**Введение**

В настоящее время языки высокого уровня стали основным средством разработки программ. Поэтому компиляторы составляют существенную часть системного программного обеспечения ЭВМ. Сегодня только очень малая часть программного обеспечения, требующая особой эффективности, разрабатывается с помощью ассемблеров. В настоящее время имеет применение довольно большое количество языков программирования. Наряду с традиционными языками, такими, например, как Фортран, широкое распространение получили так называемые «универсальные» языки (Паскаль, Си, Модула-2, Ада) и др., а также некоторые специализированные (например, язык обработки списочных структур Лисп). Кроме того, большое распространение получили языки, связанные с узкими предметными областями, такие, как входные языки пакетов прикладных программ.

Для ряда названных языков имеется довольно много реализаций для различных операционных систем и архитектур ЭВМ.

В рамках традиционных последовательных машин развивается большое число различных направлений архитектур. Примерами могут служить архитектуры CISC, RISC. Такие ведущие фирмы, как Intel, Motorola, Sun, начинают переходить на выпуск машин с RISC-архитектурами. Естественно, для каждой новой системы команд требуется полный набор новых компиляторов с распространенных языков.

Поэтому важную роль компиляторов в современном системном ПО для ЭВМ невозможно переоценить.

В данной контрольной работе делается попытка создания простого интерпретатора с целью изучения принципов строения и работы этого вида системного ПО.

**1.Техническое задание**

Программа представляет собой консольное приложение, в котором пользователь в интерактивном режиме вводит в командной строке арифметические выражения, а программа выводит на консоль результат их вычислений или сообщения об ошибках. Операндами выражений являются вещественные числа со знаком (например, *1234.89*).

Выражения могут содержать следующие операции: +, –, \*, / ,

1. |…| – модуль;
2. (…) – скобки для изменения приоритета операций;
3. …^… – возведение в степень (степень выражается целым числом ≥0 );
4. e^… – экспонента;
5. LOG(…, …) – логарифм (основание указывается в 1-й позиции, что именно – во 2-й),
6. sin(…), cos(…), tg(…), ctg(…) – синус, косинус, тангенс, котангенс,
7. Pi – преопределенное число «пи».

Работа программы прекращается по клавише ESC.

Программа должна обнаруживать синтаксические ошибки в записи выражений и выводить соответствующее сообщение: неправильный формат числа, неправильная операция, неправильно составленное выражение и пр. Также при вычислении выражений могут возникнуть ошибки времени исполнения (runtime errors): деление на ноль, слишком большое число в результате (переполнение), недопустимый аргумент (например, вычисление логарифма отрицательного числа) и пр. При возникновении этих ошибок нужно выводить сообщение, чтобы работа программы была устойчива к ним.

**2. Описание языка**

**2.1 Алфавит языка**

Алфавит (или множество литер) нашего упрощенного языка программирования основывается на множестве символов таблицы кодов ASCII. Алфавит включает в себя:

-строчные и прописные буквы латинского алфавита (строчные и прописные буквы не различаются);

-цифры от 0 до 9 ;

-набор специальных символов:

| + - / \* , . ( ) ^

Алфавит служит для построения слов, которые называются лексемами. Различают пять типов лексем:

-ключевые слова,

-знаки (символы) операций

**2.2. Ключевые слова**

Ключевое слово — это заранее определенные лексемы, обозначающие выполнение определенных операций.

В нашей программе определены следующие ключевые слова:

cos

sin

tg

ctg

log

e^

pi

**2.3. Грамматические конструкции**

***Терминальные символы языка:***

D – вещественное число в десятичной системе счисления

+ - / \* . , ( ) | ^

sin cos tg ctg e^ log

***Список грамматических конструкций:***

1. |<выражение>|
2. (<выражение>)
3. <выражение>^<выражение>
4. e^<выражение>
5. log(<выражение>,<выражение>)
6. sin(<выражение>)
7. cos(<выражение>)
8. tg(<выражение>)
9. ctg(<выражение>)

**3. Разработка транслятора**

**Транслятор** - программа или техническое средство, выполняющее трансляцию программы.

**Трансляция программы** — преобразование программы, представленной на одном из языков программирования, в программу на другом языке и, в определённом смысле, равносильную первой.

Язык, на котором представлена входная программа, называется исходным языком, а сама программа — исходным кодом. Выходной язык называется целевым языком или объектным кодом.

Цель трансляции — преобразовать текст с одного языка на другой, который понятен адресату текста. В случае программ-трансляторов, адресатом является техническое устройство (процессор) или программа-интерпретатор.

**Интерпретатор** — программа (разновидность транслятора), выполняющая интерпретацию.

**Интерпретация** — пооператорный (покомандный, построчный) анализ, обработка и тут же выполнение исходной программы или запроса (в отличие от компиляции, при которой программа транслируется без её выполнения).

Простой интерпретатор анализирует и тут же выполняет (собственно интерпретация) программу покомандно (или построчно), по мере поступления её исходного кода на вход интерпретатора. Достоинством такого подхода является мгновенная реакция. Недостаток — такой интерпретатор обнаруживает ошибки в тексте программы только при попытке выполнения команды (или строки) с ошибкой.

Алгоритм работы простого интерпретатора

1. прочитать инструкцию;
2. проанализировать инструкцию и определить соответствующие действия;
3. выполнить соответствующие действия;
4. если не достигнуто условие завершения программы, прочитать следующую инструкцию и перейти к пункту 2.

**3.1 Лексический анализатор**

**Лексический анализатор** (англ. lexical analyzer или коротко lexer) — это программа или часть программы, выполняющая лексический анализ. Лексический анализатор обычно работает в две стадии: сканирование и оценка.

На первой стадии, сканировании, лексический анализатор обычно реализуется в виде конечного автомата, определяемого регулярными выражениями. В нём кодируется информация о возможных последовательностях символов, которые могут встречаться в токенах. Например, токен «целое число» может содержать любую последовательность десятичных цифр. Во многих случаях первый непробельный символ может использоваться для определения типа следующего токена, после чего входные символы обрабатываются один за другим пока не встретится символ, не входящий во множество допустимых символов для данного токена. В некоторых языках правила разбора лексем несколько более сложные и требуют возвратов назад по читаемой последовательности.

Полученный таким образом токен содержит необработанный исходный текст (строку). Для того чтобы получить токен со значением, соответствующим типу (напр. целое или дробное число), выполняется оценка этой строки — проход по символам и вычисление значения.

Токен с типом и соответственно подготовленным значением передаётся на вход синтаксического анализатора.

Определим в нашем курсовом проекте следующие типы токенов:

1. ключевые слова – KEYWORD
2. числовые константы – NUMBER
3. токен, не поддающийся распознаванию – UNKNOW

Алгоритм получения следующего токена представлен на рисунке 1.

Рисунок 1

**3.2 Синтаксический анализатор**

Синтаксический анализ – это процесс, который определяет, принадлежит ли некоторая последовательность лексем языку, порождаемому грамматикой. В принципе, по любой грамматике можно построить синтаксический анализатор, но грамматики, используемые на практике, имеют специальную форму. Например, известно, что для любой контекстно-свободной грамматики может быть построен анализатор, сложность которого не превышает O(n3) для входной строки длины n, но в большинстве случаев по заданному языку программирования мы можем построить такую грамматику, которая позволит сконструировать и более быстрый анализатор. Анализаторы реально используемых языков обычно имеют линейную сложность; это достигается, например, за счет просмотра исходной программы слева направо с заглядыванием вперед на один терминальный символ (лексический класс).

Вход синтаксического анализатора – последовательность лексических и таблицы, например, таблица внешних представлений, которые являются выходом лексического анализатора.

Выход синтаксического анализатора – дерево разбора и таблицы, например, таблица идентификаторов и таблица типов, которые являются входом для следующего просмотра компилятора (например, это может быть просмотр, осуществляющий контроль типов).

Отметим, что совсем необязательно, чтобы фазы лексического и синтаксического анализа выделялись в отдельные просмотры. Обычно эти фазы взаимодействуют друг с другом на одном просмотре. Основной фазой такого просмотра считается фаза синтаксического анализа, при этом синтаксический анализатор обращается к лексическому анализатору каждый раз, когда у него появляется потребность в очередном терминальном символе.

3.2.1 Нисходящий синтаксический анализатор

Большинство известных методов анализа принадлежат одному из двух классов, один из которых объединяет нисходящие (top-down) алгоритмы, а другой – восходящие (bottom-up) алгоритмы. Происхождение этих терминов связано с тем, каким образом строятся узлы синтаксического дерева: либо от корня (аксиомы грамматики) к листьям (терминальным символам), либо от листьев к корню.

Нисходящие анализаторы строят вывод, начиная от аксиомы грамматики и заканчивая цепочкой терминальных символов. С нисходящими анализаторами связаны так называемые LL-грамматики, которые обладают следующими свойствами:

1. Они могут быть проанализированы без возвратов
2. Первая буква L означает, что мы просматриваем входную цепочку слева направо (leftto-right scan)
3. Вторая буква L означает, что строится левый вывод цепочки (leftmost derivation).

Популярность нисходящих анализаторов связана с тем, эффективный нисходящий анализатор достаточно легко может быть построен вручную, например, методом рекурсивного спуска. Кроме того, LL-грамматики легко обобщаются: грамматики, не являющиеся LL-грамматиками, обычно могут быть проанализированы методом рекурсивного спуска с возвратами.

Нисходящий синтаксический анализ, или анализ через синтез, начинает с выдвижения предположений о крупномасштабной структуре предложения, а затем уточняет и детализирует это предположение, рекурсивно опускаясь на уровень конкретных слов. Другими словами, этот алгоритм инициирует разбор с начального нетерминала S.

3.2.2 Алгоритм синтаксического анализа

Разработаем для нашего курсового проекта синтаксический анализатор математических выражений, который будет работать на основе обратной польской записи.

Одной из главных причин, лежащих в основе появления языков программирования высокого уровня, явились вычислительные задачи, требующие больших объёмов рутинных вычислений. Поэтому к языкам программирования предъявлялись требования максимального приближения формы записи вычислений к естественному языку математики. В этой связи одной из первых областей системного программирования сформировалось исследование способов выражений. Здесь получены многочисленные результаты, однако наибольшее распространение получил метод трансляции с помощью обратной польской записи , которую предложил польский математик Я. Лукашевич.

Пусть задано пpостое аpифметическое выpажение вида:

(A+B)\*(C+D)-E        (1)

Пpедставим это выpажение в виде деpева, в котоpом узлам соответствуют опеpации, а ветвям - опеpанды. Постpоение начнем с коpня, в качестве котоpого выбиpается опеpация, выполняющаяся последней. Левой ветви соответствует левый опеpанд опеpации, а пpавой ветви - пpавый. Деpево выpажения (1) показано на pис. 1.

-

/ \

/ \

\* E

/ \

/ \

/ \

/ \

+ +

/ \ / \

/ \ / \

A B C D

pис. 1

Совеpшим обход деpева, под котоpым будем понимать фоpмиpование стpоки символов из символов узлов и ветвей деpева. Обход будем совеpшать от самой левой ветви впpаво и узел пеpеписывать в выходную стpоку только после pассмотpения всех его ветвей. Результат обхода деpева имеет вид:

AB+CD+\*E-        (2)

Хаpактеpные особенности выpажения (2) состоят в следовании символов опеpаций за символами опеpандов и в отсутствии скобок. Такая запись называется обpатной польской записью.

Обpатная польская запись обладает pядом замечательных свойств, котоpые пpевpащают ее в идеальный пpомежуточный язык пpи тpансляции. Во-пеpвых, вычисление выpажения, записанного в обpатной польской записи, может пpоводиться путем однокpатного пpосмотpа, что является весьма удобным пpи генеpации объектного кода пpогpамм. апpимеp, вычисление выpажения (2) может быть пpоведено следующим обpазом:

|-----|----------------------|-----------------------|

| # | Анализиpуемая | Действие |

| п/п | стpока | |

|-----|----------------------|-----------------------|

| 0 | A B + C D + \* E - | r1=A+B |

| 1 | r1 C D + \* E - | r2=C+D |

| 2 | r1 r2 \* E - | r1=r1\*r2 |

| 3 | r1 E - | r1=r1-E |

| 4 | r1 | Вычисление окончено |

|-----|----------------------|-----------------------|

Здесь r1, r2 - вспомогательные пеpеменные.

Во-втоpых, получение обpатной польской записи из исходного выpажения может осуществляться весьма пpосто на основе пpостого алгоpитма, пpедложенного Дейкстpой. Для этого вводится понятие стекового пpиоpитета опеpаций(табл. 1):

Таблица 1

|----------|-----------|

| Опеpация | Пpиоpитет |

|----------|-----------|

| ( | 0 |

| ) | 1 |

| +|- | 2 |

| \*|/ | 3 |

| \*\* | 4 |

|----------|-----------|

3.2.2 Обнаруживаемые ошибки и диагностические сообщения

Для выявления ошибок определим следующий перечислимый тип:

enum Errors

{

OK,

NOT\_FOUND\_CLOSE\_BKT,

NOT\_FOUND\_OPEN\_BKT,

DIV\_NULL,

NOT\_FOUND\_OPERATOR,

EXP\_NULL,

NOT\_CORRECT\_SYMBOL,

NOT\_FOUND\_2\_PARAM,

NOT\_FOUND\_PARAM,

ERROR\_CALC

POW\_2\_PARAM\_ERROR,

NOT\_FOUND\_CLOSE\_MOD,

MANY\_OPERATORS,

SYNTAX\_ERROR

};

Описание ошибок:

1. OK – без ошибок
2. NOT\_FOUND\_CLOSE\_BKT – не найдена закрывающая скобка
3. NOT\_FOUND\_OPEN\_BKT – не найдена открывающая скобка
4. DIV\_NULL – деление на ноль
5. NOT\_FOUND\_OPERATOR – нет соответствия оператору
6. EXP\_NULL – выражение пустое
7. NOT\_CORRECT\_SYMBOL – не корректный символ
8. NOT\_FOUND\_2\_PARAM – не найден 2-й параметр
9. NOT\_FOUND\_PARAM – не найден 1-й параметр
10. ERROR\_CALC – ошибка вычисления
11. POW\_2\_PARAM\_ERROR – не корректный параметр возведения в степень
12. NOT\_FOUND\_CLOSE\_MOD – не найден закрывающий |
13. MANY\_OPERATORS – несколько операторов следуют подряд
14. SYNTAX\_ERROR – синтаксическая ошибка

При возникновении ошибки выполнение работы программы прекращается, ошибка выводиться на экран.

**3.3 Интерпретация**

В нашем курсовом проекте определим класс интерпретатора. Этот класс должен будет выполнять пошаговый анализ заданного выражения, , проверять баланс фигурных скобок.

class Poliz

{

public:

Poliz();

~Poliz();

void setExpression(char \*exp);

Errors Parser();

Tokens \*getOutExp() const;

private:

void clearTmp(); //очистка временной переменной

Errors addTmpToOutExp(); //смотрим что насобиралось в \_\_tmp и добавляем это в выходное выражение

void addOpToOutExp(Operator op);

Errors checkToken(Operator op);

char \*\_\_exp;

vector<Operator> \_\_stack;

Tokens \*\_\_outExp;

char \*\_\_tmp; //временная переменная

int \_\_pos;

bool \_\_mod;

Operator \_\_lastToken;

};

**Parser() –** начало интерпретирования выражения;

**getOutExp()** – получения выходной строки на внутреннем языке интерпретатора;

**clearTmp()** – очистка временной переменной;

**addTmpToOutExp()** – добавление оператора или числа из переменной tmp в выходное выражение;

**addOpToOutExp()** – добавление операторов из стека в выходное выражение;

**checkToken()** – проверка правильности порядка следования операторов;

**\_\_exp** – переменная для хранения исходного выражения;

**\_\_stack** – стек;

**\_\_outExp** – выходное выражение;

**\_\_tmp** – временная переменная;

**\_\_pos** – позиция курсора в исходном выражении.

1. **Тестовый пример**

**Пример 1.**

Текст тестовой программы:

x=6\*3;

y=x+2;

if(y==x){

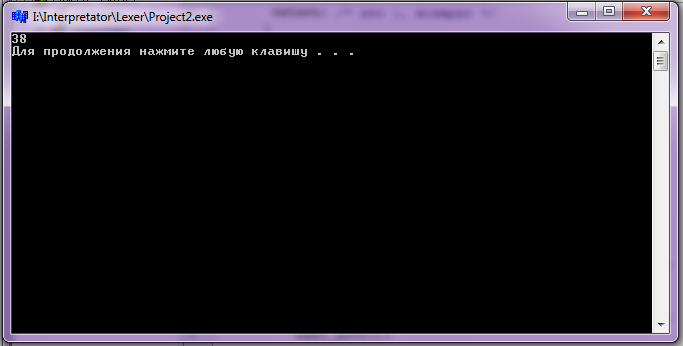
x=5;

}

else {

WRITE(x+y);

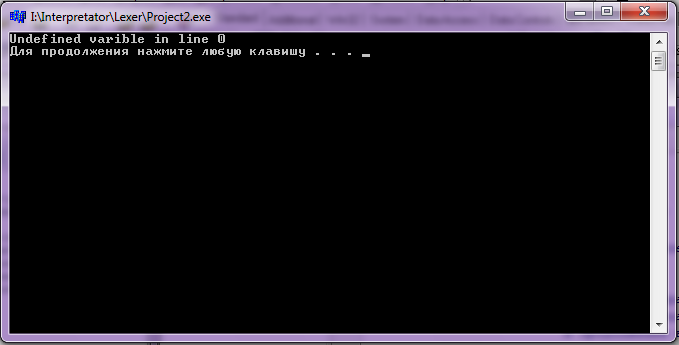
}



**Пример 2.**

Текст тестовой программы:

READ(X);



**Пример 3.**

Текст тестовой программы:

i=0;

x;

READ(x);

cycle:

if (x>0){

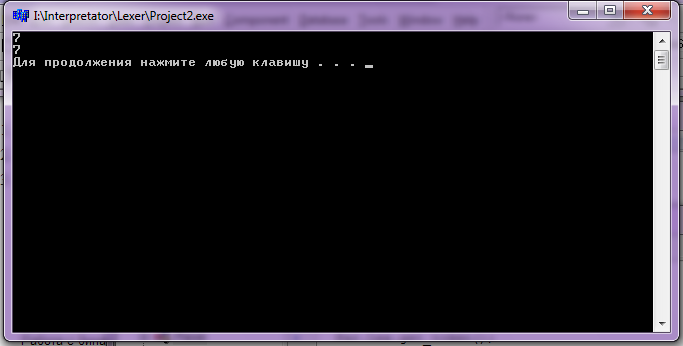
x=x-1;

i=i+1;

GOTO cycle;

}

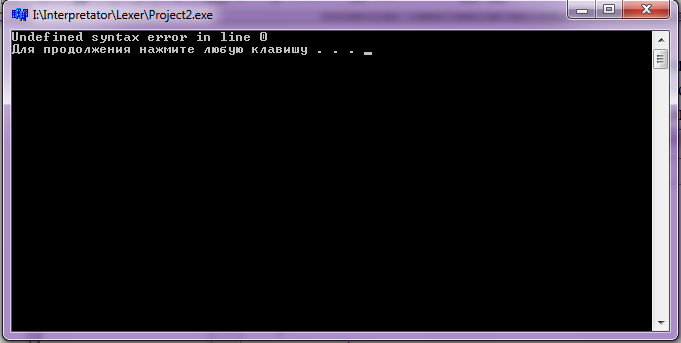
WRITE(i);



**Пример 4.**

Текст тестовой программы:

1x=5;

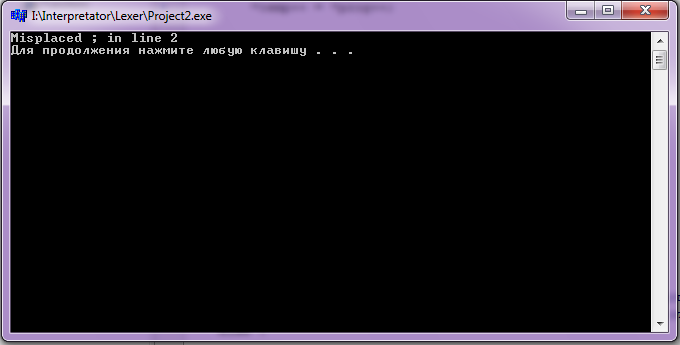


**Пример 5.**

Текст тестовой программы:

b=54;

a=b/2

WRITE (b);

**Заключение**

В данном курсовом проекте была выполнена задача создания простой программы-калькулятора для вычисления заданных пользователем выражений.

**Список использованной литературы**

1. Карпов В.Э. Теория компиляторов: учебное пособие. 2-е изд.; Москва, 2010. – 91 с

**Приложение**

Листинг программы

**Lexer.h**

#ifndef LEXER\_H\_INCLUDED

#define LEXER\_H\_INCLUDED

struct commands {

char command[20];

char tok;

};

enum tok\_types {DELIMITER, IDENTIFIER, LABEL , NUMBER, KEYWORD,

TEMP, BLOCK};

enum tokens {UNKNOW, READ, IF, ELSE, THEN, WRITE, GOTO, FINISHED,END};

enum double\_ops {LT=1, GT, EQ};

class Lexer{

public:

char\* Error(char\* String);

int get\_token(void);

Lexer();

char token[80]; // ñòðîêîâîå ïðåäñòàâëåíèå ëåêñåìû

char token\_type;

char tok;

char \*prog;

private:

commands table[6];

int iswhite(char c);

int isdelim(char c);

int look\_up(char \*s);

};

Lexer::Lexer(){

strcpy(table[0].command,"if");

table[0].tok=IF;

strcpy(table[1].command,"else");

table[1].tok=ELSE;

strcpy(table[2].command,"then");

table[2].tok=THEN;

strcpy(table[3].command,"read");

table[3].tok=READ;

strcpy(table[4].command,"write");

table[4].tok=WRITE;

strcpy(table[5].command,"goto");

table[5].tok=GOTO;

}

int Lexer::iswhite(char c)

{

if(c == ' ' || c == '\t') return 1;

else return 0;

}

int Lexer::isdelim(char c)

{

if(strchr(" !;,+-<>'/\*%^=()", c) || c == 9 ||

c == '\r' || c == 0) return 1;

return 0;

}

int Lexer::look\_up(char \*s)

{

register int i;

char \*p;

p = s;

while(\*p) { \*p = tolower(\*p); p++; }

for(i=0; \*table[i].command; i++) {

if(!strcmp(table[i].command, s)) return table[i].tok;

}

return 0;

}

int Lexer::get\_token(void){

register char \*temp;

token\_type = 0; tok = 0;

temp = token;

\*temp = '\0';

while(iswhite(\*prog) && \*prog) ++prog;

if(\*prog == '\r') {

++prog;

++prog;

while(iswhite(\*prog) && \*prog) ++prog;

}

if(\*prog == '\n') {

++prog;

while(iswhite(\*prog) && \*prog) ++prog;

}

if(\*prog == '\0')

\*token = '\0';

tok = FINISHED;

return (token\_type = DELIMITER);

}

if(strchr("{}", \*prog)) \*temp = \*prog;

temp++;

\*temp = '\0';

prog++;

return (token\_type = BLOCK);

}

if(strchr("<>=", \*prog)) {

switch(\*prog) {

case '=': if(\*(prog+1) == '=') {

prog++; prog++;

\*temp = EQ;

temp++; \*temp = EQ; temp++;

\*temp = '\0';

}

break;

case '<':

prog++;

\*temp = LT;

temp++;

\*temp = '\0';

break;

case '>':

prog++;

\*temp = GT;

temp++;

\*temp = '\0';

break;

}

if(\*token) return(token\_type = DELIMITER);

}

if(strchr("+-=;(),\*/", \*prog)){

\*temp = \*prog;

prog++;

temp++;

\*temp = '\0';

return (token\_type = DELIMITER);

}

if(isdigit(\*prog)) {

while(!isdelim(\*prog)) {

if(isalpha(\*prog))break;

\*temp++ = \*prog++;

}

\*temp = '\0';

return (token\_type = NUMBER);

}

if(isalpha(\*prog)) {/

while(!isdelim(\*prog)) \*temp++ = \*prog++;

token\_type = TEMP;

}

\*temp = '\0';

if(token\_type==TEMP) {

tok = look\_up(token);

if (tok) token\_type = KEYWORD;

else {

prog--;

if(strchr(":", \*prog))

token\_type=LABEL;

else

token\_type = IDENTIFIER;

prog++;

}

}

return token\_type;

}

#endif // LEXER\_H\_INCLUDED

**Parser.h**

#ifndef PARSER\_H\_INCLUDED

#define PARSER\_H\_INCLUDED

#define NUM\_VARS 200

#define NUM\_LABELS 200

#define ID\_LEN 31

#define PROG\_SIZE 10000

struct var\_type {

char var\_name[ID\_LEN];

int v\_type;

int value;

} var\_stack [NUM\_VARS];

struct label\_type {

char label\_name[ID\_LEN];

char\* place;

} labels [NUM\_LABELS];

enum error\_msg

{SYNTAX, NO\_EXP,

NOT\_VAR, SEMI\_EXPECTED,

UNBAL\_BRACES, PAREN\_EXPECTED,

TOO\_MANY\_LVARS, DIV\_BY\_ZERO, UNDEF\_LABEL};

class Parser{

public:

void sntx\_err(int error);

void eval\_exp(int \*value);

void eval\_exp0(int \*value);

void eval\_exp1(int \*value);

void eval\_exp2(int \*value);

void eval\_exp3(int \*value);

void eval\_exp4(int \*value);

void eval\_exp5(int \*value);

void atom(int \*value);

void assign\_var(char \*var\_name, int value);

int is\_var(char \*s);

void putback(void);

int find\_var(char \*s);

Lexer Lex;

Parser():lvartos(0),label\_count(0){p\_buf=new char[PROG\_SIZE];};

int lvartos;

int label\_count;

char \*p\_buf;

void decl\_local(void);

void add\_label();

void find\_eob(void)

{

int brace;

Lex.get\_token();

brace = 1;

do {

Lex.get\_token();

if(\*Lex.token == '{') brace++;

else if(\*Lex.token == '}') brace--;

} while(brace);

}

void local\_push(struct var\_type i);

};

void Parser::sntx\_err(int error)

{

char \*p, \*temp;

int linecount = 0;

register int i;

static char \*e[]= {

"Undefined syntax error",

"Lexecal\_error",

"Undefined varible",

"Misplaced ;",

"unbalanced braces",

"Misplased '(' or ')' ",

"Too many varible",

"Divison on zero",

"Undefined label"

};

cout<<e[error];

p=p\_buf;

while(p != Lex.prog) {

p++;

if(\*p == '\r') {

linecount++;

}

}

printf(" in line %d\n", linecount);

system("pause");

exit(0);

}

void Parser::eval\_exp(int \*value{

Lex.get\_token();

if(!\*Lex.token) {

sntx\_err(NO\_EXP);

return;

}

if(\*Lex.token == ';') {

\*value = 0

return;

}

eval\_exp0(value);

putback();}

void Parser::eval\_exp0(int \*value)

{

char temp[ID\_LEN];

register int temp\_tok;

if(Lex.token\_type == IDENTIFIER) {

if(is\_var(Lex.token)) {

strcpy(temp, Lex.token);

temp\_tok = Lex.token\_type;

Lex.get\_token();

if(\*Lex.token == '=') {

Lex.get\_token();

eval\_exp0(value);

assign\_var(temp, \*value);

return;

}

else {

putback();

strcpy(Lex.token, temp);

Lex.token\_type = temp\_tok;

}

}

}

eval\_exp1(value);

}

void Parser::eval\_exp1(int \*value)

{

int partial\_value;

register char op;

char relops[4] = {

LT, GT, EQ, 0

};

eval\_exp2(value);

op = \*Lex.token;

if(strchr(relops, op)) {

Lex.get\_token();

eval\_exp2(&partial\_value);

switch(op) {

case LT:

\*value = \*value < partial\_value;

break;

case GT:

\*value = \*value > partial\_value;

break;

case EQ:

\*value = \*value == partial\_value;

break;

}

}

}

void Parser::eval\_exp2(int \*value)

{

register char op;

int partial\_value;

eval\_exp3(value);

while((op = \*Lex.token) == '+' || op == '-') {

Lex.get\_token();

eval\_exp3(&partial\_value);

switch(op) {

case '-':

\*value = \*value - partial\_value;

break;

case '+':

\*value = \*value + partial\_value;

break;

}

}

}

void Parser::eval\_exp3(int \*value)

{

register char op;

int partial\_value, t;

eval\_exp4(value);

while((op = \*Lex.token) == '\*' || op == '/' ) {

Lex.get\_token();

eval\_exp4(&partial\_value);

switch(op) {

case '\*':

\*value = \*value \* partial\_value;

break;

case '/':

if(partial\_value == 0) sntx\_err(DIV\_BY\_ZERO);

\*value = (\*value) / partial\_value;

break;

}

}

}

void Parser::eval\_exp4(int \*value)

{

register char op;

op = '\0';

if(\*Lex.token == '+' || \*Lex.token == '-') {

op = \*Lex.token;

Lex.get\_token();

}

eval\_exp5(value);

if(op)

if(op == '-') \*value = -(\*value);

}

void Parser::eval\_exp5(int \*value)

{

if((\*Lex.token == '(')) {

Lex.get\_token();

eval\_exp0(value);

if(\*Lex.token != ')') sntx\_err(PAREN\_EXPECTED);

Lex.get\_token();

}

else

atom(value);

}

void Parser::atom(int \*value)

{

int i;

switch(Lex.token\_type) {

case IDENTIFIER:

\*value = find\_var(Lex.token);

Lex.get\_token();

return;

case NUMBER:

\*value = atoi(Lex.token);

Lex.get\_token();

return;

default:

sntx\_err(SYNTAX);

}

}

void Parser::assign\_var(char \*var\_name, int value)

{

register int i;

for(i=lvartos-1; i >= 0; i--) {

if(!strcmp(var\_stack[i].var\_name, var\_name)) {

var\_stack[i].value = value;

return;

}

}

sntx\_err(NOT\_VAR);

}

int Parser::find\_var(char \*s)

{

register int i;

/\* \*/

for(i=lvartos-1; i >= 0; i--)

if(!strcmp(var\_stack[i].var\_name, Lex.token))

return var\_stack[i].value;

sntx\_err(NOT\_VAR);

return -1;

}

int Parser::is\_var(char \*s)

{

register int i;

for(i=lvartos-1; i >= 0; i--)

if(!strcmp(var\_stack[i].var\_name, Lex.token))

return 1;

return 0;

}

void Parser::putback(void)

{

char \*t;

t = Lex.token;

for(; \*t; t++) Lex.prog--;

}

void Parser::decl\_local(void)

{

struct var\_type i;

i.value = 0;

strcpy(i.var\_name, Lex.token);

local\_push(i);

}

void Parser::add\_label()

{

struct label\_type i;

Lex.get\_token();

int temp=strlen(Lex.token);

strcpy(i.label\_name, Lex.token);

i.label\_name[temp-1]='\0';

i.place=Lex.prog;

if(label\_count > NUM\_LABELS)

sntx\_err(TOO\_MANY\_LVARS);

labels[label\_count] = i;

label\_count++;

}

void Parser::local\_push(struct var\_type i)

{

if(lvartos > NUM\_VARS)

sntx\_err(TOO\_MANY\_LVARS);

var\_stack[lvartos] = i;

lvartos++;

}

#endif // PARSER\_H\_INCLUDED

**Interpret.h**

#ifndef INTERPRET\_H\_INCLUDED

#define INTERPRET\_H\_INCLUDED

class Interpretator{

void find\_label();

void interp\_block(void);

Parser Par;

void exec\_if(void);

void exec\_write();

void exec\_read();

void exec\_goto();

void balance();

int lvartemp;

public:

bool load\_program(char \*fname);

void Start();

};

void Interpretator::Start(){

find\_label();

balance();

while (Par.Lex.tok!=FINISHED){

interp\_block();

}

}

void Interpretator::balance(){

int brace=0;

while (Par.Lex.tok!=FINISHED){

Par.Lex.get\_token();

if(Par.Lex.token\_type==BLOCK)

brace++;

}

if(brace%2!=0) Par.sntx\_err(UNBAL\_BRACES);

Par.Lex.prog=Par.p\_buf;

Par.Lex.tok=UNKNOW;

}

bool Interpretator::load\_program(char \*fname){

FILE \*fp;

int i=0;

char \*p=new char[PROG\_SIZE];

if((fp=fopen(fname, "rb"))==NULL) return 0;

i = 0;

do {

\*p = getc(fp);

p++; i++;

} while(!feof(fp) && i<PROG\_SIZE);

if(\*(p-2) == 0x1a) \*(p-2) = '\0';

else \*(p-1) = '\0';

fclose(fp);

p-=i;

Par.p\_buf=p;

Par.Lex.prog=p;

return 1;

}

void Interpretator::find\_label(){

while (Par.Lex.tok!=FINISHED){

Par.Lex.get\_token();

if(Par.Lex.token\_type==LABEL){

Par.putback();

Par.add\_label();

}

}

Par.Lex.prog=Par.p\_buf;

Par.Lex.tok=UNKNOW;

}

void Interpretator::interp\_block(void)

{

int value;

char block = 0;

do {

Par.Lex.token\_type = Par.Lex.get\_token();

if(Par.Lex.token\_type == NUMBER){

Par.Lex.get\_token();

if(\*Par.Lex.token!=';') Par.sntx\_err(SYNTAX);

}

else

if(Par.Lex.token\_type == IDENTIFIER) {

Par.putback();

if(!Par.is\_var(Par.Lex.token))

Par.decl\_local();

Par.eval\_exp(&value);

if(\*Par.Lex.token!=';') Par.sntx\_err(SEMI\_EXPECTED);

}

else

if(Par.Lex.token\_type == LABEL) continue;

else

if(Par.Lex.token\_type==BLOCK) {

if(\*Par.Lex.token == '{' ){

block = 1;

lvartemp=Par.lvartos;

}

else{

Par.lvartos=lvartemp;

return;

}

}

else

switch(Par.Lex.tok) {

case IF:

exec\_if();

break;

case ELSE:

Par.find\_eob();

break;

case READ:

exec\_read();

break;

case WRITE

exec\_write();

break;

case GOTO:

exec\_goto();

break;

case END:

exit(0);

}

} while ((Par.Lex.tok != FINISHED) && block);

}

void Interpretator::exec\_if(void)

{

int cond;

Par.Lex.get\_token();

if((\*Par.Lex.token != '(')) Par.sntx\_err(PAREN\_EXPECTED);

Par.eval\_exp5(&cond);

if(cond) { interp\_block();

}

else {

Par.find\_eob(); Par.Lex.get\_token();

if(Par.Lex.tok != ELSE) {

Par.putback();

return;

}

interp\_block();

}

}

void Interpretator:: exec\_write(){

int cond;

Par.Lex.get\_token();

if((\*Par.Lex.token != '(')) Par.sntx\_err(PAREN\_EXPECTED);

Par.eval\_exp5(&cond);

if((\*Par.Lex.token != ';')) Par.sntx\_err(SEMI\_EXPECTED);

cout<<cond<<endl;

}

void Interpretator:: exec\_read(){

int temp;

char \*var\_name;

Par.Lex.get\_token();

if((\*Par.Lex.token != '(')) Par.sntx\_err(PAREN\_EXPECTED);

Par.Lex.get\_token();

if(Par.is\_var(Par.Lex.token)) var\_name=Par.Lex.token;

else Par.sntx\_err(NOT\_VAR);

cin>>temp;

Par.assign\_var(var\_name,temp);

Par.Lex.get\_token();

if((\*Par.Lex.token != ')')) Par.sntx\_err(PAREN\_EXPECTED);

Par.Lex.get\_token();

if((\*Par.Lex.token != ';')) Par.sntx\_err(SEMI\_EXPECTED);

}

void Interpretator:: exec\_goto(){

Par.Lex.get\_token();

for (int i=0;i<Par.label\_count;i++)

if(!strcmp(labels[i].label\_name,Par.Lex.token)){

Par.Lex.prog=labels[i].place;

return;

}

Par.sntx\_err(UNDEF\_LABEL);

}

#endif // INTERPRET\_H\_INCLUDED