.second.end())  {  cout << endl << "\n\nОшибка в строке: " << item.first;  break;  }  }  }  cout << "\n\n\nТекст не соответствует программе на языке программирования Pascal\n\n";  }  else {  cout << "\n\n\nТекст прошел тестирование.\n\n";  }  }  private:  string text;  const string \_PROGRAM\_ = "program";  const string \_TYPE\_ = "type";  const string \_VAR\_ = "var";  const string \_BEGIN\_ = "begin";  const string \_END\_ = "end";  const string \_INTEGER\_ = "integer";  const string \_BOOLEAN\_ = "boolean";  const string \_RECORD\_ = "record";  const string \_IF\_ = "if";  const string \_THEN\_ = "then";  const string \_ELSE\_ = "else";  const string \_READ\_ = "read";  const string \_WRITE\_ = "write";  const string \_WHILE\_ = "while";  const string \_TO\_ = "to";  const string \_DOWNTO\_ = "downto";  const string \_DO\_ = "do";  const string \_DIV\_ = "div";  const string \_MOD\_ = "mod";  const string \_OR\_ = "or";  const string \_AND\_ = "and";  const string \_NOT\_ = "not";  const string \_TRUE\_ = "true";  const string \_FALSE\_ = "false";  vector<string> newTypes;  bool result = false;  bool record = false;  int mainIndex = 0;  set <char> setSeparator{ ' ','\t','\r','\n' };  set <string> setRelations{ "<",">","<>","<=",">=","=" };  int num\_err\_str = 1;  int fix\_ind = 0;  bool err\_r\_w = false;  int ind\_br = 0;  bool err\_last = false;  int ind\_last = 0;  bool nextToken() {  if (mainIndex == 0) {  if (text[mainIndex] == '\n') {  num\_err\_str++;  }  }  if (mainIndex < text.size()) {  mainIndex++;  if (text[mainIndex] == '\n') {  if (mainIndex > fix\_ind) {  num\_err\_str++;  }  }  return true;  }  return false;  }  bool checkScapes(int ind) {  if (text[ind] == ' ') {  return true;  }  return false;  }  bool nextTokenSkip() {  bool fl = nextToken();  if (fl) {  while (mainIndex < text.size() && (setSeparator.find(text[mainIndex]) != setSeparator.end())) {  mainIndex++;  if (text[mainIndex] == '\n') {  if (mainIndex > fix\_ind) {  num\_err\_str++;  }  }  }  }  return fl;  }  char getToken(bool skipSeparator = false) {  if (skipSeparator && (setSeparator.find(text[mainIndex]) != setSeparator.end())) {  nextTokenSkip();  }  return text[mainIndex];  }  bool setIndex(int to\_change) {  int buf = 0;  if (to\_change < text.size()) {  if (mainIndex >= fix\_ind) {  fix\_ind = mainIndex;  }  mainIndex = to\_change;  return true;  }  return false;  }  bool terminal(string word, bool skip\_sep = false) {  if (skip\_sep) {  getToken(true);  }  for (auto i : word) {  if (i == getToken() && nextToken()) {  }  else {  return false;  }  }  return true;  }  bool prog() {  if (headProgram() && getToken(true) == ';' && nextToken() && block() && last()) {  return true;  }  return false;  }  bool last() {  int index = mainIndex;  if (getToken(true) == '.') {  nextToken();  while (mainIndex < text.size() && (setSeparator.find(text[mainIndex]) != setSeparator.end())) {  mainIndex++;  }  if (mainIndex == text.size()) return true;  ind\_last = mainIndex - 1;  err\_last = true;  return false;  }  ind\_last = mainIndex - 1;  err\_last = true;  return false;  }  bool headProgram() {  if (terminal(\_PROGRAM\_, true) && nextTokenSkip() && identifier()) {  return true;  }  return false;  }  bool block() {  int index = mainIndex;  if ((typeSect() && descriptionsSect() && operatorsSect()) || (typeSect() && operatorsSect())  || (descriptionsSect() && operatorsSect()) || (setIndex(index) && operatorsSect())) {  return true;  }  return false;  }  bool typeSect() {  if (terminal(\_TYPE\_, true) && nextTokenSkip() && descriptionsType() && getToken(true) == ';' && nextToken()) {  int index = mainIndex;  while (descriptionsVars() && getToken(true) == ';' && nextToken()) {  index = mainIndex;  }  mainIndex = index;  return true;  }  return false;  }  bool descriptionsType() {  if (identifier(true) && getToken(true) == '=' && nextToken() && types("type")) {  if (record) {  if (!recordType()) {  return false;  }  }  return true;  }  return false;  }  bool endCheck() {  string buf;  buf += text[mainIndex];  buf += text[mainIndex + 1];  buf += text[mainIndex + 2];  buf += text[mainIndex + 3];  if (buf == "end;") {  return true;  }  else  return false;  }  bool recordType() {  int index;  while (descriptionsVars("type") && getToken(true) == ';' && nextToken()) {  index = mainIndex;  }    return true;  }  bool descriptionsSect() {  int index = mainIndex;  if (varsSect()) {  return true;  }  return false;  }  bool varsSect() {  if (terminal(\_VAR\_, true) && nextTokenSkip() && descriptionsVars() && getToken(true) == ';' && nextToken()) {  int index = mainIndex;  while (descriptionsVars() && getToken(true) == ';' && nextToken()) {  index = mainIndex;  }  mainIndex = index;  return true;  }  return false;  }  bool descriptionsVars(string flag = "var") {  if (listVarsNames() && getToken(true) == ':' && nextToken() && types(flag)) {  return true;  }  return false;  }  bool listVarsNames() {  if (identifier()) {  int index = mainIndex;  while (getToken(true) == ',' && nextToken() && identifier()) {  index = mainIndex;  }  mainIndex = index;  return true;  }  return false;  }  bool operatorsSect() {  if (terminal(\_BEGIN\_, true) && nextTokenSkip()) {  int index = mainIndex;  if ((operatorsList() && terminal(\_END\_, true)) || (setIndex(index) && terminal(\_END\_))) {  return true;  }  }  return false;  }  bool operator\_() {  int index = mainIndex;  if (IOoperator() || (setIndex(index) && ifoperator()) || (setIndex(index) && whileoperator()) ||  (setIndex(index) && operatorsAssignment()) || (setIndex(index) && compoundOperator())) {  return true;  }  return false;  }  bool lstWrite(string state = "write") {  int index = mainIndex;  if (setIndex(index)) {  if (state == "write" && (expr1() || getToken() == ')')) {  index = mainIndex;  char parse = getToken(true);  if (parse != ')') {  if (parse == ',') {  nextToken();  while (parse == ',' && expr1()) {  index = mainIndex;  parse = getToken(true);  nextToken();  }  if (parse != ')') return false;  return true;  }  if (expr1()) return true;  return false;  }  nextToken();  return true;  }  else if (state == "read" && (expr2() || getToken() == ')')) {  index = mainIndex;  char parse = getToken(true);  if (parse != ')') {  if (parse == ',') {  nextToken();  while (parse == ',' && expr1()) {  index = mainIndex;  parse = getToken(true);  nextToken();  }  if (parse != ')') return false;  return true;  }  if (expr1()) return true;  return false;  }  nextToken();  return true;  }  else  return false;  }  else  return false;  }  bool IOoperator() {  getToken(true);  int index = mainIndex;  if (terminal(\_WRITE\_) && getToken(true) == '(' && nextToken()) {  ind\_br = mainIndex - 1;  int index = mainIndex;  if (getToken(true) == ')' && nextToken() || setIndex(index) && lstWrite())  {  return true;  }  err\_r\_w = true;  return false;  }  else if (setIndex(index) && terminal(\_READ\_) && getToken(true) == '(' && nextToken()) {  ind\_br = mainIndex - 1;  int index = mainIndex;  if ((getToken(true) == ')' && nextToken()) || (setIndex(index) && lstWrite("read")))  {  return true;  }  err\_r\_w = true;  return false;  }  return false;  }  bool ifoperator() {  if (terminal(\_IF\_, true) && logExpr() && terminal(\_THEN\_, true) && operator\_()) {  return true;  }  return false;  }  bool whileoperator() {  if (terminal(\_WHILE\_, true) && logExpr() && terminal(\_DO\_, true) && operator\_()) {  return true;  }  return false;  }  bool compoundOperator() {  if (terminal(\_BEGIN\_, true) && nextTokenSkip()) {  int index = mainIndex;;  if ((operatorsList() && terminal(\_END\_, true)) || (setIndex(index) && terminal(\_END\_))) {  return true;  }  }  return false;  }  bool operatorsList() {  if (operator\_() && getToken() == ';') {  int index = mainIndex;  bool fl = true;  while (fl && getToken(true) == ';' && nextToken()) {  index = mainIndex;  if (operator\_() && getToken() == ';') {  index = mainIndex;  }  else {  fl = false;  }  }  mainIndex = index;  return true;  }  return false;  }  bool operatorsAssignment() {  if (var() && terminal(":=", true) && expr()) {  return true;  }  return false;  }  bool expr1() {  int index = mainIndex;  if (getToken(true) == ')' || getToken(true) == ',') return false;  if ((getToken(true) && identifier() && getToken() == '.' && mainIndex++ && identifier())  || (setIndex(index) && getToken(true) && var())  || (getToken(true) && texting()) || (setIndex(index) && getToken(true) && expr())) {  return true;  }  return false;  }  bool expr2() {  int index = mainIndex;  if (getToken(true) == ')' || getToken(true) == ',') return false;  if ((getToken(true) && identifier() && getToken() == '.' && mainIndex++ && identifier())  || (setIndex(index) && getToken(true) && var())) {  return true;  }  return false;  }  bool texting() {  char st = getToken(true);  int index = mainIndex;  if (st == '\'') {  nextTokenSkip();  while (isalpha(getToken()) || isdigit(getToken())) {  nextTokenSkip();  }  if (getToken() == '\'')  {  nextTokenSkip();  return true;  }  }  return false;  }  bool expr() {  int index = mainIndex;  if (arithmExpr() || (setIndex(index) && logExpr())) {  return true;  }  return false;  }  bool logExpr() {  int index = mainIndex;;  if (relationship() || (setIndex(index) && simpleLogExpr())) {  return true;  }  return false;  }  bool simpleLogExpr() {  if (logTerm()) {  int index = mainIndex;  if (terminal(\_OR\_, true) && simpleLogExpr()) {  index = mainIndex;  }  mainIndex = index;  return true;  }  return false;  }  bool logTerm() {  if (logMultiplier()) {  int index = mainIndex;  if (terminal(\_AND\_, true) && logTerm()) {  index = mainIndex;  }  mainIndex = index;  return true;  }  return false;  }  bool logMultiplier() {  getToken(true);  int index = mainIndex;  if (logConst()  || (setIndex(index) && terminal(\_NOT\_) && logMultiplier())  || (setIndex(index) && terminal("true"))  || (setIndex(index) && getToken(true) == '(' && nextToken() && logExpr() && getToken(true) == ')' && nextToken())  ) {  return true;  }  return false;  }  bool relationship() {  int index = mainIndex;  if ((arithmExpr() && OPcomparison() && arithmExpr())  || (setIndex(index) && simpleLogExpr() && OPcomparison() && simpleLogExpr())  ) {  return true;  }  return false;  }  bool arithmExpr() {  int index = mainIndex;  if (OPaddition()) {  index = mainIndex;  }  else {  mainIndex = index;  }  if (term()) {  index = mainIndex;  while (OPaddition() && term()) {  index = mainIndex;  }  mainIndex = index;  return true;  }  return false;  }  bool term() {  if (multiplier()) {  int index = mainIndex;  while (OPmultiplication() && multiplier()) {  index = mainIndex;  }  mainIndex = index;  return true;  }  return false;  }  bool multiplier() {  getToken(true);  int index = mainIndex;  if (intNumber()  || (setIndex(index) && var())  || (setIndex(index) && getToken(true) == '(' && nextToken() && arithmExpr() && getToken(true) == ')' && nextToken())  ) {  return true;  }  return false;  }  bool var() {  int index = mainIndex;  if ((identifier() && getToken() == '.' && mainIndex++ && identifier()) || (setIndex(index) && identifier())) {  return true;  }  return false;  }  bool intNumber() {  bool firstToken = getToken(true);  int isNumber = false;  if (firstToken == '-' || firstToken == '+') {  nextToken();  }  while (isdigit(getToken()) && nextToken()) {  isNumber = true;  }  if (isNumber) {  }  return isNumber;  }  bool OPaddition() {  getToken(true);  int index = mainIndex;  if ((getToken() == '+' && nextToken())  || (setIndex(index) && (getToken() == '-' && nextToken()))  ) {  return true;  }  return false;  }  bool OPmultiplication() {  getToken(true);  int index = mainIndex;  if ((getToken() == '\*' && nextToken())  || (setIndex(index) && checkScapes(mainIndex - 1) && terminal(\_DIV\_) && checkScapes(mainIndex)) || (setIndex(index) && checkScapes(mainIndex - 1) && terminal(\_MOD\_) && checkScapes(mainIndex)) || (setIndex(index) && (getToken() == '/' && nextToken()))  ) {  return true;  }  return false;  }  bool logConst() {  getToken(true);  int index = mainIndex;  if (terminal(\_TRUE\_)  || (setIndex(index) && terminal(\_FALSE\_))  ) {  return true;  }  return false;  }  bool OPcomparison() {  getToken(true);  int index = mainIndex;  for (auto i : setRelations) {  if (setIndex(index) && terminal(i)) {  return true;  }  }  return false;  }  bool identifier(bool save = false) {  int index = mainIndex;  if (save) {  string buf;  while (text[mainIndex] != ',' && text[mainIndex] != ' ' && text[mainIndex] != '=') {  buf += text[mainIndex];  mainIndex++;  }  newTypes.push\_back(buf);  mainIndex = index;  }  int dopIndex;  if (isalpha(getToken(true)) && nextToken()) {  while ((isalpha(getToken()) || isdigit(getToken())) && nextToken()) {  }  dopIndex = mainIndex;  if (!((setIndex(index) && (terminal(\_TRUE\_) || terminal(\_FALSE\_)))))  {  mainIndex = dopIndex;  return true;  }  else mainIndex = dopIndex;  }  return false;  }  bool types(string block = "var") {  getToken(true);  int index = mainIndex;  string buf;  while (text[mainIndex] != ' ' && text[mainIndex] != ';') {  buf += text[mainIndex];  mainIndex++;  }  for (int i = 0; i < newTypes.size(); i++) {  if (newTypes[i] == buf)  return true;  }  mainIndex = index;  if (block == "var") {  if (terminal(\_INTEGER\_)  || (setIndex(index) && terminal(\_BOOLEAN\_))) {  return true;  }  }  else {  if (terminal(\_INTEGER\_)  || (setIndex(index) && terminal(\_BOOLEAN\_))) {  return true;  }  if (setIndex(index) && terminal(\_RECORD\_)) {  record = true;  return true;  }  }  return false;  }  };  int main()  {  setlocale(LC\_ALL, "Russian");  int select;  cout << "Имя файла: ";  char name[40];  cin.getline(name, 40);  fstream fin;  fin.open(name, fstream::in);  if (fin.is\_open()) {  char ch;  string textOnPascal;  map <int, vector<int> > mp;  vector<int> line;  int num\_str = 1;  int ind = 0;  while (fin.get(ch))  {  if (ch != '\n') {  line.push\_back(ind);  }  else {  line.push\_back(ind);  mp[num\_str] = line;  num\_str++;  line.clear();  }  textOnPascal.append(1, ch);  ind++;  }  mp[num\_str] = line;  cout << "\n\n";  cout << textOnPascal;  // cout << endl << textOnPascal.find("\0");  Parser parse;  parse.CheckProgram(textOnPascal, mp);  }  else {  cout << "Файл не найден";  }  return 0;  } |

Тесты:

|  |  |
| --- | --- |
| program a1;  begin  end. | Текст прошел тестирование |
| program a1  begin  end. | Ошибка в строке: 1 Текст не соответствует программе на языке программирования Pascal |
| program one;  type  a = record  b, c: integer;  d: boolean;  end;  var  i: a;  qw, we, er: integer;  begin  read(qw, we);  while (true) do  begin  read(i.b);  if (qw > i.b) then  er := we + i.b;  end;    write(er);  end. | Текст прошел тестирование |
| program one;  type  a = record  b, c: integer;  d: boolean;  end;  var  i: a;  qw, we, er: integer  begin  read(qw, we);  while (true) do  begin  read(i.b);  if (qw > i.b) then  er := we + i.b;  end;    write(er);  end. | Ошибка в строке: 9 Текст не соответствует программе на языке программирования Pascal |
| program one;  type  a = record  b, c: integer;  d: boolean;  end;  var  i: b;  qw, we, er: integer  begin  read(qw, we);  while (true) do  begin  read(i.b);  if (qw > i.b) then  er := we + i.b;  end;    write(er);  end. | Ошибка в строке: 8 Текст не соответствует программе на языке программирования Pascal |
| program 3one;  type  a = record  b, c: integer;  d: boolean;  end;  var  i: a;  qw, we, er: integer  begin  read(qw, we);  while (true) do  begin  read(i.b);  if (qw > i.b) then  er := we + i.b;  end;    write(er);  end. | Ошибка в строке: 1 Текст не соответствует программе на языке программирования Pascal |
| program one;  type  a = record  b, c: integer;  d: boolean;  end;  var  i: a;  qw, we, er: integer  begin  read(qw, we);  while (true) do  begin  read(i.b);  if (qw > i.b) then  er := we + i.b;  end;    write(er);  end.  read(er); | Ошибка в строке: 20 Текст не соответствует программе на языке программирования Pascal |
| program one;  type  a = record  b, c: integer;  d: boolean;  end;  var  i: a;  qw, we, er: integer  begin  read(qw, we);  while (true) do  begin  read(i.b);  if (qw > i.b) then  er = we + i.b;  end;    write(er);  end. | Ошибка в строке: 16 Текст не соответствует программе на языке программирования Pascal |