Министерство образования и науки РФ

Государственное бюджетное образовательное учреждение

высшего профессионального образования

"Тверской государственный университет"

Факультет прикладной математики и кибернетики

Направление "Прикладная математика и информатика"

**Отчет по учебно-вычислительной практике**

Выполнила:

студентка 34 группы

Киут Любовь Леонидовна

Проверил:

кандидат физико-математических наук, доцент кафедры информатики

Дадеркин Дмитрий Ольгердович

Тверь 2017

**Содержание:**

Постановка задачи………………………………………...3

Построение синтаксического анализатора……………...4

Пример работы программы………………………………6

Листинг исходного кода анализатора……………….......8

**Постановка задачи**

Необходимо построить синтаксический анализатор для подмножества языка Basic - MINI-BASIC.

На вход синтаксическому анализатору подаётся список лексем, который формируется написанным нами ранее лексическим анализатором.

На выходе синтаксический анализатор определяет, является ли исходная программа программой, написанной на языке MINI-BASIC, и выдает список атомов.

Таким образом, компилятор на этапе синтаксического анализа решает две задачи:

* распознавание синтаксических единиц программы и проверка их правильности;
* интерпретация конструкций языка

Для решения первой задачи необходимо выявить в лексической свертке программы, приходящей на синтаксический анализ, структуры вхождений лексем и понятий и проверить, удовлетворяет ли эта структура синтаксису языка. Синтаксическая структура программы показывает существующие в ней связи между соответствующими частями и тем самым помогает вскрывать смысл программы.

**Построение синтаксического анализатора**

Синтаксический анализатор — это часть транслятора, которая отвечает за выявление основных синтаксических конструкций входного языка. В задачу синтаксического анализа входит: найти и выделить основные синтаксические конструкции в тексте входной программы, установить тип и проверить правильность каждой синтаксической конструкции и представить синтаксические конструкции в виде, удобном для дальнейшей генерации текста результирующей программы.

В основе синтаксического анализатора лежит распознаватель текста входной программы на основе грамматики входного языка. Распознаватель дает ответ на вопрос о том, принадлежит или нет цепочка входных символов заданному языку. Все задачи по проверке синтаксиса входного языка могут быть решены на этапе синтаксического разбора. Сканер только позволяет избавить сложный по структуре синтаксический анализатор от решения примитивных задач по выявлению и запоминанию лексем исходной программы.

Синтаксический анализатор воспринимает выход лексического анализатора и разбирает его в соответствии с грамматикой входного языка. Однако в грамматике входного языка программирования обычно не уточняется, какие конструкции следует считать лексемами. Примерами конструкций, которые обычно распознаются во время лексического анализа, служат ключевые слова, константы и идентификаторы. Но эти же конструкции могут распознаваться и синтаксическим анализатором. На практике не существует жесткого правила, определяющего, какие конструкции должны распознаваться на лексическом уровне, а какие надо оставлять синтаксическому анализатору. Обычно это определяет разработчик компилятора. Основу любого синтаксического анализатора всегда составляет распознаватель, построенный на основе какого-либо класса КС-грамматик.

Синтаксический анализатор проверяет порядок следование лексем для любого языка программирования. В основе синтаксического анализатора лежит аппарат формальных языков и грамматик.

Формальной грамматикой называется совокупность четырех объектов.

Va – нетерминальные символы.

R – правила грамматики

Vt – терминальные символы (символы допускаемые языком)

В теории компиляторов используется контекстно-свободная грамматика, для которой a меняется только на 1 нетерминальный символ.

Вывод – это замена, одного нетерминального символа на правую часть правила, ему соответствующему.

Полное описание входного языка должно быть задано в грамматике входного языка, которая строится исполнителем на первом этапе работы. Грамматика входного языка должна предусматривать любые входные цепочки, удовлетворяющие требованиям синтаксиса исходного языка.

Данный синтаксический анализатор был построен по следующей

L-атрибутной грамматике:

*<S> -> <A> <B> END*

*<A> -> <L> <A>*

*<A> -> e*

*<L> -> <B> <C>*

*<B> -> CONST*

*<C> -> LET Var = <F>*

*<C> -> GOTO <B>*

*<C> -> IF <J> <B>*

*<C> -> FOR Var = <F> TO <F> STEP <F> <A> <B> NEXT*

*<C> -> RETURN*

*<J> -> <F> <J'>*

*<J'> -> =<F> <J'>*

*<J'> -> < <F>*

*<J'> -> > <F>*

*<F> -> <M> <N> <F'>*

*<F'> -> +<M> <N> <F'>*

*<F'> -> -<M> <N> <F'>*

*<F'> -> e*

*<N> -> e*

*<N> -> \*<M> <N>*

*<N> -> /<M> <N>*

*<M> -> (<M> <N> <F'>)*

*<M> -> c*

*<M> -> Var*

**Пример работы программы**

Для демонстрирования работы синтаксического анализатора будем подавать на вход разные текстовые файлы: файл, который будет содержать ошибки и файл без ошибок.

Подадим на вход нашему лексическому анализатору файл с исходным кодом, представленным на рисунке:

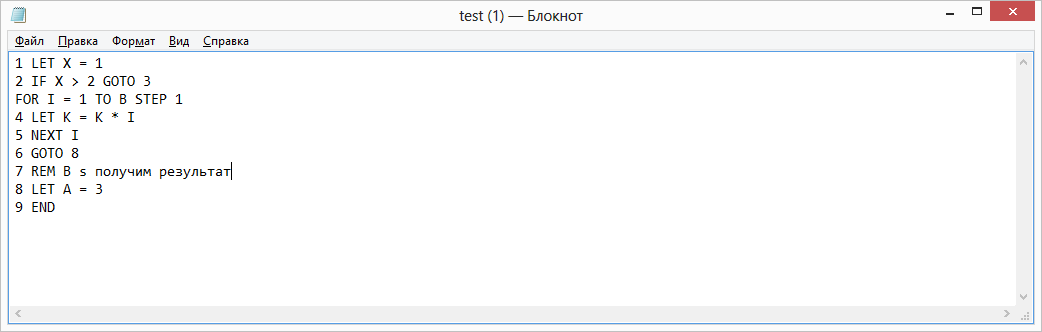
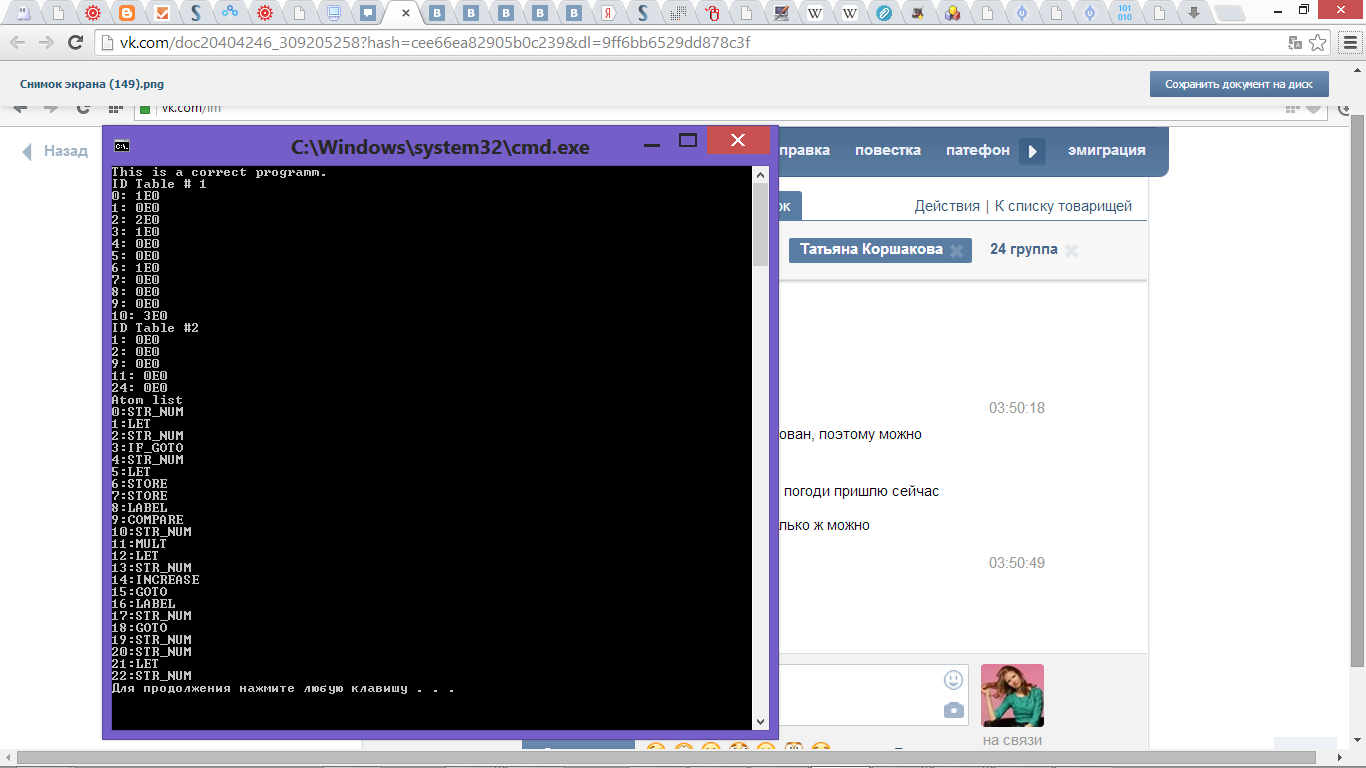
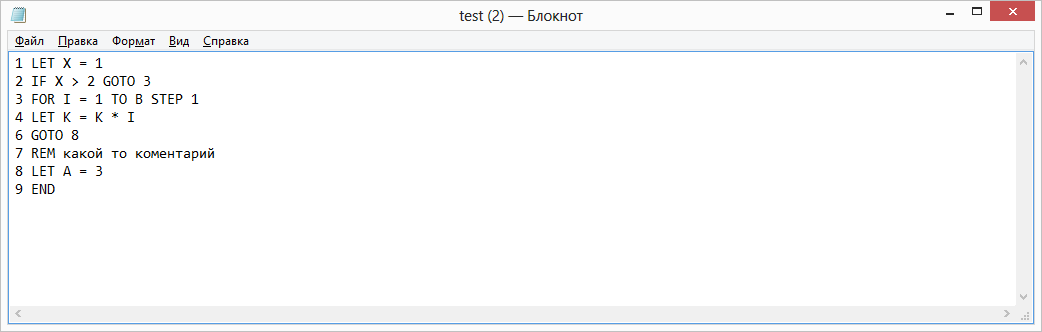


Рисунок 3

Скриншот результата работы синтаксического анализатора – рисунок 4:

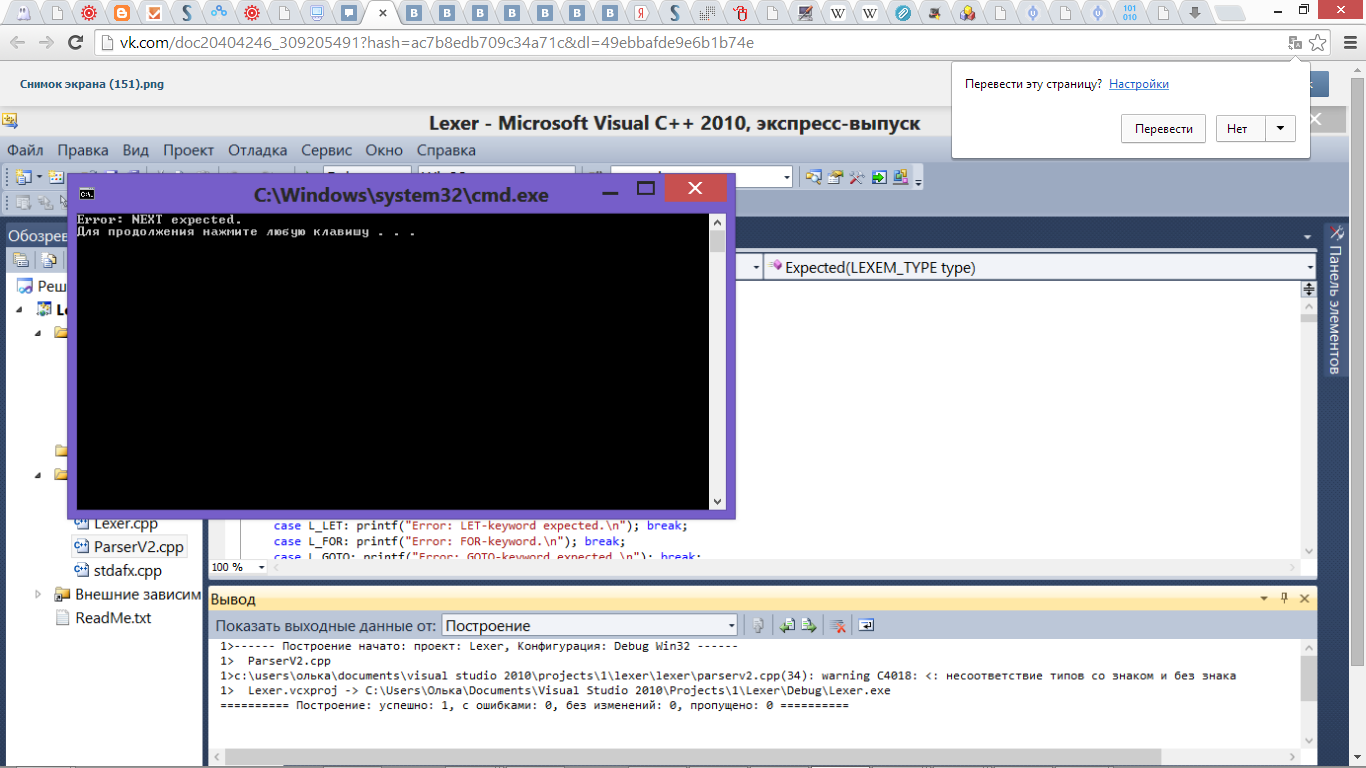
Символ # – указатель на таблицу имен

Теперь подадим на вход файл, содержащий ошибки (рисунок 5):



Рисунок

Скриншот результата выполнения программы – рисунок 6:



Рисунок

**Листинг исходного кода анализатора**

#include "stdafx.h"

#include "LexBlock.h"

enum ACTIONS { A\_END, A\_STR\_NUM, A\_SET\_STR\_NUM, A\_LET, A\_GOTO, A\_STORE\_GOTO, A\_IF\_GOTO, A\_RETURN, A\_STORE, A\_LABEL, A\_COMPARE, A\_INCREASE, A\_ADD, A\_SUBTR,

A\_MULT, A\_DIV, A\_EXP, A\_PLUS, A\_MINUS, A\_CONTROL};

enum OPERS { VAR, CONST };

extern Lex look;

extern vector<Lex> \*lexVector;

extern int current;

extern vector<Constant> Constant\_Pool;

extern Constant NameSpace[287];

struct Operand

{

OPERS type;

int index;

};

struct ATOM

{

ACTIONS type;

int Line;

int relatSign;

int ResultIndex1;

int ResultIndex;

int VariableIndex;

int leftOper;

int rightOper;

ATOM \*AtomPointer;

//По умолчанию, они все ноль

ATOM()

{

type = A\_END;

ResultIndex = 0;

VariableIndex = 0;

Line = 0;

relatSign = 0;

ResultIndex1 = 0;

AtomPointer = NULL;

leftOper = 0;

rightOper = 0;

}

};

extern vector<ATOM> atom\_vector;

void Parse( vector<Lex> \*vec );

void Programm();

void ProgrammBody();

void Step();

void OtherLines();

void E\_List();

void Term();

void T\_List();

void Factor();

void F\_List();

void Primary();

void Match(LEX\_TYPE type);

void GetLexem();

void Expected(LEX\_TYPE type);

void relat();

int current\_line;

void Expected(LEX\_TYPE type)

{

//Вывод сообщения, что ожидалось что-то

switch( type )

{

case STR: printf("Error: Line number expected.\n"); break;

case OPER: printf("Error: Operand expected.\n"); break;

case ARIF\_OP: printf("Error: Operation sign expected.\n"); break;

case OTN\_OP: printf("Error: relat sign expected.\n"); break;

case NEXT: printf("Error: NEXT expected.\n"); break;

case LET: printf("Error: LET expected.\n"); break;

case FOR: printf("Error: FOR expected.\n"); break;

case GOTO: printf("Error: GOTO expected.\n"); break;

case GOSUB: printf("Error: GOSUB expected.\n"); break;

case LBR: printf("Error: Left bracket expected.\n"); break;

case RBR: printf("Error: Right bracket expected.\n"); break;

case IF: printf("Error: IF expected.\n"); break;

case RETURN: printf("Error: RETURN expected.\n"); break;

case END: printf("Error: END expected.\n"); break;

case TO: printf("Error: TO expected.\n"); break;

case STEP: printf("Error: STEP expected.\n"); break;

default: printf("Error! \n");

}

exit(-1);

}

void GetLexem()

{

if ( current < lexVector->size() )

{

look = lexVector->at( current );

current++;

}

}

void Match(LEX\_TYPE type)

{

if ( look.type == type ) GetLexem();

else Expected(type);

}

void Primary()

{

//Встретилась скобка

if ( look.type == LBR)

{

Match( LBR);

relat();

Match( RBR );

return;

}

//Иначе, это либо переменная, либо константа(нулевой указатель на таблицу имен)

Constant c;

if ( look.type == OPER )

{

//Константа

if ( look.ptr == 0 )

c = look.n\_const;

//Переменная

else

c = NameSpace[ look.ptr ];

Constant\_Pool.push\_back( c );

}

Match( OPER );

}

void F\_List()

{

while ( look.type == AR\_OP && (look.otn == 5) )

{

Match( ARIF\_OP );

int Left\_Operand = Constant\_Pool.size() - 1;

Primary();

//Где будет храниться результат экспоненты

Constant\_Pool.push\_back( Constant(0,0,0) );

//Выдать атом ЭКСП

ATOM atom;

atom.type = A\_EXP;

atom.leftOper = Left\_Operand;

atom.rightOper = Constant\_Pool.size() - 2;

atom.ResultIndex = Constant\_Pool.size() - 1;

atom\_vector.push\_back( atom );

}

}

void Factor()

{

Primary();

F\_List();

}

void T\_List()

{

while ( look.type == AR\_OP && (look.otn == 3 || look.otn == 4) )

{

if ( look.otn == 3 )

{

Match( ARIF\_OP );

int Left\_Operand = Constant\_Pool.size() - 1;

Factor();

//Где будет храниться результат умножения

Constant\_Pool.push\_back( Constant(0,0,0) );

//Выдать атом УМНОЖ

ATOM atom;

atom.type = A\_MULT;

atom.leftOper = Left\_Operand;

atom.rightOper = Constant\_Pool.size() - 2;

atom.ResultIndex = Constant\_Pool.size() - 1;

atom\_vector.push\_back( atom );

continue;

}

if ( look.otn == 4 )

{

Match( ARIF\_OP );

int Left\_Operand = Constant\_Pool.size() - 1;

Factor();

//Где будет храниться результат деления

Constant\_Pool.push\_back( Constant(0,0,0) );

//Выдать атом РАЗДЕЛИТЬ

ATOM atom;

atom.type = A\_DIV;

atom.leftOper = Left\_Operand;

atom.rightOper = Constant\_Pool.size() - 2;

atom.ResultIndex = Constant\_Pool.size() - 1;

atom\_vector.push\_back( atom );

continue;

}

printf("Error: Multop expected.\n");

exit(-1);

}

}

void Term()

{

Factor();

T\_List();

}

void E\_List()

{

while ( look.type == ARIF\_OP && (look.otn == 1 || look.otn == 2) )

{

if ( look.otn == 1 )

{

Match( ARIF\_OP );

int Left\_Operand = Constant\_Pool.size() - 1;

Factor();

//Где будет храниться результат сложения

Constant\_Pool.push\_back( Constant(0,0,0) );

//Выдать атом СЛОЖ

ATOM atom;

atom.type = A\_ADD;

atom.leftOper = Left\_Operand;

atom.rightOper = Constant\_Pool.size() - 2;

atom.ResultIndex = Constant\_Pool.size() - 1;

atom\_vector.push\_back( atom );

continue;

}

if ( look.otn == 2 )

{

Match( ARIF\_OP );

int Left\_Operand = Constant\_Pool.size() - 1;

Factor();

//Где будет храниться результат вычитания

Constant\_Pool.push\_back( Constant(0,0,0) );

//Выдать атом ВЫЧИТ

ATOM atom;

atom.type = A\_SUBTR;

atom.leftOper = Left\_Operand;

atom.rightOper = Constant\_Pool.size() - 2;

atom.ResultIndex = Constant\_Pool.size() - 1;

atom\_vector.push\_back( atom );

continue;

}

printf("Error: Addop expected.\n");

exit(-1);

}

}

void relat()

{

Term();

E\_List();

}

void OtherLines()

{

//Если это не номер строки, то скажем, что ожидался номер строки

if ( look.type != STR )

Expected( STR );

current\_line = look.str;

Match(STR);

//Выдать атом НОМЕРСТРОКИ

ATOM atom;

atom.type = A\_STR\_NUM;

atom.Line = current\_line;

atom\_vector.push\_back( atom );

ProgrammBody();

}

void Step()

{

if ( look.type == STEP )

{

Match( STEP );

relat();

return;

}

//Иначе, закинуть константу 1

Constant\_Pool.push\_back( Constant(0,1,0) );

}

void ProgrammBody()

{

//Пустой оператор

if ( look.type == STR )

{

current\_line = look.str;

Match(STR);

//Выдать атом НОМЕРСТРОКИ

ATOM atom;

atom.type = A\_STR\_NUM;

atom.Line = current\_line;

atom\_vector.push\_back( atom );

ProgrammBody();

return;

}

//Присваивание

if ( look.type == LET )

{

int VariableToAssign = look.ptr;

Match( LET );

relat();

//Выдать атом ПРИСВ

ATOM atom;

atom.type = A\_LET;

atom.VariableIndex = VariableToAssign;

atom.ResultIndex = Constant\_Pool.size() - 1;

atom\_vector.push\_back( atom );

OtherLines();

return;

}

//GOTO-оператор

if ( look.type == GOTO )

{

int LineToJump = look.str;

Match( GOTO );

//Выдать атом ПЕРЕХОД НА

ATOM atom;

atom.type = A\_GOTO;

atom.Line = LineToJump;

atom\_vector.push\_back( atom );

OtherLines();

return;

}

//IF-оператор

if ( look.type == IF )

{

Match( IF );

relat();

int relat1 = Constant\_Pool.size() - 1;

int otnSign;

if ( look.type == OTN\_OP )

otnSign = look.otn;

Match( OTN\_OP );

relat();

int relat2 = Constant\_Pool.size() - 1;

int LineToJump;

if ( look.type == GOTO )

LineToJump = look.str;

Match( GOTO );

//Выдать атом УСЛПЕРЕХ

ATOM atom;

atom.type = A\_IF\_GOTO;

atom.relatSign = otnSign;

atom.ResultIndex = relat1;

atom.ResultIndex1 = relat2;

atom.Line = LineToJump;

atom\_vector.push\_back( atom );

OtherLines();

return;

}

//GOSUB-оператор

if ( look.type == GOSUB )

{

int LineToJump = look.str;

Match( GOSUB );

//Выдать атом ХРАНПЕРЕХ

ATOM atom;

atom.type = A\_STORE\_GOTO;

atom.Line = LineToJump;

atom\_vector.push\_back( atom );

OtherLines();

return;

}

//RETURN-оператор

if ( look.type == RETURN )

{

Match( RETURN );

//Выдать атом ВОЗВРПЕРЕХОД

ATOM atom;

atom.type = A\_RETURN;

atom\_vector.push\_back( atom );

OtherLines();

return;

}

//FOR-оператор

if ( look.type == FOR )

{

int VariableToAssign = look.ptr;

Match( FOR );

relat();

//Выдать атом ПРИСВ

ATOM atom;

atom.type = A\_LET;

atom.VariableIndex = VariableToAssign;

atom.ResultIndex = Constant\_Pool.size() - 1;

atom\_vector.push\_back( atom );

Match( TO );

relat();

//Выдать атом ХРАНИТЬ

ATOM atom1;

atom1.type = A\_STORE;

atom1.ResultIndex = Constant\_Pool.size() - 1;

//Новый элемент таблицы

Constant\_Pool.push\_back( Constant(0,0,0) );

atom1.VariableIndex = Constant\_Pool.size() - 1;

atom\_vector.push\_back( atom1 );

Step();

//Выдать атом ХРАНИТЬ

ATOM atom2;

atom2.type = A\_STORE;

atom2.ResultIndex = Constant\_Pool.size() - 1;

atom2.VariableIndex = Constant\_Pool.size() - 1;

atom\_vector.push\_back( atom2 );

//Выдать атом МЕТКА

ATOM atom3;

atom3.type = A\_LABEL;

atom3.Line = current\_line;

atom\_vector.push\_back( atom3 );

//Выдать атом ПРОВЕРКА

ATOM atom4;

atom4.type = A\_COMPARE;

atom4.VariableIndex = VariableToAssign;

atom4.ResultIndex = atom1.VariableIndex;

atom4.ResultIndex1 = atom2.VariableIndex;

atom4.AtomPointer = &atom3;

atom\_vector.push\_back( atom4 );

OtherLines();

if ( look.type == NEXT )

{

if ( VariableToAssign != look.ptr ) printf("Warning: Variable in FOR is different from the NEXT variable.\n");

//Выдать атом УВЕЛИЧИТЬ

ATOM atom5;

atom5.type = A\_INCREASE;

atom5.VariableIndex = VariableToAssign;

atom5.ResultIndex = atom2.ResultIndex;

atom\_vector.push\_back( atom5 );

}

Match( NEXT );

//Выдать атом ПЕРЕХОД НА

ATOM atom6;

atom6.type = A\_GOTO;

atom6.Line = atom3.Line;

atom\_vector.push\_back( atom6 );

//Выдать атом МЕТКА

ATOM atom7;

atom7.type = A\_LABEL;

atom7.Line = current\_line;

atom\_vector.push\_back( atom7 );

OtherLines();

return;

}

//REM-оператор

if ( look.type == REM )

{

Match( REM );

OtherLines();

return;

}

if ( look.type != NEXT && look.type != END && look.type != EOF )

{

printf("Error: Unexpected lexem.\n");

exit(-1);

}

}

void Programm()

{

if ( look.type != STR )

{

printf("Error: Programm starts incorectly.");

exit(-1);

}

current\_line = look.str;

Match( STR );

//Выдать атом НОМЕРСТРОКИ

ATOM atom;

atom.type = A\_STR\_NUM;

atom.Line = current\_line;

atom\_vector.push\_back( atom );

ProgrammBody();

Match(END);

if ( look.type != END\_OF\_FILE )

{

printf("Error: Programm continues after 'END' keyword.");

exit(-1);

}

Match(END\_OF\_FILE);

}

void Parse( vector<Lex> \*vec )

{

//Инициализация

lexVector = vec;

current = 0;

current\_line = 0;

atom\_vector.clear();

GetLexem();

//Запуск синтаксического анализатора

Programm();

}

#define FILE\_NAME "test.txt"

vector<ATOM> atom\_vector;

Lex look;

vector<Lex> \*lexVector = NULL;

vector<Constant> Constant\_Pool;

//Создадим нашу таблицу идентификаторов, заметим, что нулевой индекс не используется

Constant NameSpace[287];

int current;

//Вывод константы на экран

void PrintConstant(Constant &c)

{

if ( c.Znak ) printf("-");

printf("%dE%d", c.Chislo, c.Poryadok );

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

//Список строк, считаем, что в нашей программе возможно только 1000 строк

TS\_El \_strList[1000];

for(int i = 0; i < 1000; i++)

\_strList[i].s\_begin = NULL;

//Лексический анализатор

vector<Symbol> v\_liters = transliterater("test.txt");

vector<Lex> v\_lexems = lexBlock(v\_liters,NameSpace, \_strList);

//Синтаксический анализаор

Parse( &v\_lexems );

printf("This is a correct programm.\nID Table # 1\n");

//Вывод таблицы констант

for(int i = 0; i < Constant\_Pool.size(); i++)

{

printf("%d: ",i);

PrintConstant( Constant\_Pool.at(i) );

printf("\n");

}

printf("ID Table #2\n");

for(int i = 1; i < 287; i++)

{

//Если ячейка памяти будет использоваться в программе, то выдать её

if ( NameSpace[i].Chislo && NameSpace[i].Poryadok && NameSpace[i].Znak ) continue;

printf("%d: ",i);

PrintConstant( NameSpace[i] );

printf("\n");

}

printf("Atom list\n");

for(int i = 0; i < atom\_vector.size() ;i++)

{

printf("%d:",i);

switch( atom\_vector.at(i).type )

{

case A\_END: printf("END\n"); break;

case A\_STR\_NUM: printf("STR\_NUM\n"); break;

case A\_LET: printf("LET\n"); break;

case A\_GOTO: printf("GOTO\n"); break;

case A\_STORE\_GOTO: printf("STORE\_GOTO\n"); break;

case A\_IF\_GOTO: printf("IF\_GOTO\n"); break;

case A\_RETURN: printf("RETURN\n"); break;

case A\_STORE: printf("STORE\n"); break;

case A\_LABEL: printf("LABEL\n"); break;

case A\_COMPARE: printf("COMPARE\n"); break;

case A\_INCREASE: printf("INCREASE\n"); break;

case A\_ADD: printf("ADD\n"); break;

case A\_SUBTR: printf("SUBTR\n"); break;

case A\_MULT: printf("MULT\n"); break;

case A\_DIV: printf("DIV\n"); break;

case A\_EXP: printf("EXP\n"); break;

case A\_PLUS: printf("PLUS\n"); break;

case A\_MINUS: printf("MINUS\n"); break;

case A\_CONTROL: printf("CONTROL\n"); break;

default: break;

}

}

return 0;

}