Элементы теории трансляции

Лексический анализ

В основе лексических анализаторов лежат регулярные грамматики.

Соглашения:

- для описания лексем будем использовать **<u>леволинейную автоматную грамматику</u> без пустых правых частей**.

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

  $A \to Bt$ либо

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

- анализируемая цепочка заканчивается специальным символом $\perp$ - *признаком конца цепочки (символ-терминатор)*.

Алгоритм разбора по леволинейной грамматике.

(1) первый символ исходной цепочки $a_1 a_2 ... a_n \perp$ заменяем нетерминалом A, для которого в грамматике есть правило вывода

$$A \rightarrow a_1$$

(производим "свертку" терминала a_1 к нетерминалу A)

(2) пока не считаем признак конца цепочки выполняем следующее: полученный на предыдущем шаге нетерминал **A** и расположенный непосредственно справа от него очередной терминал $\mathbf{a_i}$ исходной цепочки заменяем нетерминалом **B**, для которого в грамматике есть правило вывода

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

Алгоритм разбора по леволинейной грамматике.

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 $a_1a_2...a_n\perp$ **принадлежит языку**.

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

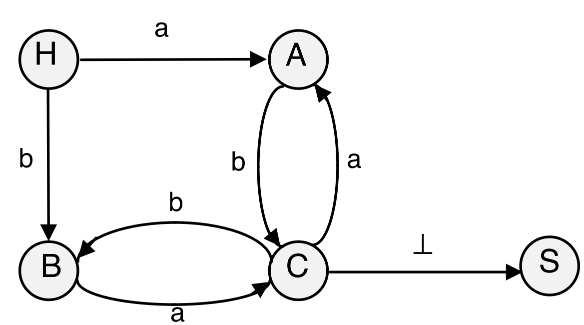
а) для каждого правила вида $W \to t$ соединяем дугой состояния H и W (от H к W) и помечаем дугу символом t;

б) для каждого правила вида $W \to Vt$ 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: $S \to C\perp$
$C \to Ab \mid Ba$
$A \to a \mid Ca$
$B \to b \mid Cb$

будет выглядеть так:



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

Алгоритм разбора по диаграмме состояний

(1) объявляем текущим состояние H;

(2) 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

(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:

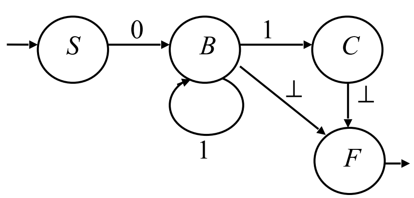
а) для каждого правила вида $W \to t$ соединяем дугой состояния F и W (от W к F) и помечаем дугу символом t;

б) для каждого правила вида $W \to t\,V$ соединяем дугой состояния V и W (от W к V) и помечаем дугу символом t;

Диаграмма состояний для грамматики G_ex2 = ({ 0, 1, ⊥ }, { S, B, C }, P, S), где

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

будет выглядеть так:



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

Теорема

Имея ДС, построенную по любой регулярной грамматике, можно по ней восстановить как леволинейную, 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 признаком конца ⊥.
- Каждой дуге из состояния *V* в заключительное состояние *S* (или S'), помеченной символом *t* (⊥) 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 леволинейной грамматики G_{rleft} по ДС, построенной по праволинейной грамматике G_{right}.

- Нетерминалами $\boldsymbol{G_{left}}$ будут все состояния из ДС для $\boldsymbol{G_{right}}$, кроме *S.*

- Каждой дуге из состояния S в состояние W, помеченной символом *t*, ставится в соответствие правило W → t. Если в ДС есть входящие дуги в S, то в $\boldsymbol{G_{left}}$ добавляется также и правило W → St.

- Каждой дуге из состояния *V* в состояние *W*, помеченной символом *t*, ставится в соответствие правило *W* → *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*** $a_1 a_2 \ldots a_n$, если

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

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

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 леволинейных 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

$$(K, T, \delta , H, S) ,$$ 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!).

$\delta (A,t) = \{B1,B2,...,Bn\}$ означает, что из состояния A по входному символу t можно осуществить переход в любое из состояний Bi, i = 1, 2, ... ,n.

Алгоритм построения детерминированного КА по НКА

Вход: M = (K, T, δ , H, S) - НКА.
Выход: M1 = (K1, T, δ1, H1, S1) - ДКА, допускающий тот же язык, что и автомат M.
Метод:
Строим отображение δ1 (K1 × T → K1) по δ, **начиная с состояния H**.

1. δ1 для H ⊂ K1 строим т.о.:

- если в НКА переход из H по какому-то символу был детерминированным, переносим соответствующие отображение в результирующий КА. Состояния, появляющиеся в правой части отображений δ1 принадлежат K1;
- если же в НКА переход из H по какому-то символу t был недетерминированным, то в КА включаем отображения δ1 по правилу:
  δ1 (H, t) = A1A2...An, где Ai ⊂ K, и в НКА есть F (H, t) = Ai для всех 1 <= i <= n.

Состояние A1A2...An ⊂ K1.

2. Для всех состояний, появившихся в правой части отображений δ1 результирующего КА, определяем отображения δ1 т.о.:
δ1 (A1A2...An, t) = B1B2...Bm, где для каждого 1 <= j <= m в НКА существует δ (Ai, t) = Bj для каких-либо 1 <= i <= n.

Заключительными состояниями построенного детерминированного КА являются все состояния, содержащие S ⊂ K в своем имени. Для написания грамматики по КА все эти состояния **необходимо свести к** одному заключительному состоянию S1 ⊂ K1 с помощью символа ⊥.

Пример работы алгоритма преобразования НКА в КА

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

$\delta(H, 1) = B$

$\delta(A, 1) = B$

$\delta(A, 1) = S$

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

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

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 $\underline{BS}$.

Таким образом, $M1 = (\{H, B, A, \underline{BS}\}, \{0, 1\}, \delta 1, H, \underline{BS})$.

Комментарии и проблемы

- Алгоритм детерминизации конечного автомата был впервые предложен Эдвардом Форестом Муром.

- Быстрый рост размера автомата в результате детерминизации. (экспоненциальный взрыв)

- Избыточность детерминированного автомата.

Комментарии

- Можно ли описать автоматный язык без использования автоматов?

- Да, ведь он не зря называется регулярным.

- Регулярные выражения зачастую используются в текстовых редакторах и не только. Многие из современных языков программирования включают в себя явный механизм разбора регулярных выражений.

Регулярные выражения

Регулярные выражения над алфавитом A={a1, a2, …, an} — это формулы, которые определяются над множеством констант {0, 1, a1, a2, …, an} и множеством операций, состоящим из двухместных операций “x”, “+” и одноместной операции “*”.

Регулярным выражением называется всякая формула, удовлетворяющая требованиям:

1. каждая константа - регулярное выражение;

2. если формулы R1 и R2 являются регулярными выражениями, то формулы (R1xR2), (R1+R2), (R1*) также являются регулярными выражениями.

Приоритеты: *
x
+

Регулярные выражения

Значением регулярного выражения R является формальный язык L(R), который определяется по правилам:

1. $L(0) = \varnothing$
2. $L(1) = \{\varepsilon\}$
3. $L(a_i) = \{a_i\}$, $1<=i<=n$
4. $L(R1xR2) = L(R1)L(R2)$ (конкатенация)
5. $L(R1+R2) = L(R1) \cup L(R2)$ (объединение)
6. $L(R1^*) = L^*(R1)$ (итерация)

Язык называется регулярным, если он является значением некоторого регулярного выражения.

Теорема Клини

Формальный язык является автоматным тогда и только тогда, когда он является регулярным.

Задачи лексического анализа

1. выделить в исходном тексте цепочку символов, представляющую лексему, и проверить правильность ее записи;

2. удалить пробельные литеры и комментарии;

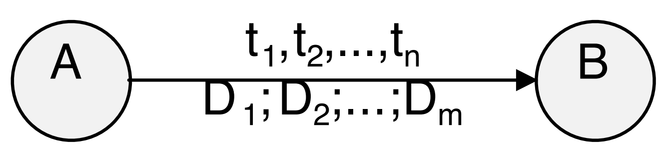
3. преобразовать цепочку символов, представляющих лексему, в пару:

   (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 вводится 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):

$I \rightarrow a \mid b \mid \ldots \mid z \mid Ia \mid Ib \mid \ldots \mid Iz \mid I0 \mid I1 \mid \ldots \mid I9$

целое без знака (N):

$N \rightarrow 0 \mid 1 \mid \ldots \mid 9 \mid N0 \mid N1 \mid \ldots \mid N9$

Описание модельного языка

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

Замечания:

- запись вида **{**α**}** 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 **[** α **|** β **]** означает, что в порождаемой цепочке в этом месте может находиться либо α, либо β.
- **P** - цель грамматики; символ ⊥ - 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_лексемы, значение_лексемы).

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

<u>Соглашение об используемых таблицах лексем</u>:

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

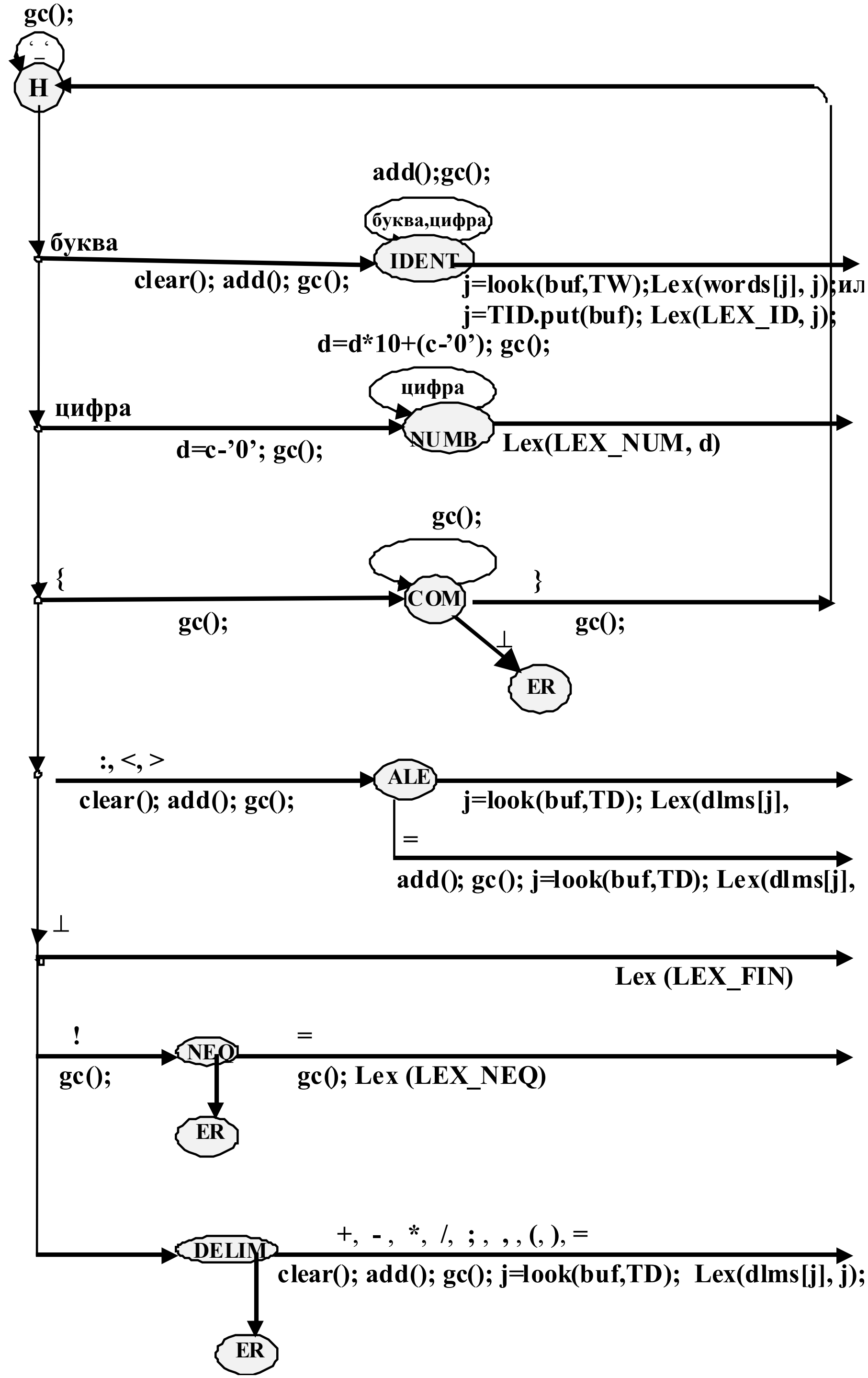
```
void  Scanner::clear ( ) {
      buf_top = 0;
      for (int j = 0;  j < 80;  j++ )
              buf [ j ] = '\0';
}
void  Scanner::add ( ) {
      buf [ buf_top ++ ] = c;
}
int  Scanner::look (const char * buf,  char * * list) {
      int i = 0;
      while (list [ i ]) {
          if ( ! strcmp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


gc();
H
add();gc();
буква,цифра
буква
IDENT
clear(); add(); gc();
j=look(buf,TW);Lex(words[j], j);ил
j=TID.put(buf); Lex(LEX_ID, j);
d=d*10+(c-'0'); gc();
цифра
цифра
NUMB
Lex(LEX_NUM, d)
d=c-'0'; gc();
gc();
{
COM
}
gc();
gc();
⊥
ER
:, <, >
ALE
clear(); add(); gc();
j=look(buf,TD); Lex(dlms[j],
=
add(); gc(); j=look(buf,TD); Lex(dlms[j],
⊥
Lex (LEX_FIN)
!
NEQ
=
gc();
gc(); Lex (LEX_NEQ)
ER
DELIM
+, -, *, /, ;, ,, (,), =
clear(); add(); gc(); j=look(buf,TD); Lex(dlms[j], j);
ER

```
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 );
    }
}
```