

Элементы теории трансляции

Лексический анализ

В основе лексических анализаторов лежат регулярные грамматики.

Соглашения:

- для описания лексем будем использовать леволинейную автоматную грамматику без пустых правых частей.

Грамматика $G = (T, N, P, S)$ автоматная *леволинейная*, если каждое правило из P имеет вид

$A \rightarrow Bt$ либо

$A \rightarrow t$, где $A \in N$, $B \in N$, $t \in T$.

- анализируемая цепочка заканчивается специальным символом \perp - *признаком конца цепочки*.

Алгоритм разбора по левосторонней грамматике.

- (1) первый символ исходной цепочки $a_1a_2\dots a_n\perp$ заменяем нетерминалом A , для которого в грамматике есть правило вывода

$$A \rightarrow a_1$$

(производим "свертку" терминала a_1 к нетерминалу A)

- (2) многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующее: полученный на предыдущем шаге нетерминал A и расположенный непосредственно справа от него очередной терминал a_i исходной цепочки заменяем нетерминалом B , для которого в грамматике есть правило вывода

$$B \rightarrow Aa_i \quad (i = 2, 3, \dots, n)$$

Если в результате работы алгоритма прочитана **вся цепочка**, на каждом шаге находилась **единственная нужная «свертка»**, а на последнем шаге произошла **свертка к символу S** , значит цепочка $a_1a_2\dots a_n\perp$ **принадлежит языку**.

Если на каком-то шаге **не нашлось нужной свертки** или на последнем шаге произошла свертка к символу, **отличному от S** , значит цепочка $a_1a_2\dots a_n\perp$ **не принадлежит языку**.

Если на каком-то шаге нашлось более одной подходящей свертки, значит разбор по заданной грамматике **недетерминированный**.

Диаграммы состояний (ДС)

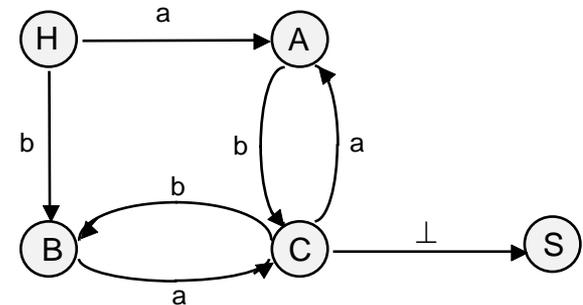
Диаграмма состояний (ДС) - неупорядоченный ориентированный помеченный граф, который строится следующим образом:

- (1) строим вершины графа, помеченные нетерминалами грамматики (для каждого нетерминала - одну вершину), и еще одну вершину, помеченную символом, отличным от нетерминальных (например, H). Эти вершины называются **состояниями**. **H - начальное состояние**.
- (2) соединяем эти состояния дугами по следующим правилам:
 - а) для каждого правила вида $W \rightarrow t$ соединяем дугой состояния H и W (от H к W) и помечаем дугу символом t;
 - б) для каждого правила вида $W \rightarrow Vt$ соединяем дугой состояния V и W (от V к W) и помечаем дугу символом t.

Диаграмма состояний для грамматики $G_{ex} = (\{ a, b, \perp \}, \{ S, A, B, C \}, P, S)$, где

P:
 $S \rightarrow C\perp$
 $C \rightarrow Ab \mid Ba$
 $A \rightarrow a \mid Ca$
 $B \rightarrow b \mid Cb$

будет выглядеть так:



$L(G) = \{ ([ab \mid ba])^n \perp \mid n \geq 1 \}$

Алгоритм разбора по диаграмме состояний

(1) объявляем текущим состояние N ;

(2) многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующее: считываем очередной символ исходной цепочки и переходим из текущего состояния в другое состояние по дуге, помеченной этим символом. Состояние, в которое мы при этом попадаем, становится текущим.

- Если в результате работы алгоритма прочитана **вся цепочка**, на каждом шаге находилась **единственная дуга**, помеченная очередным символом анализируемой цепочки, а последний переход произошел в **состояние S** , значит исходная цепочка **принадлежит языку**.
- Если на каком-то шаге **не нашлось дуги**, выходящей из текущего состояния и помеченной очередным анализируемым символом или на последнем шаге произошел переход в состояние, **отличное от S** , значит исходная цепочка **не принадлежит языку**.
- Если на каком-то шаге работы алгоритма нашлось несколько дуг, выходящих из текущего состояния, помеченных анализируемым символом, но ведущих в разные состояния, значит разбор по заданной грамматике **недетерминированный**.

Диаграммы состояний для праволинейных грамматик

ДС по праволинейной регулярной грамматике строится следующим образом:

- (1) строим вершины графа, помеченные нетерминалами грамматики, и еще одну вершину, помеченную символом, отличным от нетерминальных (например, F).

Эти вершины называются *состояниями*. S - начальное состояние.
F – заключительное состояние.

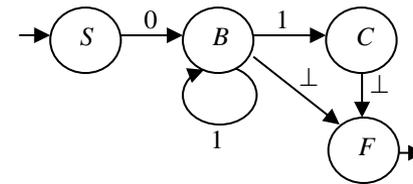
- (2) соединяем эти состояния дугами по следующим правилам:

- а) для каждого правила вида $W \rightarrow t$ соединяем дугой состояния F и W (от W к F) и помечаем дугу символом t;
- б) для каждого правила вида $W \rightarrow tV$ соединяем дугой состояния V и W (от W к V) и помечаем дугу символом t;

Диаграмма состояний для грамматики $G_{ex2} = (\{0, 1, \perp\}, \{S, B, C\}, P, S)$, где

P:
 $S \rightarrow 0B$
 $B \rightarrow 1B \mid 1C \mid \perp$
 $C \rightarrow \perp$

будет выглядеть так:



$L(G) = \{0 1^n \perp \mid n \geq 0\}$

Имея ДС, построенную по любой регулярной грамматике, можно по ней восстановить как левوليнейную, так и эквивалентную ей праволинейную регулярную грамматику.

Алгоритм построения праволинейной грамматики G_{right} по ДС, построенной по леволинейной грамматике G_{left}

- Нетерминалами G_{right} будут все состояния из ДС, кроме S .
- Если в ДС есть исходящие дуги из S , то вводим в ДС новое заключительное состояние S' , в которое из S ведет дуга, помеченная знаком конца \perp .
- Каждой дуге из состояния V в заключительное состояние S (или S'), помеченной символом t (\perp) ставится в соответствие правило $V \rightarrow t$ ($V \rightarrow \perp$).
- Каждой дуге из состояния V в состояние W , помеченной символом t , ставится в соответствие правило $V \rightarrow tW$.
- Начальное состояние H объявляется начальным символом грамматики.

Алгоритм построения левосторонней грамматики G_{left} по ДС, построенной по правосторонней грамматике G_{right}

- Нетерминалами G_{left} будут все состояния из ДС для G_{right} кроме S .
- Каждой дуге из состояния S в состояние W , помеченной символом t , ставится в соответствие правило $W \rightarrow t$. Если в ДС есть входящие дуги в S , то в G_{left} добавляется также и правило $W \rightarrow St$.
- Каждой дуге из состояния V в состояние W , помеченной символом t , ставится в соответствие правило $W \rightarrow Vt$.
- Заключительное состояние F объявляется начальным символом грамматики.

Соглашения для работы с диаграммами состояний

- a) Если из одного состояния в другое выходит несколько дуг, помеченных разными символами, то изображается одна дуга, помеченная всеми этими символами.
- b) Непомеченная никакими символами дуга соответствует переходу при любом символе, кроме тех, которыми помечены другие дуги, выходящие из этого состояния.
- c) Вводится дополнительное состояние ошибки (ER); переход в это состояние означает, что исходная цепочка языку не принадлежит.

Конечный автомат (КА)

Конечный автомат (КА) - это пятерка (K, T, δ, H, S) , где

K - конечное множество состояний;

T - конечное множество допустимых входных символов;

δ - отображение (функция переходов) множества $K \times T \rightarrow K$, определяющее поведение автомата;

$H \in K$ - начальное состояние;

$S \in K$ - заключительное состояние (либо конечное множество заключительных состояний, но для грамматик – одно!).

$\delta(A, t) = B$ означает, что из состояния A по входному символу t происходит переход в состояние B .

Конечный автомат **допускает цепочку** $a_1a_2...a_n$, если

$\delta(H, a_1) = A_1; \delta(A_1, a_2) = A_2; \dots; \delta(A_{n-2}, a_{n-1}) = A_{n-1}; \delta(A_{n-1}, a_n) = S$,

где $a_i \in T, A_j \in K, j = 1, 2, \dots, n-1; i = 1, 2, \dots, n$;

H - начальное состояние, S - одно из заключительных состояний.

Множество цепочек, допускаемых конечным автоматом, составляет определяемый им **ЯЗЫК**.

Анализатор для регулярной грамматики G_ex.

```
char c;
void gc ( ) { cin >> c; }

bool scan_G ( ) {
    enum state { H, A, B, C, S, ER };
    state CS; // CS - текущее состояние
    CS = H;
    do { gc ();
        switch (CS) {
            case H:  if (c == 'a') { CS = A;}
                    else if (c == 'b') { CS = B;}
                    else CS = ER;
                    break;
            case A:  if (c == 'b') { CS = C;}
                    else CS = ER;
                    break;
            case B:  if (c == 'a') { CS = C;}
                    else CS = ER;
                    break;
            case C:  if (c == 'a') { CS = A;}
                    else if (c == 'b') { CS = B;}
                    else if (c == '⊥') CS = S;
                    else CS = ER;
                    break;
        }
    } while (CS != S && CS != ER);
    if (CS == ER) return false;
    else return true;
}
```

О недетерминированном разборе

При анализе по ДС для регулярной грамматики может оказаться, что из одного состояния выходит несколько дуг, ведущих в разные состояния, но помеченных одним и тем же символом.

Для левوليнейных грамматик эта ситуация возникает, когда в правилах вывода есть **совпадающие правые части**.

Для праволинейных грамматик аналогичная ситуация возникает, когда альтернативы вывода из одного нетерминала грамматики начинаются **одинаковыми терминальными символами**.

Недетерминированный конечный автомат (НКА) – это пятерка

(K, T, δ, H, S) , где

K - конечное множество состояний;

T - конечное множество допустимых входных символов;

δ - отображение множества $K \times T$ в множество подмножеств K ;

$H \subset K$ - конечное множество начальных состояний (у нас одно!);

$S \subset K$ - конечное множество заключительных состояний (у нас одно!).

$\delta(A, t) = \{B_1, B_2, \dots, B_n\}$ означает, что из состояния A по входному символу t можно осуществить переход в любое из состояний B_i , $i = 1, 2, \dots, n$.

Алгоритм построения детерминированного КА по НКА

Вход: $M = (K, T, \delta, H, S)$ - НКА.

Выход: $M1 = (K1, T, \delta1, H1, S1)$ - детерминированный КА, допускающий тот же язык, что и автомат M .

Метод:

1. Строим отображение $\delta1 (K1 \times T \rightarrow K1)$ по δ , начиная с состояния H .
2. $\delta1$ для $H \subset K1$ строим т.о.:
 - если в НКА переход из H по какому-то символу был детерминированным, переносим соответствующее отображение в результирующий КА. Состояния, появляющиеся в правой части отображений $\delta1$ принадлежат $K1$;
 - если же в НКА переход из H по какому-то символу t был недетерминированным, то в КА включаем отображения $\delta1$ по правилу:
 $\delta1 (H, t) = \underline{A1A2...An}$, где $Ai \subset K$, и в НКА есть $F (H, t) = Ai$ для всех $1 \leq i \leq n$.

Состояние $A1A2...An$ $\subset K1$.

3. Для всех состояний, появившихся в правой части отображений $\delta1$ результирующего КА, определяем отображения $\delta1$ т.о.:
 $\delta1 (\underline{A1A2...An}, t) = \underline{B1B2...Bm}$, где для каждого $1 \leq j \leq m$ в НКА существует $\delta (Ai, t) = Bj$ для каких-либо $1 \leq i \leq n$.
4. Заключительными состояниями построенного детерминированного КА являются все состояния, содержащие $S \subset K$ в своем имени. Для написания грамматики по КА все эти состояния **необходимо свести к одному заключительному состоянию $S1 \subset K1$ с помощью символа \perp** .

Пример работы алгоритма преобразования НКА в КА

Задан НКА $M = (\{ H, A, B, S \}, \{ 0, 1 \}, \delta, \{ H \}, \{ S \})$, где

$$\delta(H, 1) = B$$

$$L = \{ 1(01)^n \mid n \geq 1 \}$$

$$\delta(A, 1) = B$$

$$\delta(A, 1) = S$$

$$\delta(B, 0) = A ,$$

Функции переходов КА, построенные по НКА в соответствии с предложенным алгоритмом.

$$\delta_1(H, 1) = B$$

$$\delta_1(B, 0) = A$$

$$\delta_1(A, 1) = \underline{BS}$$

$$\delta_1(\underline{BS}, 0) = A$$

Заключительным состоянием построенного детерминированного КА является состояние BS.

Таким образом, $M_1 = (\{H, B, A, \underline{BS}\}, \{0, 1\}, \delta_1, H, \underline{BS})$.

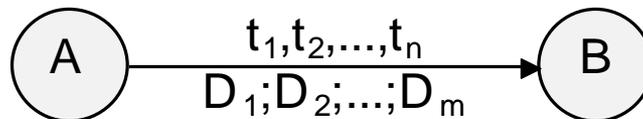
Задачи лексического анализа

1. выделить в исходном тексте цепочку символов, представляющую лексему, и проверить правильность ее записи;
2. удалить пробельные литеры и комментарии;
3. преобразовать цепочку символов, представляющих лексему, в пару:

(тип_лексемы, указатель_на_информацию_о_ней);
4. зафиксировать в специальных таблицах для хранения разных типов лексем факт появления соответствующих лексем в анализируемом тексте.

Чтобы решить эту задачу, опираясь на способ анализа с помощью ДС, на дугах вводятся дополнительные пометки-действия.

Теперь каждая дуга в ДС может выглядеть так:



Лексический анализ языков программирования

Принято выделять следующие типы лексем: **идентификаторы, служебные слова, константы и ограничители.**

Каждой лексеме сопоставляется пара:

(тип_лексемы, указатель_на_информацию_о_ней).

Таким образом, лексический анализатор - это транслятор, входом которого служит цепочка символов, представляющих исходную программу, а выходом - последовательность лексем.

Лексемы перечисленных выше типов можно описать с помощью регулярных грамматик. Например,

идентификатор (I):

$$I \rightarrow a | b | \dots | z | Ia | Ib | \dots | Iz | I0 | I1 | \dots | I9$$

целое без знака (N):

$$N \rightarrow 0 | 1 | \dots | 9 | N0 | N1 | \dots | N9$$

Описание модельного языка

$P \rightarrow$ **program** D1; B \perp
 $D1 \rightarrow$ **var** D { , D }
 $D \rightarrow$ I { , I } : [**int** | **bool**]
 $B \rightarrow$ **begin** S { ; S } **end**
 $S \rightarrow$ I := E | **if** E **then** S **else** S | **while** E **do** S | B | **read** (I) | **write** (E)
 $E \rightarrow$ E1 [= | < | > | <= | >= | !=] E1 | E1
 $E1 \rightarrow$ T { [+ | - | *or*] T }
 $T \rightarrow$ F { [* | / | *and*] F }
 $F \rightarrow$ I | N | L | *not* F | (E)
 $L \rightarrow$ **true** | **false**
 $I \rightarrow$ a | b | ... | z | Ia | Ib | ... | Iz | I0 | I1 | ... | I9
 $N \rightarrow$ 0 | 1 | ... | 9 | N0 | N1 | ... | N9

Замечания:

- запись вида $\{\alpha\}$ означает итерацию цепочки α , т.е. в порождаемой цепочке в этом месте может находиться либо ε , либо α , либо $\alpha\alpha$, либо $\alpha\alpha\alpha$ и т.д.
- запись вида $[\alpha | \beta]$ означает, что в порождаемой цепочке в этом месте может находиться либо α , либо β .
- **P** - цель грамматики; символ \perp - маркер конца текста программы.

Контекстные условия.

- Любое имя, используемое в программе, должно быть описано и только один раз.
- В операторе присваивания типы переменной и выражения должны совпадать.
- В условном операторе и в операторе цикла в качестве условия возможно только логическое выражение.
- Операнды операции отношения должны быть целочисленными.
- Тип выражения и совместимость типов операндов в выражении определяются по обычным правилам; старшинство операций задано синтаксисом.
- В любом месте программы, кроме идентификаторов, служебных слов и чисел, может находиться произвольное число пробелов и комментариев вида { < любые символы, кроме '}', '{' и '⊥' > }.
- **Program, var, int, bool, begin, end, if, then, else, while, do, true, false, read и write** - служебные слова (их нельзя переопределять).
- Используется паскалевское правило о разделителях между идентификаторами, числами и служебными словами.

Представление лексем

```
enum type_of_lex {
```

```
    LEX_NULL, /*0*/
```

```
    LEX_AND, LEX_BEGIN, ... LEX_WRITE, /*18*/
```

```
    LEX_FIN, /*19*/
```

```
    LEX_SEMICOLON, LEX_COMMA, ... LEX_GEQ, /*35*/
```

```
    LEX_NUM, /*36*/
```

```
    LEX_ID, /*37*/
```

```
    POLIZ_LABEL, /*38*/
```

```
    POLIZ_ADDRESS, /*39*/
```

```
    POLIZ_GO, /*40*/
```

```
    POLIZ_FGO }; /*41*/
```

Содержательно внутреннее представление лексем - это пара
(тип_лексема, значение_лексема).

Значение лексема - это номер строки в таблице лексем соответствующего класса, содержащей информацию о лексеме, или непосредственное значение, например, в случае целых чисел.

Соглашение об используемых таблицах лексем:

TW - таблица служебных слов М-языка;

TD - таблица ограничителей М-языка;

TID - таблица идентификаторов анализируемой программы;

Класс Lex

```
class Lex {
    type_of_lex  t_lex;
    int  v_lex;
public:
    Lex ( type_of_lex  t = LEX_NULL, int  v = 0) {
        t_lex = t;
        v_lex = v;
    }
    type_of_lex  get_type ( ) {
        return t_lex;
    }
    int  get_value ( ) {
        return v_lex;
    }
    friend ostream& operator << (ostream & s, Lex l ) {
        s << '(' << l.t_lex << ',' << l.v_lex << ");" ;
        return s;
    }
};
```

Класс Ident

```
class Ident {
    char *name;
    bool declare;
    type_of_lex type;
    bool assign;
    int value;
public:
    Ident ( ) { declare = false; assign = false; }
    char *get_name ( ) { return name; }
    void put_name ( const char * n ) {
        name = new char [ strlen ( n ) + 1];
        strcpy ( name, n );
    }
    bool get_declare ( ) { return declare; }
    void put_declare ( ) { declare = true; }
    type_of_lex get_type ( ) { return type; }
    void put_type ( kind_of_lex t ) { type = t; }
    bool get_assign ( ) { return assign; }
    void put_assign ( ) { assign = true; }
    int get_value ( ) { return value; }
    void put_value ( int v ) { value = v; }
};
```

Класс `tabl_ident`

```
class tabl_ident{
    ident *p;
    int size;
    int top;
public:
    tabl_ident ( int max_size ) {
        p = new ident [ size = max_size ];
        top = 1;
    }
    ~tabl_ident ( ) { delete [ ] p; }
    ident & operator [ ] ( int k ) { return p [ k ]; }
    int put (const char * buf);
};

int tabl_ident::put(const char * buf) {
    for (int j = 1; j < top; j++)
        if ( ! strcmp ( buf,p [ j ].get_name ( ) ) ) return j;
    p [ top ].put_name ( buf );
    top++;
    return top - 1;
}
```

Класс Scanner

```
class Scanner {
    enum state { H, IDENT, NUMB, COM, ALE, DELIM, NEQ };
    static char* TW [ ];
    static type_of_lex words [ ];
    static char * TD [ ];
    static type_of_lex dlms [ ];
    state CS;
    FILE *fp;
    char c;
    char buf [ 80 ];
    int buf_top;
    void clear ( ) ;
    void add ( );
    int look (const char * buf, char * * list);
    void gc ( ) { c = fgetc ( fp ); }
public:
    Scanner (const char * program);
    Lex get_lex ();
};
```

Класс Scanner

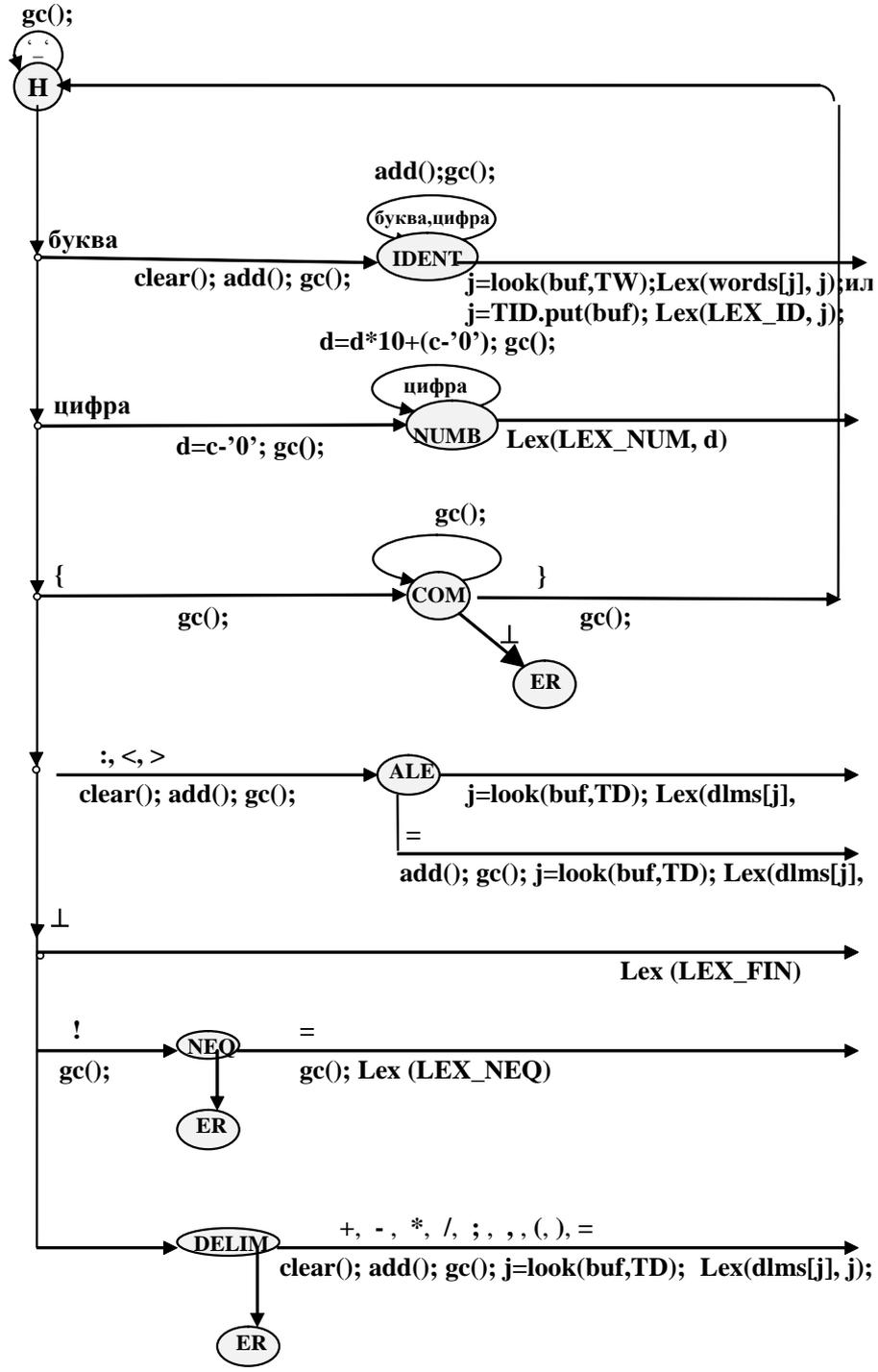
```
void Scanner::clear ( ) {
    buf_top = 0;
    for (int j = 0; j < 80; j++ )
        buf [ j ] = '\0';
}
void Scanner::add ( ) {
    buf [ buf_top ++ ] = c;
}
int Scanner::look (const char * buf, char * * list) {
    int i = 0;
    while (list [ i ]) {
        if ( ! strcmp (buf, list [ i ]) )
            return i;
        i++;
    }
    return 0;
}
Scanner::Scanner (const char * program) {
    fp = fopen ( program, "r" );
    CS = H;
    clear();
}
```

Таблицы лексем для М-языка

```
char * Scanner:: TW [ ] =
{ NULL, "and", "begin", "bool", "do", "else", "end",
//      1      2      3      4      5      6
  "if", "false", "int", "not", "or", "program", "read",
// 7      8      9      10     11     12     13
  "then", "true", "var", "while", "write" };
//      14     15     16     17     18

char * Scanner:: TD [ ] = {NULL, ";", "@", ",", ":", ":=", "(", ")" ,
//      1      2      3      4      5      6      7
  "=", "<", ">", "+", "-", "*", "/", "<=", ">="};
//      8      9     10     11     12     13     14     15     16

tabl_ident TID (100);
```



```
Lex Scanner::get_lex ( ) {
```

```
int d, j;
```

```
CS = H;
```

```
do { gc ();
```

```
switch(CS) {
```

```
case H:
```

```
if ( c == ' ' || c == '\n' || c == '\r' || c == '\t' )
```

```
else
```

```
if ( isalpha(c) ) { clear(); add(); CS = IDENT; }
```

```
else
```

```
if ( isdigit (c) ) { d = c - '0'; CS = NUMB; }
```

```
else
```

```
if ( c == '{' ) { CS = COM; }
```

```
else
```

```
if ( c == ':' || c == '<' || c == '>' ) {
```

```
clear(); add(); CS = ALE; }
```

```
else
```

```
if ( c == '@' ) return Lex (LEX_FIN);
```

```
else
```

```
if ( c == '!' ) { clear(); add (); CS = NEQ; }
```

```
else CS = DELIM;
```

```
break;
```

case IDENT:

```
    if ( isalpha(c) || isdigit(c) ) {  
        add(); }  
    else  
    if ( j = look (buf, TW) )  
        return Lex (words[ j ], j);  
    else {  
        j = TID.put (buf);  
        return Lex (LEX_ID, j);  
    }  
    break;
```

case NUMB:

```
    if ( isdigit (c) ) {  
        d = d * 10 + (c - '0'); }  
    else  
        return Lex ( LEX_NUM, d);  
    break;
```

case COM:

```
    if ( c == '}' ) { CS = H; }  
    else  
    if (c == '@' || c == '{' ) throw c;  
    break;
```

```
case ALE:
```

```
    if ( c == '=' ) {        add();
                            j = look ( buf, TD );
                            return Lex ( dlms [ j ], j);
    }
    else {    j = look (buf, TD);
            return Lex ( dlms [ j ], j);    }
    break;
```

```
case NEQ:
```

```
    if ( c == '=' ) {
        add(); j = look ( buf, TD );
        return Lex ( LEX_NEQ, j ); }
    else throw '!';
    break;
```

```
case DELIM:
```

```
    clear( ); add( );
    if ( j = look(buf, TD) ) {
        return Lex ( dlms [ j ], j ); }
    else throw c;
    break;
```

```
    } //end switch
```

```
    } while ( true );
```

```
    }
```

```
}
```