Министерство образования и науки РФ

ГОУ ВПО Тамбовский государственный технический университет

Кафедра САПР

**Курсовая работа**

По дисциплине «Лингвистическое программное обеспечение систем

автоматизированного проектирования»

**Тема:** «Разработка трёхпроходного транслятора с исходного языка

на язык С»

Выполнил студент гр.

Проверила: доцент.

Тамбов 20 г.

**Пояснительная записка**

**Задание на проектирование**

Разработать двухпроходный транслятор с исходного языка на язык паскаль. В качестве метода синтаксического анализа использовать метод операторного предшествования.

*Исходный текст программы:*

*программа А;*

*переменные k:целые;*

*t, eps, sx, s:вещественные;*

*начало*

*читать(eps, sx);*

*s=0; k=1; t=1;*

*пока |t| > eps выполнять*

*{*

*k:=k+2;*

*t:=-t\*sx/(k\*(k-1));*

*s:=s+t };*

*вывести(k)*

*конец.*

Дата принятия задания к исполнению 27.02.20

Подпись\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Содержание**

|  |  |
| --- | --- |
| Пояснительная записка |  |
| Задание на проектирование |  |
| Содержание |  |
| Введение |  |
| Описание процесса решения задачи |  |
| Блок-схема основной программы и процедур |  |
| Распечатка программных модулей. |  |
| Описание программы |  |
| Назначение и общее описание программы |  |
| Описание логической структуры программы |  |
| Способ обращения к программе |  |
| Перечень технических средств. |  |
| Описание входных и выходных данных |  |
| Текстовые примеры работы программы |  |

## Введение

В состав любой вычислительной системы может входить комплекс программ, которые называются трансляторами. Транслятор обеспечивает автоматический перевод программ с алгоритмического языка в машинные коды.

По функциональному назначению трансляторы делятся на компиляторы (перевод программ на языке высокого уровня в машинные коды без выполнения), интерпретаторы (перевод каждой конструкции алгоритмического языка в машинные коды с одновременным выполнением) и ассемблеры (перевод программы с языка низкого уровня в машинные коды).

Более подробно остановимся на компиляторах. Компилятор – это не что иное, как программа, написанная на некотором языке, для которой входной информацией служит исходная программа, а результатом является эквивалентная ей объектная программа. Раньше компиляторы писались на автокоде. Часто это был единственно доступный язык. Однако сейчас существует тенденция писать компиляторы на языках высокого уровня, поскольку при этом уменьшается время, затрачиваемое на программирование и отладку, а также обеспечивается удобочитаемость компилятора, когда работа над ним завершена.

Компиляторам присущ ряд общих черт, что упрощает процесс создания компилирующих программ. Наша цель состоит в том, чтобы описать известные уже модельные представления структуры компиляторов и показать, как с их помощью создается работоспособная компилирующая программа.

Компилятор должен выполнить анализ исходной программы и синтез объектного кода. В соответствии с этим любой компилятор включает три основные части: лексический анализатор, синтаксический анализатор и генератор кода.

Взаимодействие между компонентами компилятора может осуществляться разнообразными способами.

В задании требуется, что бы программа обеспечивала перевод текста программы написанного на языке высокого уровня на язык высокого уровня C, а также проверяла исходный текст программы на наличие ошибок.

Курсовая работа содержит текст и описание программы, в которой реализовано задание, описание процесса решения задачи, текстовые примеры работы программы, блок - схемы основных процедур.

Финальная версия программы «Транслятор» написана к 18.04.2014.

## Описание процесса решения задачи

**Грамматика исходного языка**

В качестве формы записи грамматики выбрана форма Бекуса-Наура (БНФ). Грамматика БНФ состоит из множества правил вывода, каждое из которых определяет синтаксис некоторой конструкции языка программирования. Строки символов, заключённые в угловые скобки < и >, называются нетерминальными символами, то есть являются именами конструкций, определенных внутри грамматики. То, что не заключено в угловые скобки, называется терминальными символами (лексемами).

<программа> -> программа <имя программы>;{переменные <раздел переменных>};начало <раздел операторов>;конец.

<имя программы> -> идентификатор

<раздел переменных> → <список переменных> : <тип>

<список переменных> -> идентификатор {, идентификатор}

<тип> -> вещественные/целые

<раздел операторов> -> <оператор>; {оператор;}

<оператор> -> <сложный оператор> / <присваивание> / <цикл> / <вывод> / <ввод>

<сложный оператор> -> { <оператор>{;оператор} }

<присваивание> -> идентификатор :=/= <выражение>

<выражение>-><слагаемое>{+/-<слагаемое>}

<слагаемое>-> идентификатор / константа/(<выражение>)/-<выражение>/|<выражение>|/<выражение>\*//<выражение>

<цикл> -> пока <выражение> выполнять <оператор>

<вывод>-> вывести ( <список переменных> )

<ввод> → читать( <список переменных> )

Работа транслятора начинается с распознавания имени программы и всех переменных, описанных между “переменные” и “:” через запятую. Имя программы и все переменные заносятся в массив. Тут же происходит проверка на некорректное имя программы (не должны содержаться пробелы в имени программы и на повторное описание идентификаторов, выдавая соответствующие ошибки соответственно. При нахождении этих ошибок программа тут же закрывается. Следующий этап, он же основной, это лексический анализатор вместе с синтаксическим анализатором. Рассмотрим этот этап подробнее.

|  |  |
| --- | --- |
| номер | Лексема |
| 1 | программма |
| 2 | переменные |
| 3 | начало |
| 4 | конец |
| 5 | { |
| 6 | } |
| 7 | Пока |
| 8 | выполнять |
| 9 | читать |
| 10 | вывести |
| 11 | = |
| 12 | | |
| 13 | ( |
| 14 | ) |
| 15 | ; |
| 16 | : |
| 17 | , |
| 18 | . |
| 19 | + |
| 20 | - |
| 21 | \* |
| 22 | / |
| 23 | > |
| 24 | Целые |
| 25 | Вещественные |
| 26 | Переменная |
| 27 | Константа |
| 28 | Имя программы |

**Лексический и синтаксический анализ:**

Текст программы записывается в строку из поля codeWindow, вызывается синтаксический анализатор, который в цикле до конца программы или этой строки вызывает лексический анализатор, тот считывает лексемы и записывает в массив stack по одной. После каждой считанной лексемы список передаётся анализатору. Если произошла ошибка, анализатор выдаёт ошибку и синтаксический анализ кода дальше не производится.

В синтаксическом анализаторе используется двумерный массив matrix – описанная ранее матрица. Сравниваются соседние нетерминальные символы этого массива и если между ними символ “ ”, то выдаётся синтаксическая ошибка Был предусмотрен случай неровного количества открывающих и закрывающих скобок. Цепочки терминальных символов, изъятые из стэка, помещаются в массив nts, а на место этих цепочек появляется ссылка на них в этом массиве (нетерминальный символ). Таким образом эмулируется синтаксическое дерево.

Матрица операторного предшествования представлена в программе массивом matrix, имеющим следующий вид:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | программа | переменные | начало | конец | { | } | Пока | выполнять | читать | вывести | = | | | ( | ) | ; | : | , | . | + | - | \* | / | > | Целые | Вещественне | Переменная | Константа | Имя программы |
| программма | < | < | < |  | < |  | < |  | < | < | < | < |  | < |  | < | < | < |  |  |  |  |  |  |  |  | < |  |
| переменные |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | = |  |  |  |  |  |  |  |  |  |  | < |  |
| начало |  |  | > |  |  |  |  |  |  |  |  |  |  |  |  | = | < | < |  |  |  |  |  |  | < | < | < |  |
| конец |  |  |  | = |  |  | < |  | < | < | < | < |  |  |  | = | < |  |  |  |  |  |  |  |  |  | < |  |
| { |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | = |  |  | = |  |  |  |  |  |  |  |  |  |
| } |  |  |  |  |  | = | < |  | < | < | < | < |  |  |  | = | < |  |  |  |  |  |  |  |  |  | < |  |
| Пока |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | > | > |  |  |  |  |  |  |  |  |  |  |  |
| выполнять |  |  |  |  |  |  |  | = |  |  |  | < |  | < |  |  |  |  |  | < | < | < | < | < |  |  | < | < |
| читать |  |  |  |  | < |  | < |  | < | < | < |  |  |  |  | > | < |  |  |  |  |  |  |  |  |  | < |  |
| вывести |  |  |  |  |  |  |  |  |  |  |  |  |  | = |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| = |  |  |  |  |  |  |  |  |  |  |  |  |  | = |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| | |  |  |  |  |  | > |  |  |  |  |  | < |  | < |  | > |  |  |  | < | < | < | < |  |  |  | < | < |
| ( |  |  |  |  |  |  |  |  |  |  |  | = | = | < |  | > |  |  |  | < | < | < | < | > |  |  | < | < |
| ) |  |  |  |  |  |  |  |  |  |  |  |  |  |  | > | > |  |  |  | > | > | > | > | > |  |  |  |  |
| ; |  |  |  |  |  |  |  |  |  |  |  | < |  | < | = |  |  | < |  | < | < | < | < |  |  |  | < | < |
| : |  |  |  | > |  |  |  | > |  |  |  | > |  |  | > | > |  |  |  | > | > | > | > | > |  |  |  |  |
| , |  | = | = | = |  | = | < |  | < | < | < |  |  |  |  | = | < | < |  |  |  |  |  |  | < | < | < |  |
| . |  |  |  |  | < |  |  |  |  |  | = |  |  |  |  |  |  |  |  |  |  |  |  |  | > | > |  |  |
| + |  |  |  |  |  |  |  |  |  |  |  |  |  |  | > |  | = | = |  |  |  |  |  |  |  |  | < | < |
| - |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| \* |  |  |  |  |  | > |  | > |  |  |  | < | > | < | > | > |  |  |  | > | > | < | < | > |  |  | < | < |
| / |  |  |  |  |  | > |  | > |  |  |  | < | > | < | > | > |  |  |  | > | > | < | < | > |  |  | < | < |
| > |  |  |  |  |  | > |  | > |  |  |  | < | > | < | > | > |  |  |  | > | > | > | > | > |  |  | < | < |
| Целые |  |  |  |  |  | > |  | > |  |  |  | < | > | < | > | > |  |  |  | > | > | > | > | > |  |  | < | < |
| Вещественные |  |  |  |  |  |  |  | > |  |  |  | < | > | < | > |  |  |  |  | < | < | < | < |  |  |  | < | < |
| Переменная |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | > |  |  |  |  |  |  |  |  |  |  |  |  |
| Константа |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | > |  |  |  |  |  |  |  |  |  |  |  |  |
| Имя |  |  |  |  |  | > |  | > |  |  | > | > | > |  | > | > | > | > |  | > | > | > | > | > |  |  |  |  |

**Генератор кода:**

При отсутствии ошибок на стадии анализа происходит генерация кода.

Генерация кода осуществляется из синтаксического дерева рекурсивной процедурой codeGen, осуществляющей проход по дереву от корня до листьев, при этом выводя код в выходной поток.

**Блок–схема лексического анализатора**

Начало

Известный идентификатор?

да нет

Идёт объявление переменных?

да нет

Ошибка

Объявить переменную

Запись лексемы в массив

Конец оператора/программы?

Нет да

Возврат в синтаксический анализатор

**Блок-схема синтаксического анализатора**

Начало

Вовращена ошибка?

да нет

Ошибка?

Проверка соседних символов

нет да

Вывести сообщение

Объявить переменную

Установление флага ошибки в true

Конец программы?

Вывод сообщения

Нет да

Были ошибки?

Вывод сгенерированного кода

**Блок-схем генерация кода**

Для всех потомков узла синтаксического дерева

Терминальный символ?

конец

Нет

а

Да

Вывод символа в поле результата

ГГенерация кода для этого узла дерева

Текст программы:

//---------------LEXER.H

#ifndef LEXER\_H

#define LEXER\_H

#include <QDebug>

#include <QTextStream>

#include <qfile.h>

enum LexemType {

T\_START = 0,

T\_PROG,

T\_VARS,

T\_PROG\_BEGIN,

T\_PROG\_END,

T\_BEGIN,

T\_END,

T\_WHILE,

T\_DO,

T\_READ,

T\_WRITE,

T\_SET,

T\_O\_ABS,

T\_C\_ABS,

T\_O\_PAR,

T\_C\_PAR,

T\_SEMICOLON,

T\_COLON,

T\_COMMA,

T\_DOT,

T\_PLUS,

T\_MINUS,

T\_MUL,

T\_DIV,

T\_GT,

T\_INT,

T\_REAL,

T\_VAR,

T\_CONST,

T\_NAME,

THE\_END,

NT

};

//Лексема. Описывается типом(переменная, скобка, константа и тд) и значением (нужно только для переменных и констант)

struct Lexem{

LexemType type;

QString value;

int line;

};

//Проверяет, можно ли пропустить этот символ(чтобы пропускать несколько пробелов подряд)

bool isSkipable(QChar c)

{

return !c.isDigit() && !c.isLetter() && !c.isPunct() && !c.isSymbol();

}

//Проверяет, является ли символ символом-разделителем(тем, что слова друг от друга отделяет)

bool isSeparator(QChar c)

{

return isSkipable(c) || (QString(":\*/+-|{}();.,=").indexOf(c)!=-1);

}

class Lexer {

public:

Lexer(const QString &string){

str = string;

end = str.length();

lexemTypes["программа"]=T\_PROG;

lexemTypes["переменные"]=T\_VARS;

lexemTypes["начало"]=T\_PROG\_BEGIN;

lexemTypes["конец"]=T\_PROG\_END;

lexemTypes["{"]=T\_BEGIN;

lexemTypes["}"]=T\_END;

lexemTypes["пока"]=T\_WHILE;

lexemTypes["выполнять"]=T\_DO;

lexemTypes["читать"]=T\_READ;

lexemTypes["вывести"]=T\_WRITE;

lexemTypes["="]=T\_SET;

lexemTypes["|"]=T\_O\_ABS;

lexemTypes["("]=T\_O\_PAR;

lexemTypes[")"]=T\_C\_PAR;

lexemTypes[";"]=T\_SEMICOLON;

lexemTypes[":"]=T\_COLON;

lexemTypes[","]=T\_COMMA;

lexemTypes["."]=T\_DOT;

lexemTypes["+"]=T\_PLUS;

lexemTypes["-"]=T\_MINUS;

lexemTypes["/"]=T\_DIV;

lexemTypes["\*"]=T\_MUL;

lexemTypes["целые"]=T\_REAL;

lexemTypes["вещественные"]=T\_INT;

lexemTypes[">"] = T\_GT;

}

Lexem getNextLexem()

{

QString buffer = "";

Lexem lexem;

QChar c;

lexem.line = line;

//Один символ считываем в любом случае

if(pos < end){

c = readChar();

buffer+=c;

if(!isSeparator(c)){

//Считываем посимвольно лексему до разделителя

while(pos < end){

c = readChar();

if(isSeparator(c)){

unreadChar();

break;}

buffer+=c;

}

}

} else {

lexem.type = THE\_END;

return lexem;

}

//Пропускаем пробелы и переносы строк

while(pos < end){

c = readChar();

if(!isSkipable(c)){

unreadChar();

break;

}

}

lexem.value = buffer;

lexem.type = getLexemType(buffer);

return lexem;

}

private:

//Считываем один символ из файла

QChar readChar()

{

if(str[pos]=='\n')

line++;

return str[pos++];

}

void unreadChar()

{

/\*if(str[pos]=='\n')

line--;\*/

pos--;

}

// Находим тип лексемы

LexemType getLexemType(const QString &str)

{

// Если он есть в массиве типов, то возвращаем его

if(lexemTypes.contains(str.toLower())){

return lexemTypes[str];

} else {

//если это число, то константа

if(str[0].isDigit())

return T\_CONST;

else

//иначе считаем его переменной

return T\_VAR;

}

}

// Массив типов лексем

QMap<QString, LexemType> lexemTypes;

QString str;

int pos = 0;

int end = 0;

int line = 0;

};

#endif // LEXER\_H

//--------------------SYNTAX.H

#ifndef SYNTAX\_H

#define SYNTAX\_H

#include <qvector.h>

#include "lexer.h"

//Матрица операторного предшествования

char matrix[THE\_END+3][THE\_END+3] = {

// spvbe{} drw=||();:, +-\*/

" <<< < < <<<< < <<< < =", //start

" = < = ", //program

" > =<< <<< ", // peremennye

" = < <<<< =< < ", //begin

" = = ", //end

" =< <<<< =< < ", //{

" >> ", //}

" = < < <<<<< << ", //while

" < < <<< >< < ", //do

" = ", //read

" = ", //write

" > < < > <<<< << ", //=

" ==< > <<<<> << ", //||

" >> >>>>> ", //||

" < <= < <<<< << ", //(

" > > > >> >>>>> ", //)

" === =< <<< =<< <<< ", // ;

" < = >> ", // :

" > == << ", // ,

" >", // .

" > > <><>> >><<> << ", //plus

" > > <><>> >><<> << ", //minus

" > > <><>> >>>>> << ", //mulSS

" > > <><>> >>>>> << ", //div

" > <><> <<<< << ", //gt

" > ",

" > ",

" > > >>> >>>> >>>>> ",

" > >> >> >>>>> ",

" ",

" "

};

int find\_prev\_term(QVector<Lexem> &stack, int pos)

{

while(pos>0){

pos--;

if(stack[pos].type!=NT)

return pos;

}

return -1;

}

// Ищем отношение >

bool is\_closing(QVector<Lexem> &stack, int pos)

{

int ppos = find\_prev\_term(stack, pos);

if(ppos == -1)

return false;

if(matrix[stack[ppos].type][stack[pos].type] == '>')

return true;

return false;

}

// Ищем отношение <

int find\_open\_term(QVector<Lexem> &stack, int pos)

{

int tpos = pos, ppos;

while(true){

ppos = find\_prev\_term(stack, tpos);

if(ppos == -1) return -1;

if(matrix[stack[ppos].type][stack[tpos].type] == '<')

return ppos + 1;

if(matrix[stack[ppos].type][stack[tpos].type] == ' ')

return -1;

tpos = ppos;

}

}

void debug\_vector(QVector<Lexem> &v, QTextStream &out)

{

for(int i = 0; i < v.length(); i++){

out << v[i].value << " ";

}

out << "\n";

}

// роверяем, может ли лексема быть частью выражения

bool is\_operand(Lexem l)

{

return l.type == T\_VAR || l.type == T\_CONST || l.type == NT;

}

// Проверка, оператор ли это

bool is\_operator(Lexem l)

{

return l.type == T\_PLUS || l.type == T\_MINUS || l.type == T\_MUL || l.type == T\_DIV || l.type == T\_GT || l.type == T\_SET;

}

// Все функции, начинающиеся с match - проверка синтаксиса разных выражений

// Проверка правильности операции

bool match\_operation(QVector<Lexem> &chain)

{

// Проверяем унарный минус

if (chain.length() == 2 &&

chain[0].type == T\_MINUS &&

is\_operand(chain[1]))

return true;

if(chain.length() == 3 &&

is\_operand(chain[0]) &&

is\_operator(chain[1]) &&

is\_operand(chain[2]) ){

return true;

} else {

// Список переменных

for(int i = 0; i< chain.length(); i+=2){

if((chain[i].type == T\_VAR || chain[i].type == NT)){

if(i!=0){

if(chain[i-1].type != T\_COMMA)

return false;

}

} else {

return false;

}

}

return true;

}

return false;

}

// Проверяем всю программу

bool match\_program(QVector<Lexem> &chain)

{

if(chain[0].type == T\_PROG && chain[chain.length()-1].type == T\_DOT)

return true;

return false;

}

// Проверяем блок кода (между { и } )"

bool match\_code\_block(QVector<Lexem> &chain)

{

if(chain[0].type == T\_BEGIN && chain[chain.length()-1].type == T\_END)

return true;

return false;

}

//Правильность цикла

bool match\_cycle(QVector<Lexem> &chain)

{

if(chain[0].type == T\_WHILE

&& is\_operand(chain[1])

&& chain[2].type == T\_DO)

return true;

return false;

}

// Модуль

bool match\_abs(QVector<Lexem> &chain)

{

if(chain.length() == 3 &&

chain[0].type == T\_O\_ABS &&

is\_operand(chain[1]) &&

chain[2].type == T\_C\_ABS)

return true;

return false;

}

// Одиночная переменная

bool match\_var(QVector<Lexem> &chain)

{

if(chain.length() == 1 &&

is\_operand(chain[0])) return true;

return false;

}

//Тип переменной

bool match\_vartype(QVector<Lexem> &chain)

{

if(chain.length() == 2 &&

is\_operand(chain[0]) &&

(chain[1].type == T\_REAL || chain[1].type == T\_INT)) return true;

return false;

}

// Читать или вывести

bool match\_function(QVector<Lexem> &chain)

{

if(chain.length() == 2 &&

(chain[0].type == T\_READ || chain[0].type == T\_WRITE ) &&

is\_operand(chain[1]))

return true;

return false;

}

// Удаляем все открытые скобки , точки с запятой и двоеточия из дерева (они только мешают при анализе)

// И проверяем синтаксис, чтобы не было, например, переменная=+, а было переменная=переменная+переменная

bool prepare\_chain(QVector<Lexem> &chain)

{

QVector<Lexem> tmp;

for(int i = 0; i < chain.length(); i++){

if(chain[i].type != T\_O\_PAR &&

chain[i].type != T\_C\_PAR &&

chain[i].type != T\_SEMICOLON &&

chain[i].type != T\_COLON)

tmp.push\_back(chain[i]);

}

if(tmp.length() == 0) return true;

if(match\_var(tmp)) return true;

if(match\_vartype(tmp)) return true;

if(match\_operation(tmp)) return true;

if(match\_abs(tmp)) return true;

if(match\_cycle(tmp)) return true;

if(match\_function(tmp)) return true;

if(match\_code\_block(tmp)) return true;

if(match\_program(tmp)) return true;

return false;

}

// Убираем цепочку лексем из списка

QVector<Lexem> remove\_chain(QVector<Lexem> &stack, int start, int end)

{

QVector<Lexem> chain;

for(int i = start; i < end; i++){

chain.push\_back(stack[i]);

}

stack.remove(start, end-start);

return chain;

}

// Проверка наличия переменной в списке объявленных

bool has\_var(const QVector<Lexem> &vars, Lexem lex)

{

for(int i = 0; i < vars.length(); i++){

if(vars[i].value == lex.value)

return true;

}

return false;

}

//Генерация кода

void genCode(QVector<Lexem> &lex, QVector<QVector<Lexem> > &nts, QTextStream &res)

{

// Чтобы типы переменных норм выводились

if(match\_vartype(lex)){

switch(lex[1].type){

case T\_REAL:

res<<"double ";

break;

case T\_INT:

res << "int ";

break;

default:

break;

}

lex.pop\_back();

}

// Выводим верный скелет программы

if(match\_program(lex)){

res << "int main(int argc, char \*argv[])\n{\n";

lex.remove(0,3);

}

// Общий случай

for(int i = 0; i<lex.length(); i++){

Lexem l = lex[i];

qDebug() << l.value;

switch(l.type){

case NT:

qDebug() << l.line;

genCode(nts[l.line], nts, res);

break;

case T\_PROG:

break;

case T\_VARS:

case T\_START:

case T\_COLON:

case T\_PROG\_BEGIN:

case T\_DOT:

break;

case T\_O\_ABS:

res << "abs(";

break;

case T\_C\_ABS:

case T\_DO:

res << ") ";

break;

case T\_WHILE:

res << "while(";

break;

case T\_SEMICOLON:

res << ";\n";

break;

case T\_READ:

res << "read";

break;

case T\_WRITE:

res << "write";

break;

case T\_PROG\_END:

res << ";\n}";

break;

case T\_BEGIN:

res << "{\n";

break;

case T\_END:

res << ";\n}";

break;

default:

res << l.value;

break;

}

}

}

// Компиляция

int compile(const QString &path, QString \*str, QString \*res)

{

QTextStream out(str);

QTextStream result(res);

Lexer \*lexer = new Lexer(path);

Lexem lex;

QVector<QVector<Lexem> > nts; //Вектор нетерминальных символов

QVector<Lexem> vars; // Список переменных

QVector<Lexem> ints;

QVector<Lexem> reals;

LexemType v\_type = T\_INT;

int line = 0;

int abs\_counter = 0;

int pos = 0, ppos = 0;

bool var\_section = false, prog\_section = false;

Lexem prog\_name;

QVector<Lexem> stack;

QVector<Lexem> chain;

lex.type = T\_START;

stack.push\_back(lex);

// Здесь реализована двухпроходность - читаем по одной лексеме и проводим анализ

while(lex.type != THE\_END){

// Читаем лексему

lex = lexer->getNextLexem();

if(lex.type == T\_O\_ABS){

abs\_counter++;

if(abs\_counter % 2 == 0)

lex.type = T\_C\_ABS;

}

//if(lex.type == T\_INT || lex.type == T\_REAL)

stack.push\_back(lex);

pos++;

line = lex.line;

if(lex.type == T\_PROG)

prog\_section = true;

if(lex.type == T\_VARS){

var\_section = true;

prog\_section = false;

}

if(lex.type == T\_PROG\_BEGIN)

var\_section = false;

// Если это переменная. заносим её в список переменных

if(lex.type == T\_VAR && var\_section)

vars.push\_back(lex);

// Проверка - была ли переменная объявлена

if(lex.type == T\_VAR && !var\_section){

if(!prog\_section) {

if(!has\_var(vars, lex) || lex.value == prog\_name.value){

out << "Error: unknown variable in line " << line << "\n";

return -1;

}

} else {

prog\_name = lex;

}

}

// Проводим синт. анализ

while(true){

debug\_vector(stack, out);

if(!is\_closing(stack, pos)) break;

ppos = find\_open\_term(stack, pos);

if(ppos < 0) {

out << "Syntax error in line " << line << "\n";

return -1;

}

// Удаляем цепочку лексем

chain = remove\_chain(stack, ppos, pos);

if(!prepare\_chain(chain)){

out << "Syntax error in line " << line << "\n";

return -1;

}

nts.push\_back(chain);

// Заменяем на обозначение нетерминального символа

Lexem tlex;

tlex.type = NT;

tlex.line=nts.length() - 1 ;

tlex.value=QString("<N%1>").arg(nts.length() - 1);

stack.insert(ppos, tlex);

pos = stack.length()-1;

}

}

debug\_vector(stack, out);

// Финальная проверка, что от тела программы остаётся один нетерминальный символ

if(stack.length() == 3 &&

stack[0].type == T\_START &&

stack[1].type == NT &&

stack[2].type == THE\_END){

// Генерация кода

genCode(stack, nts, result);

return 1;

} else {

out << "Syntax error in line " << line << "\n";

return -1;

}

return 1;

}

#endif // SYNTAX\_H

**Описание программы:**

**1. Описание технических средств**

Программа была разработана на IBM совместимой машине с процессором AMD Athlon™ 64 X2 Dual, 2.20 GHz на языке C++ (QTCREATOR) под управлением ОС Microsoft Windows Vista Home Basic Service Pack 1. Особых системных требований не имеет. Предполагается работа программы на любом IBM совместимом компьютере под управлением Windows 95 / 98 / Me / Xp / Vista / Windows 7. Запуск в других системах не возможен.

**2. Описание входа**

Входной представляет собой программу на структурно языке высокого уровня. Вывод конечного результата работы анализатора осуществляется на экран.

**3. Способ обращения к программе**

Для запуска программы необходимо выполнить следующий порядок действий:

* Запустить на выполнение файл Project.exe.
* В поле «Исходный код” ввести исходный текст.
* В поле «Результат» будет записан сгенерированный код.

**4. Описание логической структуры программы**

Программа состоит из одного логического модуля.

**5. Назначение и общее описание программы**

Данная программа предназначена для перевода кода с исходного языка на язык C Программа реализована на основе двухпроходного транслятора.

На вход лексическому анализатору, поступает цепочка отдельных терминальных символов, которые группируются им в единые синтаксические объекты – лексемы. Цепочка лексем представляет собой два массива чисел, объединённых в структуру. После распознания лексемы, исследуется цепочка лексем и устанавливается, принадлежит ли она структурным объектам заданного языка. Если проверка прошла успешно, то генерируется код.

Синтаксический и лексический анализаторы распознают следующие типы ошибок:

* Лексические ошибки (связанные с неправильным написанием идентификатора / неверным его обозначением)
* Синтаксические ошибки (связанные с использованием неверных синтаксических конструкций)
* Неизвестные конструкции (неверное написание ключевого слова)
* Двойное описание идентификатора

При обнаружении ошибки выводится сообщение и работа анализатора прекращается

Процесс работы синтаксического анализатора не скрыт от пользователя и на выходе ему доступен не только сгенерированный код, но и последовательность синтаксического разбора. При ошибках пользователю выводятся соответствующие сообщения.

**Текстовые примеры работы программы:**

* **Программа с правильным исходным кодом:**

*программа А;*

*переменные k:целые;*

*t, eps, sx, s:вещественные;*

*начало*

*читать(eps, sx);*

*s=0; k=1; t=1;*

*пока |t| > eps выполнять*

*{*

*k:=k+2;*

*t:=-t\*sx/(k\*(k-1));*

*s:=s+t };*

*вывести(k)*

*конец.*

*Сгенерированный код:*

int main(int argc, char \*argv[])

{

double k;

int t,eps,sx,s;

read(eps,sx);

s=0;

k=1;

t=1;

while(abs(t) >eps) {

k=k+2;

t=-t\*sx/(k\*(k-1));

s=s+t;

};

write(k);

}

* **Программа с добавленными операторами и идентификаторами:**

*программа А;*

*переменные k:целые;*

*t, eps, sx, s,spx:вещественные;*

*начало*

*читать(eps, sx, spx);*

*s=0; k=1; t=1;*

*пока |t+spx| > eps выполнять*

*{*

*k:=k+2+spx;*

*t:=-t\*sx/(k\*(k-1));*

*s:=s+t };*

*вывести(k);*

*вывести(spx);*

*конец.*

*Сгенерированный код:*

int main(int argc, char \*argv[])

{

double k;

int t,eps,sx,s,spx;

read(eps,sx,spx);

s=0;

k=1;

t=1;

while(abs(t+spx) >eps) {

k=k+2+spx;

t=-t\*sx/(k\*(k-1));

s=s+t;

};

write(k);

write(spx);

}

* **Программа с лексической ошибкой**

программа А;

переменные k:целые;

t, eps, sx:вещественные;

начало

читать(eps, sx);

s=0; k=1; t=1;

пока |t| > eps выполнять

{

k:=k+2;

t:=-t\*sx/(k\*(k-1));

s:=s+t };

вывести(k)

конец.

В результате анализа выдано следующее сообщение: "Error: unknown variable in line 6".

* **Программа с синтаксической ошибкой**

программа А;

переменные k:целые;

t, eps, sx,s:вещественные;

начало

читать(eps, sx);

s=0; k=1; 3=1;

пока |t| > eps выполнять

{

k:=k+2;

t:=-t\*sx/(k\*(k-1));

s:=s+t };

вывести(k)

конец.

В результате работы программы выдано сообщение: " Syntax error in line 6".