Мы рассмотрели основные понятия теории формальных языков, что дает математическую базу для изучения основ трансляции.

ТФЯ является одной из старейших и наиболее фундаментальных областей информатики, ее результаты используются не только в теории трансляции, но и в других областях математики, лингвистики, биологии.

Основы трансляции

- задача разбора
- лексический анализ
- синтаксический анализ
- семантические действия
- формальный перевод
- генерация кода (на языке ПОЛИЗ)
- интерпретация ПОЛИЗ

***Задача разбора**:*

Даны КС-грамматика G и цепочка x.

$x \in L(G)$?

Если да, то построить дерево вывода для x

(или левый вывод для x, или правый вывод для x).

***Задача распознавания**: $x \in L(G)$? Дерево или вывод не требуются в качестве ответа, только ответ - «да» или «нет».*

Задача распознавания алгоритмически неразрешима в классе языков типа 0;

разрешима в классе языков типа 1.

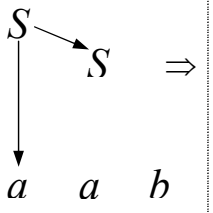
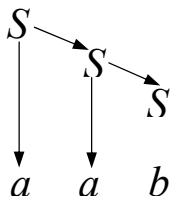
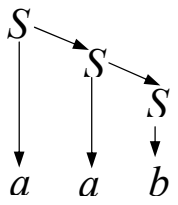
Для КС-языков и регулярных языков существуют эффективные алгоритмы разбора.

Регулярные и КС-языки используются при описании синтаксиса языков программирования

Построение дерева вывода

G: $S \to a\,S \mid b$ Цепочка: aab

Сверху вниз:

<table>
<tr>
<td>S
$\Rightarrow$
$a \quad a \quad b$</td>
<td> 
$a \quad a \quad b$</td>
<td> 
$a \quad a \quad b$</td>
<td> 
$a \quad a \quad b$</td>
</tr>
<tr>
<td>Левый вывод:
$S \to$</td>
<td>$a\,S \to$</td>
<td>$a\,a\,S \to$</td>
<td>$a\ a\ b$</td>
</tr>
</table>

Построение дерева вывода

G: $S \to a\ S \mid b$ Цепочка: aab

Свертка – это применение правила вывода «в обратную сторону», замена правой части на нетерминал из левой части:

$aa\underline{b} \leftarrow aaS$ — *свертка* по правилу $S \to b$. Обозначаем свертку с помощью обратной стрелки $\leftarrow$.

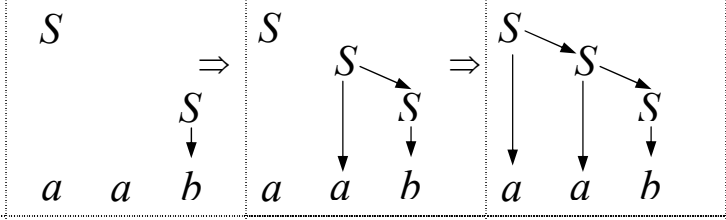
С помощью сверток можно построить вывод «задом наперед» (обращение вывода): от цепочки к цели грамматики S. Например, сентенциальную форму aaS можно свернуть к aS, а затем к S :

$aab \leftarrow aaS \leftarrow aS \leftarrow S$

Построение дерева вывода

G: $S \to a\,S \mid b$ Цепочка: aab

Снизу вверх:

			
Обратный правый вывод: $a\,a\,b \leftarrow$	$a\,a\,S \leftarrow$	$a\,S \leftarrow$	S

РЕГУЛЯРНЫЕ ЯЗЫКИ

способы описания:

-- регулярные грамматики

(леволинейные либо праволинейные)

-- конечные автоматы

(недетерминированные или детерминированные)

-- регулярные выражения

(см. материал “О регулярных языках” на cmcmsu.no-ip.info)

Недетерминированный конечный автомат (НКА) — это пятерка $\mathcal{A} = (K, \Sigma, \delta, I, F)$, где:

K — конечное множество состояний, или вершин;
Σ — входной алфавит (также конечный);
$\delta \subseteq K \times \Sigma \times K$ — множество команд, или дуг;
$I \subseteq K$ — множество начальных состояний;
$F \subseteq K$ — множество заключительных состояний.

Множество δ можно также интерпретировать как отображение $K \times \Sigma$ в множество подмножеств K.

$\mathcal{A} = (K, \Sigma, \delta, I, F)$ — НКА.

Каждая дуга НКА $\mathcal{A}$ имеет пометку из Σ.

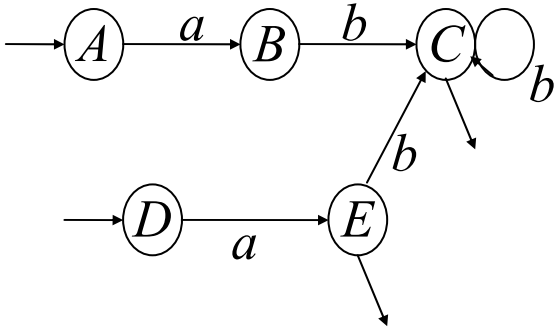
Путь в ориентированном графе может быть представлен последовательностью дуг. *Пустой* путь можно представить одной вершиной, которая считается одновременно началом и концом пути.

Пометка пути — это сцепление (конкатенация) пометок его дуг. Пустой путь имеет пустую пометку. Путь из начальной вершины в заключительную называется *успешным*.

Язык, допускаемый автоматом $\mathcal{A}$ (обозначается $L(\mathcal{A})$), — это множество пометок всех успешных путей автомата.

Пример 1. $\mathcal{A}_1 = (\{A, B, C, D, E\}, \{a, b\}, \delta, \{A, D\}, \{C, E\})$, где $\delta = \{(A, a, B), (D, a, E), (B, b, C), (E, b, C), (C, b, C)\}$.

Автомат удобно представлять в виде ориентированного размеченного графа:



Входящими непомеченными стрелками отмечены начальные вершины A и D, исходящими — заключительные вершины E и C.

$$L(\mathcal{A}_1) = \{ab^n \mid n \geq 0\}$$

Алгоритм построения НКА по леволинейной грамматике (без пустых правых частей)

1. Множество вершин НКА состоит из нетерминалов грамматики и еще одной новой вершины H, которая объявляется начальной.
2. Каждому правилу вида $A \to Ba$ в автомате соответствует дуга из вершины B в вершину A, помеченная символом a: $B \xrightarrow{a} A$. Каждому правилу вида $A \to a$ соответствует дуга $H \xrightarrow{a} A$. Других дуг нет.
3. Заключительной вершиной автомата является вершина, соответствующая начальному символу грамматики. Начальной является вершина H, построенная на шаге 1.

$G = (\{a, b, \perp\}, \{S, A, B, C\}, P, S)$, где

P: $S \to C\perp$

$C \to Ab \mid Ba$

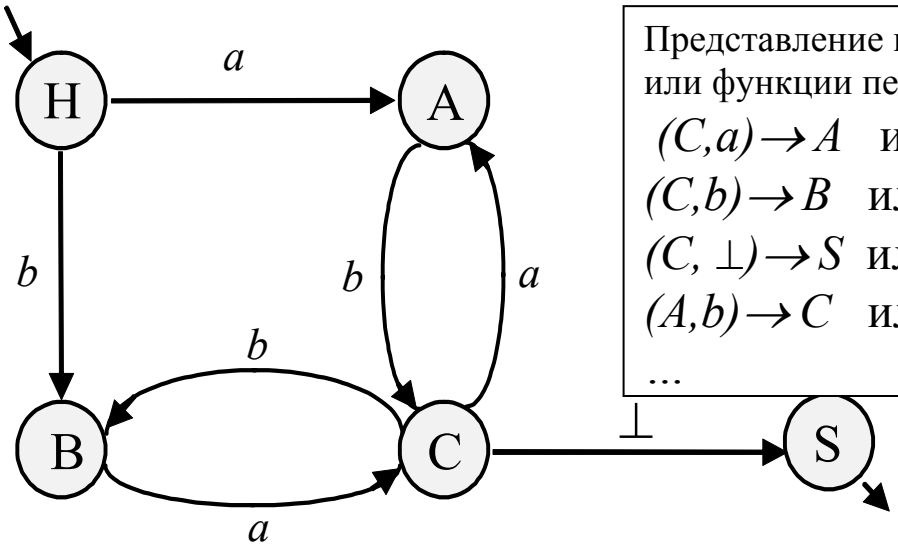
$A \to a \mid Ca$

$B \to b \mid Cb$

Диаграмма состояний для грамматики G – это граф, представляющий конечный автомат, построенный нашим алгоритмом по грамматике G.

	a	b	$\perp$
C	A	B	S
A	-	C	-
B	C	-	-
S	-	-	-

Представление в виде таблицы



Представление в виде набора команд или функции переходов :

$(C,a) \to A$	или	$\delta(C,a) = \{A\}$
$(C,b) \to B$	или	$\delta(C,b) = \{B\}$
$(C, \perp) \to S$	или	$\delta(C, \perp) = \{S\}$
$(A,b) \to C$	или	$\delta(A,B) = \{C\}$
...		...

Алгоритм построения леволинейной грамматики по НКА с единственным заключительным состоянием

1. Нетерминалами грамматики будут вершины автомата, терминалами — пометки дуг.
2. Для каждой дуги $A \xrightarrow{a} B$ в грамматику добавляется правило $B \to Aa$. Для каждой начальной вершины B в грамматику добавляется правило $B \to \varepsilon$.
3. Начальным символом будет нетерминал, соответствующий заключительной вершине.
4. К построенной по пунктам 1—3 грамматике применить алгоритм устранения ε-правил.

Конечный автомат называется **детерминированным** конечным автоматом (ДКА), если он имеет единственное начальное состояние, и любые две дуги, исходящие из одной и той же вершины имеют различные пометки.

Множество δ в ДКА можно интерпретировать как отображение $K \times \Sigma$ в множество K.

Тогда конечный автомат ***допускает цепочку*** $a_1a_2...a_n$, если $\delta\ (H,a_1) = A_1$; $\delta\ (A_1,a_2) = A_2$; ... ; $\delta\ (A_{n-2},a_{n-1}) = A_{n-1}$; $\delta(A_{n-1},a_n) = S$, где $a_i \in \Sigma$, $A_j \in K$, $j = 1, 2,..., n-1$; $i = 1,2,..., n$; H – начальное состояние, S – одно из заключительных состояний.

Язык, допускаемый ДКА — это множество всех допускаемых им цепочек.

Алгоритм построения ДКА по НКА

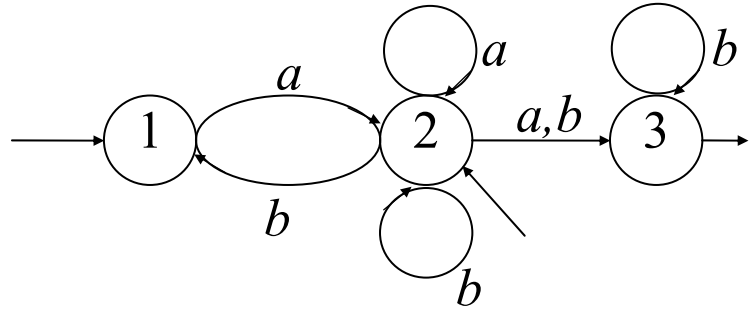
Вход: $\mathcal{A}' = (K', \Sigma, \delta', I, F)$ — НКА.

Выход: $\mathcal{A} = (K, \Sigma, \delta, InitState, FinalStates)$ — ДКА.

Метод: Вершинами (состояниями) автомата $\mathcal{A}$ будут подмножества множества K' автомата $\mathcal{A}'$. *CurState* и *NewState* — вспомогательные переменные для хранения таких подмножеств. Сам алгоритм запишем в паскалеподобном стиле. Фигурные скобки означают конструкторы множеств.

begin $InitState := \{s \mid s \in I\}$; $K := \{InitState\}$; $\delta := \varnothing$;
  while (*в K есть нерассмотренный элемент*)
  begin
    $CurState :=$ *нерассмотренный элемент из K*;
    for (*каждого* $a \in \Sigma$)
    begin
      $NewState := \{q \mid (p \xrightarrow{a} q) \in \delta, p \in CurState\}$;
      $K := K \cup \{NewState\}$;
      $\delta := \delta \cup \{(CurState \xrightarrow{a} NewState)\}$;
    end
  end;
  $FinalStates := \{P \in K \mid$ *существует* $q \in P$: $q \in F\}$
end.

Пример. A':



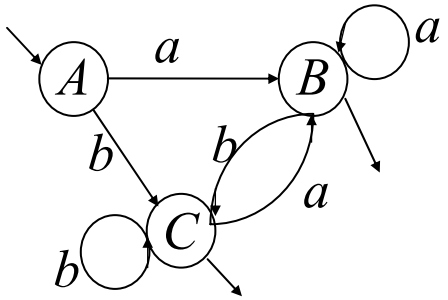
Процесс построения ДКА удобно изобразить в виде таблицы, начав с состояния {1, 2}. Затем заполняем строки для вновь появляющихся состояний.

символ / состояние	*a*	*b*
{1, 2}	{2, 3}	{1, 2, 3}
{2, 3}	{2, 3}	{1, 2, 3}
{1, 2, 3}	{2, 3}	{1, 2, 3}

Обозначим состояние {1, 2} через A, {2, 3} — B, {1, 2, 3} — C.

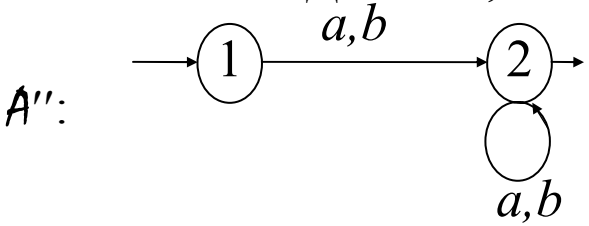
символ / состояние	a	b
A	B	C
B	B	C
C	B	C

С учетом переобозначений построим по таблице ДКА $\hat{A}$:



$L(\hat{A})=\{a, b\}^+$

Можно заметить, что язык $L = \{a, b\}^+$, допускаемый автоматом A, допускается также ДКА A'', имеющим только два состояния.



Существует алгоритм, позволяющий по любому ДКА построить эквивалентный ДКА с минимальным числом состояний.

Для более удобной работы с диаграммами состояний введем несколько соглашений:

a) если из одного состояния в другое выходит несколько дуг, помеченных разными символами, то будем изображать одну дугу, помеченную всеми этими символами;

b) непомеченная дуга будет соответствовать переходу при любом символе, кроме тех, которыми помечены другие дуги, выходящие из этого состояния.

c) введем состояние ошибки (ERR); переход в это состояние будет означать, что исходная цепочка языку не принадлежит.

Алгоритм моделирования работы ДКА

Вход: ДКА $\mathcal{A} = (K, \Sigma, \delta, I, F)$ и цепочка $x\perp$, где $x \in \Sigma^*$, $\perp \notin \Sigma$ — маркер конца цепочки.

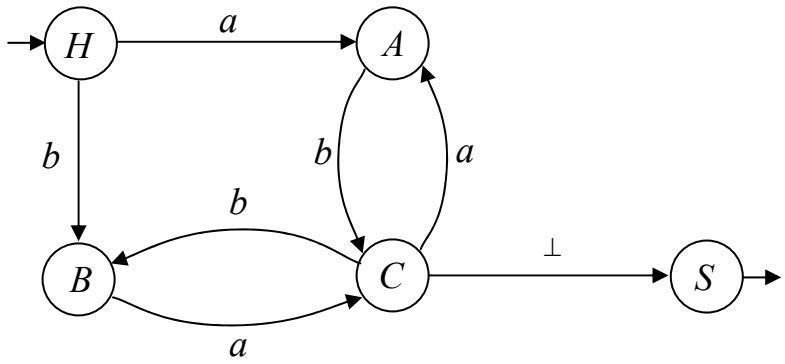
Выход: «Да», если $x \in L(\mathcal{A})$, иначе — «Нет».

Метод: Введем переменные *St* для хранения текущего состояния автомата и *c* для хранения очередного считанного символа входной цепочки *x*.

```
begin
    c := первый символ цепочки x;
    St := I; {начальное состояние}
    while ( St ≠ ERR and c ≠ '⊥' )
    begin
        St := δ (St, c);
        c := очередной символ
    end;
    if St ∈ F then
        write ('Да')
    else
        write ('Нет')
end;
```

Пример анализатора для грамматики $G = \langle \{a, b, \perp\}, \{S, A, B, C\}, P, S \rangle$, где

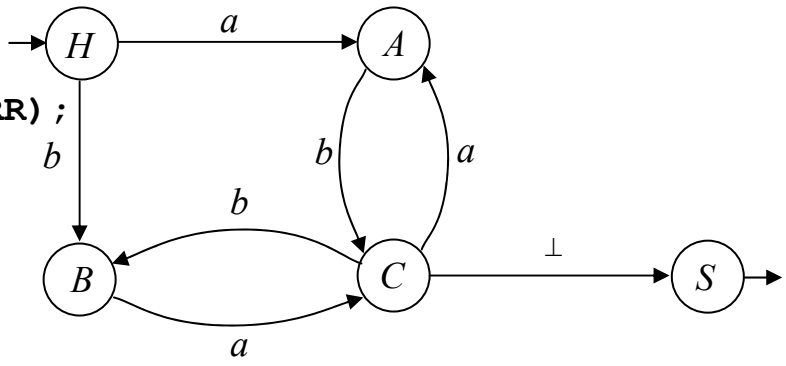
$$P: S \to C\perp$$
$$C \to Ab \mid Ba$$
$$A \to a \mid Ca$$
$$B \to b \mid Cb$$



Программа-анализатор на C++ :

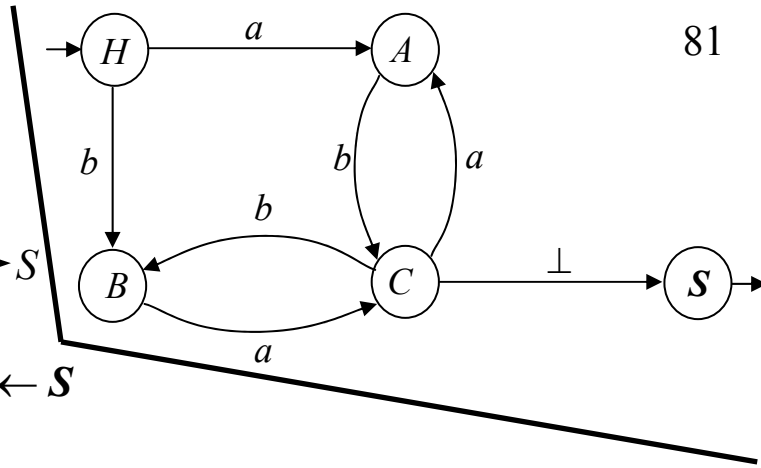
```
#include <iostream.h>
char c; //текущий символ

void gc () {cin >> c;} // считать очередной символ со входа

bool scan_G(){

enum state {H, A, B, C, S, ERR};   //множество состояний

state CS;   // CS — текущее состояние

CS=H;

gc(); // считать первый символ
```

```
do {switch (CS) {
   case H: if (c == 'a') { gc(); CS = A;}
           else if (c == 'b') { gc(); CS = B;}
                else CS = ERR;
           break;
   case A: if (c == 'b') { gc(); CS = C;}
           else CS = ERR;
           break;
   case B: if (c == 'a') { gc(); CS = C;}
           else CS = ERR;
           break;
   case C: i (c =='a') { gc(); CS = A;}
           else if (c == 'b') { gc(); CS = B;}
                else if (c == '⊥') CS = S;
                     else CS = ERR;
           break;
                }
  } while (CS != S && CS != ERR);
if (CS == ERR)
  return false;
else
  return true;

}
```

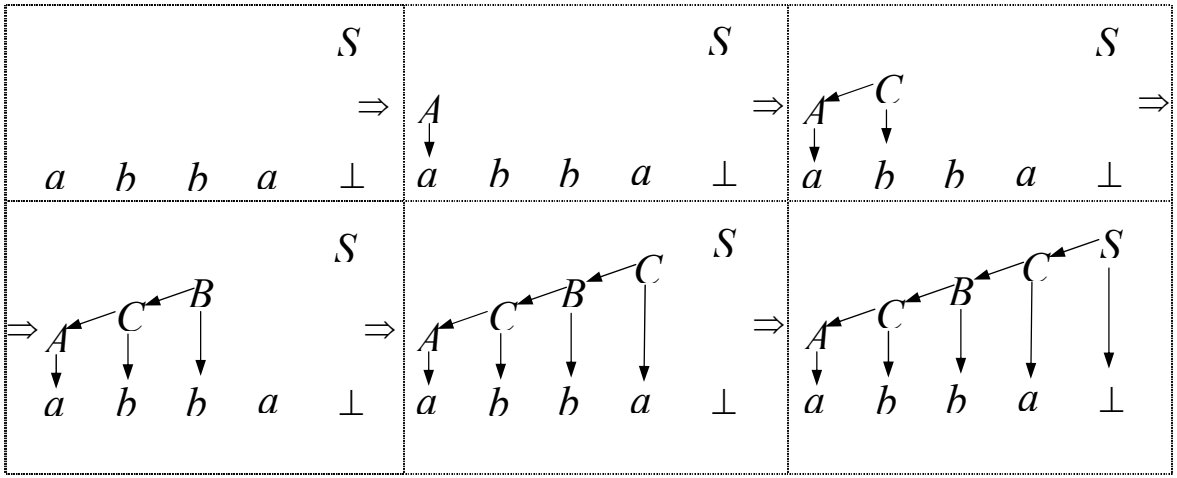


Пример разбора цепочки $abba\perp$

$$H \xrightarrow{a} A \xrightarrow{b} C \xrightarrow{b} B \xrightarrow{a} C \xrightarrow{\perp} S$$



$$abba\perp \leftarrow Abba\perp \leftarrow Cba\perp \leftarrow Ba\perp \leftarrow \underline{C\perp} \leftarrow \boldsymbol{S}$$



Недетерминированный разбор

Если конечный автомат, построенный по грамматике, недетерминированный, то нужно перебирать все возможные варианты переходов. Можно также преобразовать его в эквивалентный ДКА и проводить детерминированный разбор.

Пример использования автоматов в решении теоретических задач

Утверждение. *Контекстно-свободный язык*

$$L = \{a^n b^n \mid n \geq 1\}$$

нерегулярен

(доказательство см. http://cmcmsu.no-ip.info/download/regular.languages.pdf)

Конечные автоматы (диаграммы состояний) с действиями

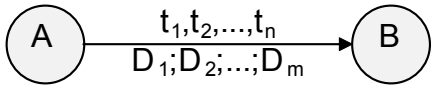
идентификатор (I):

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

целое без знака (N):

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

ДС с действиями может выглядеть так:



Смысл t_i прежний — если в состоянии A очередной анализируемый символ совпадает с t_i для какого-либо i = 1, 2 ,... n, то осуществляется переход в состояние B; при этом необходимо выполнить действия D_1, D_2, ... ,D_m.

Лексический анализ (ЛА) — это первый этап процесса компиляции. На этом этапе литеры, составляющие исходную программу, группируются в отдельные элементы, называемые лексемами.

задачи лексического анализатора:

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

- зафиксировать в специальных таблицах для хранения разных типов лексем факт появления соответствующих лексем в анализируемом тексте;

- преобразовать цепочку символов, представляющих лексему, в пару:

(тип_лексемы, указатель_на_информацию_о_ней);

- удалить пробельные литеры и комментарии.

Лексический анализатор для М-языка

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

Контекстные условия:

1. Любое имя, используемое в программе, должно быть описано и только один раз.

2. В операторе присваивания типы переменной и выражения должны совпадать.

3. В условном операторе и в операторе цикла в качестве условия возможно только логическое выражение.

4. Операнды операции отношения должны быть целочисленными.

5. Тип выражения и совместимость типов операндов в выражении определяются по обычным (паскалевским) правилам; старшинство операций задано синтаксисом.

Проектирование структуры классов лексического анализатора М-языка

Представление лексем: выделим следующие типы лексем:

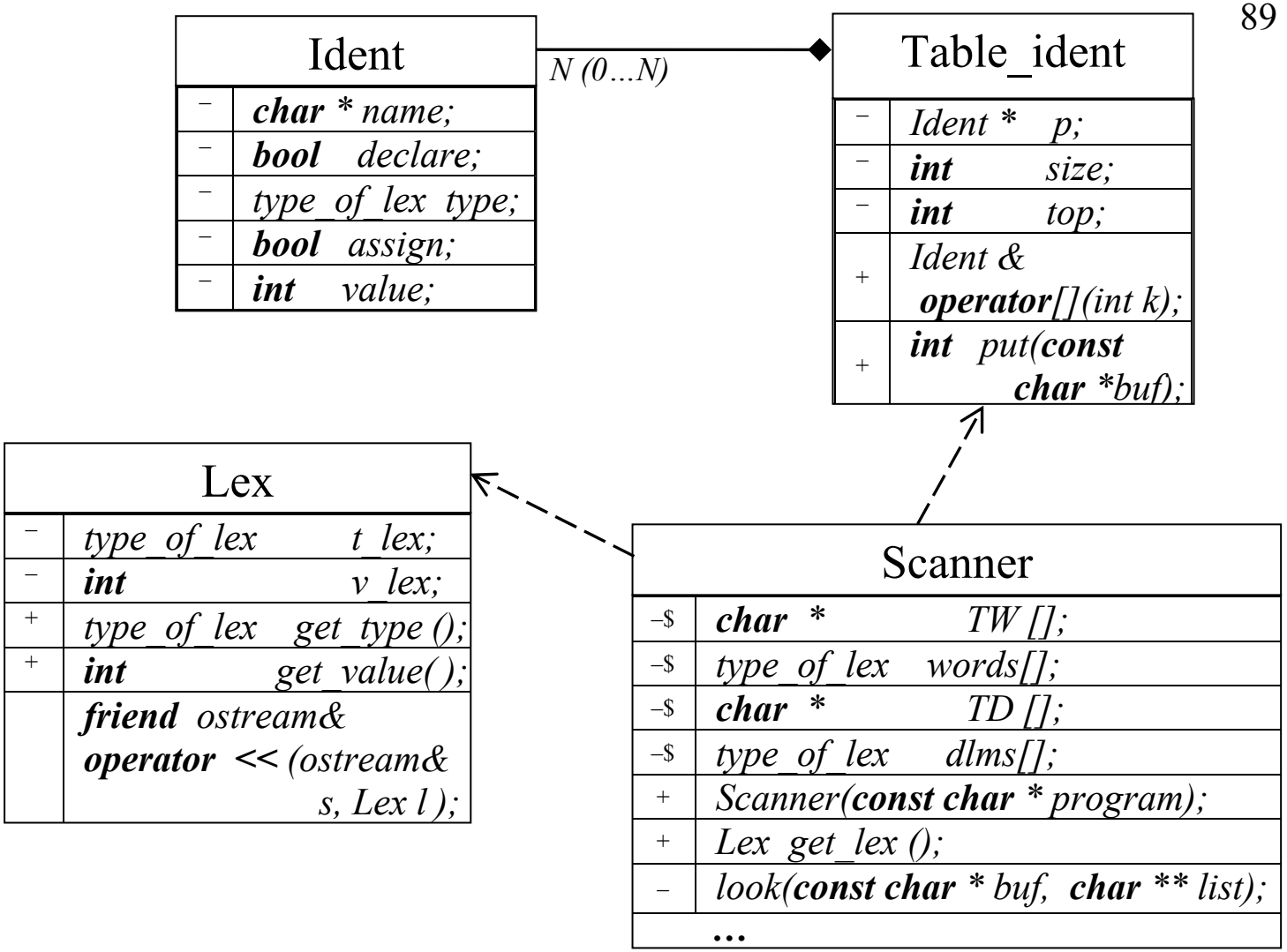
```
enum  type_of_lex {LEX_NULL, /*0*/
                   LEX_AND, LEX_BEGIN, … LEX_WRITE, /*18*/
                   LEX_FIN, /*19*/
                   LEX_SEMICOLON, LEX_COMMA, … LEX_GEQ, /*35*/
                   LEX_NUM, /*36*/
                   LEX_ID, /*37*/
                   POLIZ_LABEL, /*38*/
                   POLIZ_ADDRESS, /*39*/
                   POLIZ_GO, /*40*/
                   POLIZ_FGO}; /*41*/
```

Соглашение об используемых таблицах лексем:

TW – таблица служебных слов М-языка;
TD – таблица ограничителей М-языка;
TID – таблица идентификаторов анализируемой программы.

Таблицы TW и TD заполняются заранее, т.к. их содержимое не зависит от исходной программы.

Таблица TID формируется в процессе анализа.


Ident
– char * name;
– bool declare;
– type_of_lex type;
– bool assign;
– int value;
N (0...N)
Table_ident
– Ident * p;
– int size;
– int top;
+ Ident & operator[](int k);
+ int put(const char *buf);
Lex
– type_of_lex t_lex;
– int v_lex;
+ type_of_lex get_type();
+ int get_value();
friend ostream& operator << (ostream& s, Lex l);
Scanner
–$ char * TW [];
–$ type_of_lex words[];
–$ char * TD [];
–$ type_of_lex dlms[];
+ Scanner(const char * program);
+ Lex get_lex ();
– look(const char * buf, char ** list);
...

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   (type_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 () {
                 buf_top = 0;
                 for (int j = 0; j < 80; j++ )
                   buf[j] = '\0';
       }
```

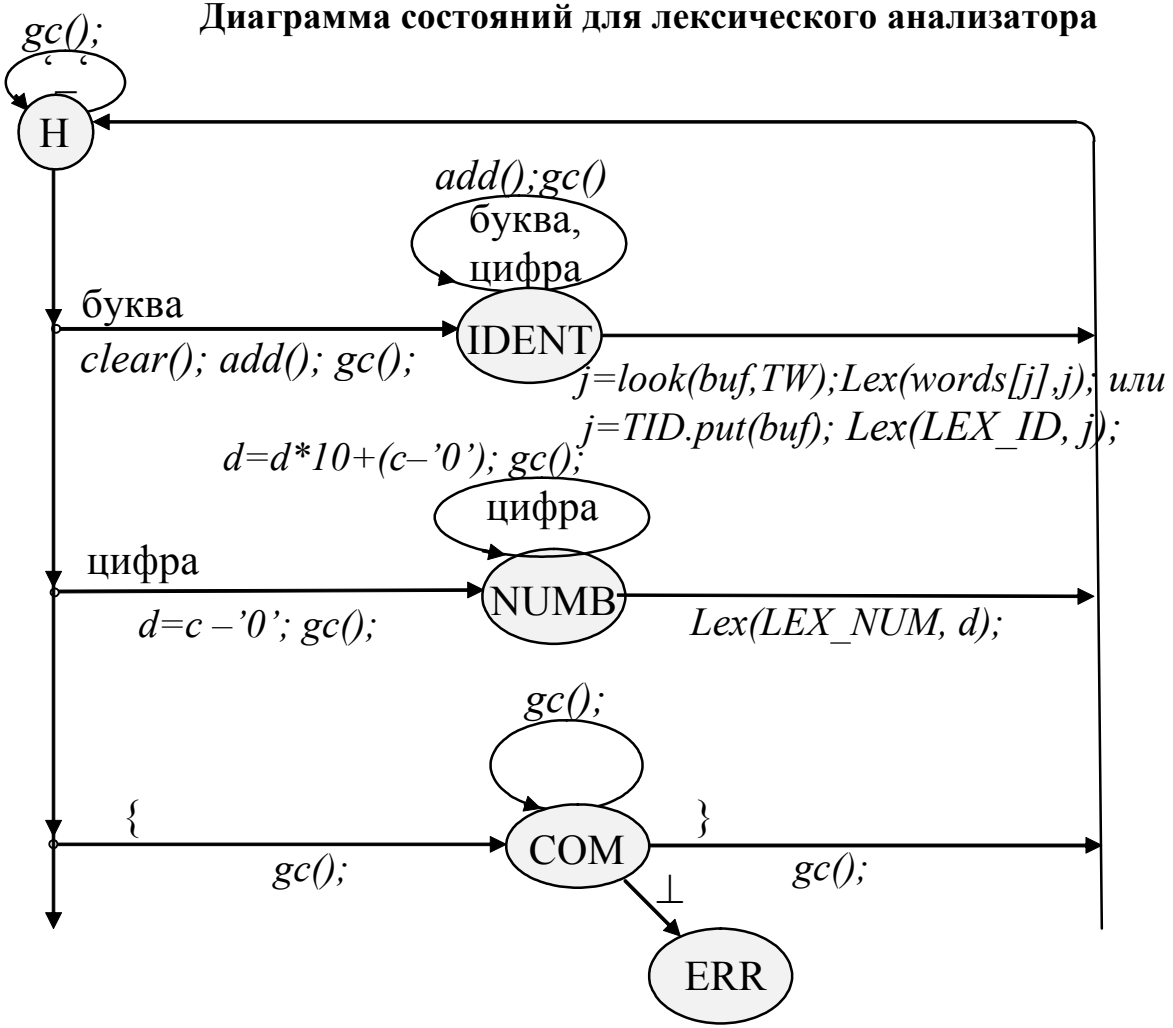
```
    void  add () { buf [ buf_top ++ ] = c; }
    int   look (const char *buf, char **list) {
            int i = 0;
            while (list[i]) {
              if (!strcmp(buf, list[i])) return i;
              i++;
            }
            return 0;
    }
    void  gc () { c = fgetc (fp); }

  public:
      Scanner (const char * program){
          fp = fopen ( program, "r" ); CS = H;
          clear(); gc();
      }
      Lex get_lex ();
};
```

Таблицы лексем М-языка:

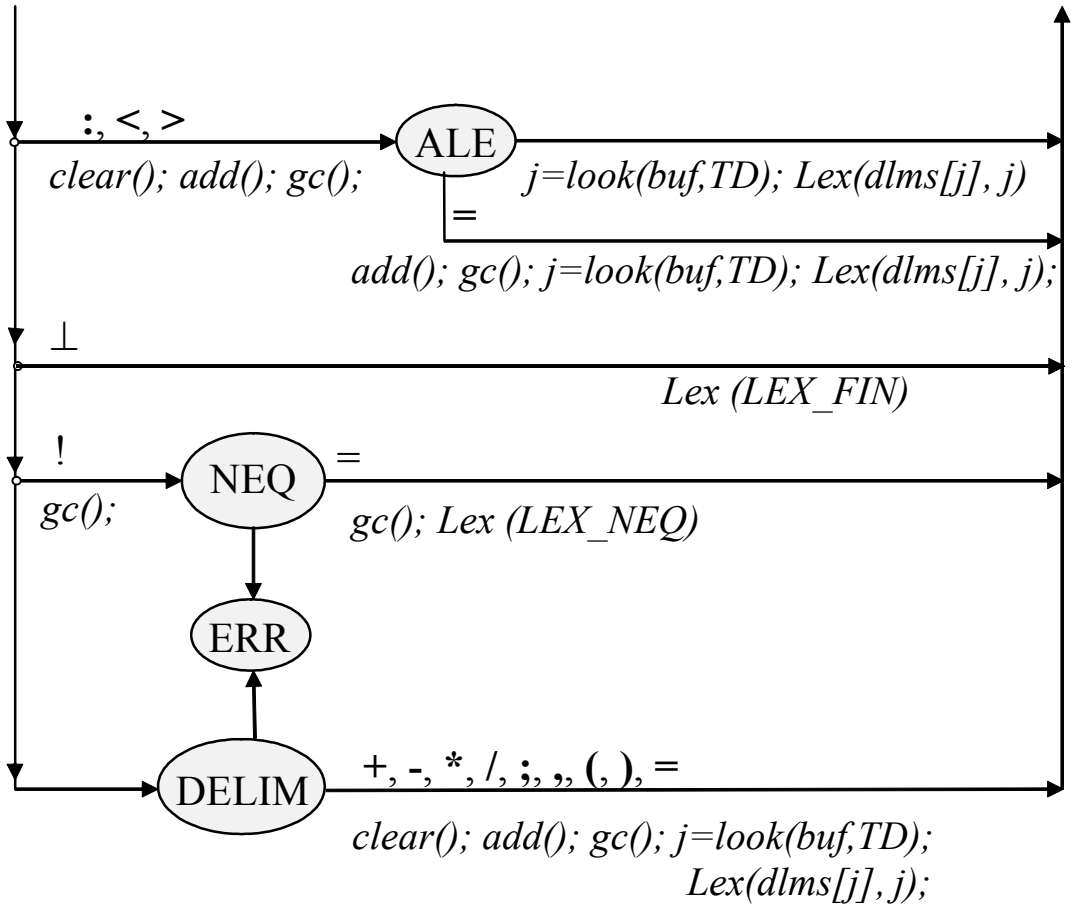
```
char * Scanner:: TW [] = {

    NULL,"and","begin","bool","do","else","end",
//   0      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,";", "@", ",", ":", ":=", "(", ")",
//         0     1     2    3     4     5      6    7
         "=","<",">","+","-","*","/","<=","!=",">="
//       8    9   10   11   12  13   14   15   16    17
  };

  tabl_ident  TID(100);
```

```
type_of_lex
  Scanner::words [] = {LEX_NULL, LEX_AND, LEX_BEGIN,
        LEX_BOOL, LEX_DO, LEX_ELSE, LEX_END, LEX_IF,
        LEX_FALSE, LEX_INT,LEX_NOT, LEX_OR,
        LEX_PROGRAM, LEX_READ, LEX_THEN, LEX_TRUE,
        LEX_VAR, LEX_WHILE, LEX_WRITE, LEX_NULL};

type_of_lex
  Scanner::dlms  [] = {LEX_NULL, LEX_FIN,LEX_SEMICOLON,
        LEX_COMMA, LEX_COLON, LEX_ASSIGN, LEX_LPAREN,
        LEX_RPAREN, LEX_EQ,LEX_LSS, LEX_GTR, LEX_PLUS,
        LEX_MINUS, LEX_TIMES, LEX_SLASH, LEX_LEQ,
        LEX_NEQ, LEX_GEQ, LEX_NULL};
```

Диаграмма состояний для лексического анализатора



ДС ЛА (продолжение)



```
Lex Scanner::get_lex (){
  int d, j;
  CS = H;
  do {
      switch(CS) {
      case H:
          if(c ==' ' || c =='\n' || c=='\r' || c =='\t') gc();
          else
          if(isalpha(c)) {clear(); add(); gc(); CS = IDENT;}
          else
          if ( isdigit(c) ) { d = c - '0'; gc(); CS = NUMB; }
          else
          if ( c== '{' ) { gc(); CS = COM; }
          else
          if (c== ':' || c== '<' || c== '>') {clear(); add();
                                              gc(); CS = ALE; }
          else
          if (c == '@') return Lex(LEX_FIN);
          else
              if (c == '!') {clear(); add(); gc(); CS = NEQ; }
              else CS = DELIM;
              break;
```

```
case IDENT:
        if ( isalpha(c) || isdigit(c) ) {add(); gc();}
        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'); gc(); }
        else  return Lex ( LEX_NUM, d);
        break;
case COM:
        if ( c == '}' ) { gc(); CS = H; }
        else
        if (c == '@' || c == '{' ) throw c;
        else gc();
        break;
case ALE:
        if (c=='=') { add(); gc(); 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(); gc(); j = look ( buf, TD );
                return Lex ( LEX_NEQ, j ); }
            else throw '!';
            break;
case DELIM:
            clear(); add();
            if (j = look(buf, TD)) {
                gc(); return Lex (dlms[j], j);}
            else throw c;
            break;

        } //end of switch

    } while (true);

}// end of getlex()
```