Карпов Л. Е.

Материалы к лекциям "Системы программирования"

ВМ и К, 2-й курс, 4 семестр

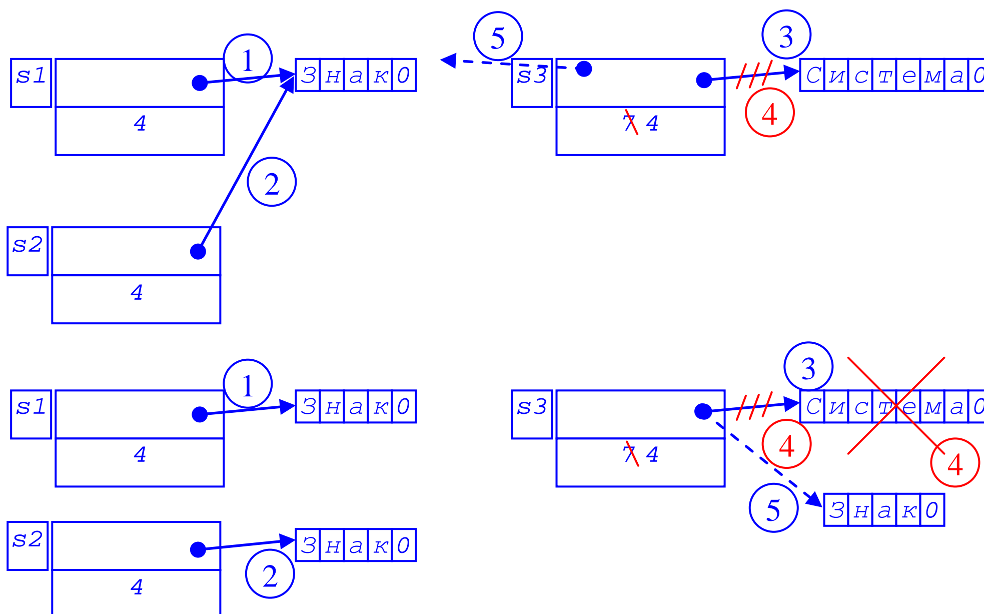
Системы классов, используемые в лекциях

Класс Box

```
   class Box {  double len;    // length – длина
                double wid;    // width  – ширина
                double hei;    // height – высота
      public:   double volume () { return len * wid * hei; }
                void set_dimensions (double l, double w, double h)
                               { len = l;  wid = w;  hei = h; }
//для задания начальных значений всех трёх параметров параллелепипедов:
                Box (double l, double w, double h )
                               { hei = h; wid = w; len = l; }
// если часто используются кубики, то достаточно одного параметра:
                Box (double s) { hei = wid = len = s; }
// если часто используются коробки одного типа (кирпичи), параметры не нужны:
                Box ()         { hei = 6; wid = 12; len = 24; }
// другой вариант конструктора-умолчания:
                Box (double l = 6, double w = 12, double h = 24) {}
// конструктор копирования:
                Box (Box &a) // или Box (const Box &a)
                       { if (this != &a) { len = a.len; wid = a.wid; hei = a.hei; }
                              // этот конструктор (как и любой другой) не может
                              // возвращать никакого значения!
                       }
   };

Box b1 (1, 2, 3);
Box b2 (5);
Box b3;       // конструктор умолчания
Box * b4 = new Box (2.3);
Box b5 = Box ();

void f(){ Box * bf4 = new Box (2, 3, 5); // создание нового объекта
                                         // и инициализация указателя на него
          Box bf5 = * bf4;               // инициализация описываемого объекта
                                         // значениями, извлекаемыми по указателю
          Box bf6 = Box (4, 7, 1);       // создание временного
}                                        // объекта и инициализация bf6
```

Класс Str

```
class str { char * p; int size;

public: str (const char * s = 0) // для инициализации строками-константами
               { if (s) { p = new char [(size = strlen(s))+1]; strcpy (p, s); } }
       ~str () { delete [] p; }
        str (const str & a)  // конструктор копирования
               { p = new char [(size = a.size) + 1];      strcpy (p, a.p); }
  str & operator = (str & a)
   { if (this != & a)          // равны ли адреса объектов?
        { delete [] p;         // уничтожение старого значения
          p = new char [(size = a.size) + 1];      strcpy (p, a.p);
        }
     return * this;            // для s3 = s1: *this == s3, a == s1
   }
  const int   length () const  { return strlen (p); }
  const char* gets    () const  { return p; }
  str & concat (const str &s)  // конкатенация строк
   {  str temp;
      temp.p = new char [(temp.size = length()) + 1];
      strcpy (temp.p, p);         delete[] p;
      p = new char [(size = temp.length() + s.length()) + 1];
      strcpy (p, temp.p);         strcpy (p + length(), s.p);
      return *this;              // возврат ссылки на свой объект
   }
  str & operator += (const str & s){ return concat (s); }
  char& operator [] (int i) // индексация есть нестатический метод
   {  if (i < 0 || i >= length ())
        { cerr << "str: ошибка размера:" << i << endl; exit (1); }
      return p [i];
   }
};
```



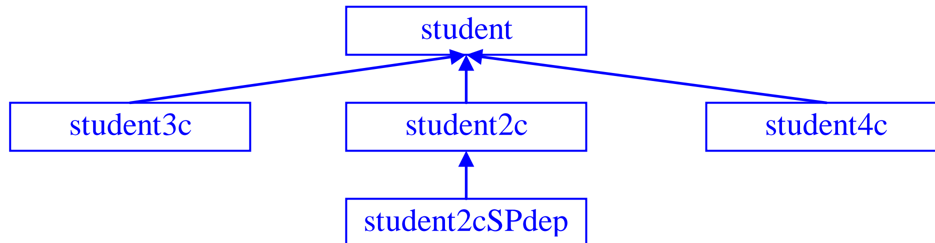

```
void fs () { str s ("Системы программирования");
  char c;     c = s [3]; // эквивалентно c = s.operator[](3); => c == 'т'
}
void sf()
{ str s1 ("Знак");     cout << s1.gets (); str s2 = s1; cout << s2.gets () << endl;
  str s3 ("Система"); cout << s3.gets ();      s3 = s1; cout << s3.gets () << endl;
  s1.concat (" и ").concat (s3).concat (" - понятия"); cout << s1.gets () << endl;
}
```

Класс complex

```
class complex { double re, im;
    public:
                // 3 конструктора:
complex (double r = 0, double i = 0)    {re = r; im = i;}
                // конструктор копирования:
complex (const complex & a)             {re = a.re; im = a.im;}
                // деструктор:
~complex () {}
                // операция присваивания:
complex & operator = (const complex & a) { re = a.re; im = a.im; return * this; }

   void ChangeModule (const double a)   { re *= a; im *= a; }
   void print () const { cout << "re = " << re << ", im = " << im << endl; }

//                  Перегрузка операции методом класса
   const  complex   operator+ (const complex & a) const
                       { complex temp (re + a.re, im + a.im); return temp; }
//                  Перегрузка операции функцией-другом класса
   friend complex   operator* (const complex &a, double b);

//                  Перегрузка унарной операции методом класса
   const  complex   operator-() const { complex temp (- re, - im); return temp; }
//                  Перегрузка унарной операции функцией-другом класса
   friend complex   operator+(complex &a);

//                  Перегрузка префиксной и постфиксной унарных операций
   const  complex & operator++() { ++ re; return * this; }
   const  complex   operator++(int pusto)
                        { complex temp = * this; ++ re; return temp; }

};


complex operator * (const complex & a, double b) {
        complex temp (a.re * b, a.im * b);
        return temp;
}
complex operator+(complex &a) { return a; }



  complex a (16.2, -8.3);
  complex d = a; // инициализация существующим объектом
  complex e = complex (1, 2);
  d.operator = (e);
      complex x (1, 2), y, z;
      z = ++ x; // z.re = 2, z.im = 2, x.re = 2, z.im = 2
      z = x ++; // z.re = 2, z.im = 2, x.re = 3, z.im = 2
      ++ ++ x;  // ОШИБКА, так как возвращается не адресное значение
      y = (x + y)++;          // ошибка, так как сложение возвращает
                              // не адресное значение
```

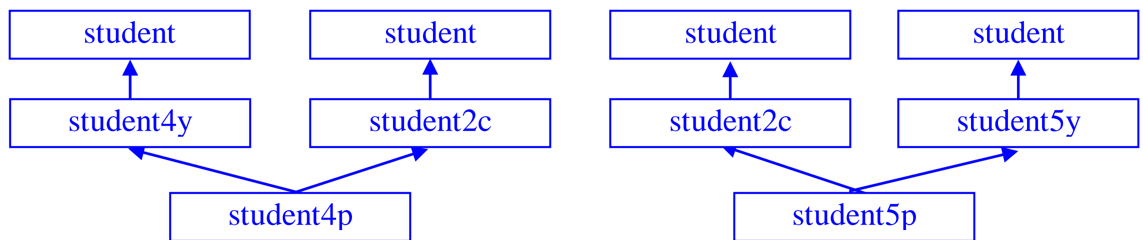
Системы классов student

Одиночное наследование



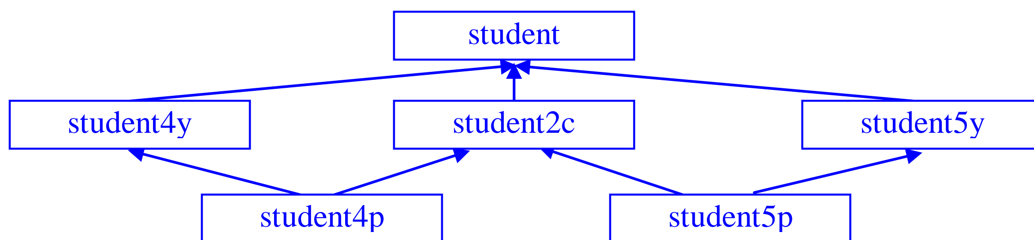

```
class student { protected: char * name;
                int year;        // год обучения
                double avb;      // средний балл
                int student_id;  // номер зачётной книжки
        public: student (char* nm, int y, double b, int id):
                         year (y), avb(b), student_id (id)
                  { name = new char [strlen (nm) + 1];  strcpy (name, nm); }
                char * get_name () const { return name; }
    virtual     void print ();
    virtual    ~student () { delete [] name; } // виртуальный деструктор
};
class student2c: public student { // указание на базовый класс
  protected: char* pract2;   char* tutor2;
     public: student2c (char* n, double b, int id,
                        char* p, char* t) : student (n, 2, b, id)
               { pract2 = new char [strlen (p) + 1];   strcpy (pract2, p);
                 tutor2 = new char [strlen (t) + 1];   strcpy (tutor2, t); }
    virtual  void print (); // эта функция печати атрибутов скрывает
               // print () базового класса (из другой области видимости)
    // полностью унаследован и может использоваться селектор get_name ()
    virtual  ~student2c () { delete [] pract2; delete [] tutor2; }
};
void student :: print ()
{ cout << "ФИО            = " << name       << endl;
  cout << "Курс           = " << year       << endl;
  cout << "Средний балл   = " << avb        << endl;
  cout << "Номер зачётки  = " << student_id << endl;
}
void student2c :: print ()
{ student :: print (); // выдаёт в файл name, year, avb, student_id
  cout << "Тема курсовой = " << pract2     << endl;
  cout << "Преподаватель = " << tutor2     << endl;
}
class student3c: protected student   { ... };
class student4c: private   student   { ... };
class student2cSPdep:      student2c { ... };
```

Множественное наследование




```
class student5c: student              { ... };
class student4y: public student { // четырёхлетнее обучение
     protected: char* certificate;    ... };
class student5y: public student { // пятилетнее обучение
     protected: char* diploma;        ... };
class student4p: public student2c, public student4y { protected: bool test;
                                      ... };
class student5p: public student2c, public student5y { protected: int exam;
                                      ... };
```

Виртуальное множественное наследование




```
class student                           { ... protected: double avb;       ... };
class student2c: virtual public student { ... protected: char * pract2;    ... };
class student3c: virtual public student { ... protected:                   ... };
class student4y: virtual public student { ... protected: char * certificate;... };
class student5y: virtual public student { ... protected: char * diploma;   ... };
class student4p: virtual public student2c, virtual public student4y
                                        {     protected: bool   test;      ... };
class student5p: virtual public student2c, virtual public student5y
                                        { ... protected: int    exam;      ... };

      void fst ()
      { student    s (“Катя”, 2, 4.18, 20050210); // базовый конструктор
        student2c ds (“Таня”, 4.08, 20050211,     // производный - // -
                      “Компилятор Си++”, “Виктор Петрович”);
        student   * ps = &  s; // указатель на базовый класс, хотя курс = 2
        student2c *pds = & ds; // указатель на производный класс
        ps  -> print (); // student   :: print (); // напечатается главное
        pds -> print (); // student2c :: print (); // напечатается всё
        ps = pds;        // допустимо (будет стандартное преобразование)
        ps -> print ();  // student :: print () – функция выбирается
                         // статически по типу указателя.
                         // Если же в определении класса student добавлено слово
                         // virtual, будет вызвана динамически выбираемая
                         // функция student2c :: print ()
      }
```

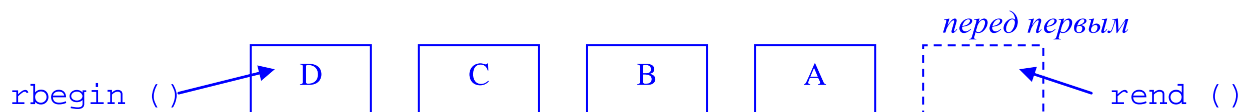
Операции над контейнерами с различными итераторами

Итераторы	Чтение	Доступ	Запись	Изменение	Сравнение
Вывода			*p=e	p++ ++p	
Ввода	x=*p	p->f		p++ ++p	p==q p!=q
Однонаправленные	x=*p	p->f	*p=e	p++ ++p	P==q p!=q
Двунаправленные	x=*p	p->f	*p=e	p++ ++p p-- --p	P==q p!=q
Произвольный доступ	x=*p	p->f p[n]	*p=e	p++ ++p p-- --p p+n n+p p-n p-q p+=n p-=n	p==q p!=q p<q p>q p>=q p<=q

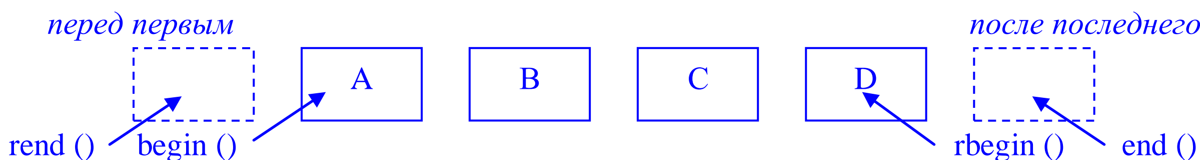
Последовательный доступ к элементам данных контейнерных типов осуществляется от первого элемента к последнему:

```
template<class C> typename C::value_type sum (const C& c)
{   typename C::value_type s = 0;
    typename C::const_iterator p = c.begin ();
    while (p != c.end ()) s += * (p ++);
}
```

С помощью обратных итераторов последовательный доступ к элементам данных контейнерных типов осуществляется от последнего элемента к первому:




```
template<class C> typename C::value_type sum (const C& c)
{   typename C::value_type s = 0;
    typename C::const_reverse_iterator p = c.rbegin ();
    while (p != c.rend ()) s += * (p ++);
}
```



Использование адаптеров контейнеров:

```
stack<vector<int>>           // стек целых чисел на базе вектора
queue<deque<char>>           // очередь символов на базе двойной очереди
```

Использование адаптеров функциональных объектов:

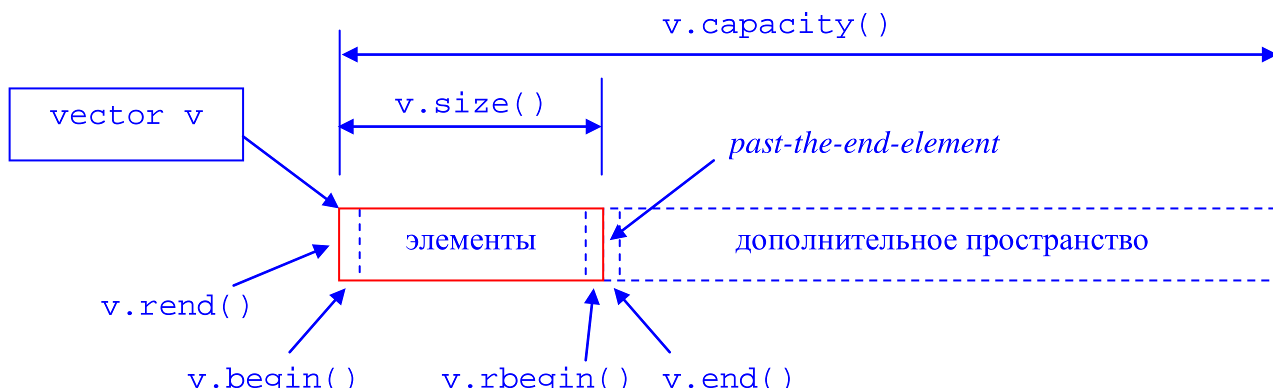
```
void f (list<int> & c)
{   list<int>::iterator p = find_if (c.begin (), c.end (),
                                     bind2nd (less<int> (), 21));
}
```

Эта программа эффективнее, чем использование неадаптированного функционального объекта:

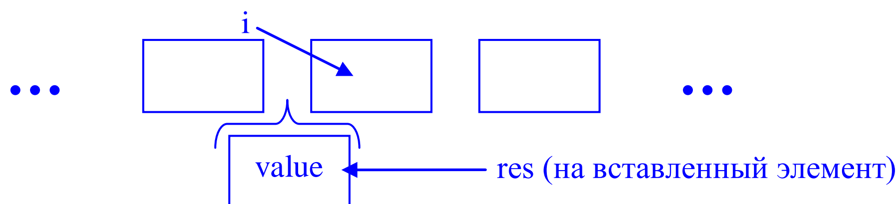
```
bool less_than_21 (int v) { return v < 21; }
void f (list<int> & c)
{   list<int>::iterator p = find_if (c.begin (), c.end (), less_than_21);
}
```

Векторы, строящиеся на основе контейнеров класса **`vector`*****:***

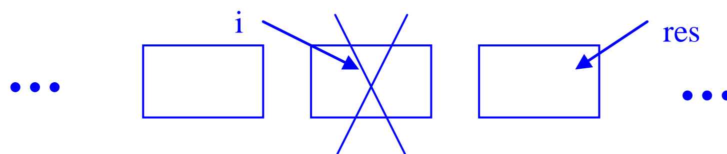
```
#include <vector>      // определено для всех контейнеров STL (vector, list, ...)
using namespace std;   // все стандартные контейнеры определены в стандартном
                       // пространстве именования (std)
template<class T, class A = allocator<T>> class vector;
//  по умолчанию используется распределитель памяти из класса,
//  к которому относятся элементы контейнера
vector& operator = (const vector <T, A> & obj);
vector (const vector <T, A> & obj);  // конструктор копирования

// инициализация вектора выборочным копированием элементов из [first, last)
// It - итератор для чтения
vector (It first, It last, const A& = A());

// Конструкторы, которые могут вызываться с одним параметром, во избежание
// случайного преобразования объявлены как явные (explicit). Это означает,
// что конструктор может вызываться только явно.
// (vector<int>v = 10 - ошибка, попытка неявного преобразования 10 в vector<T>)
explicit vector (const A& = A ());
                           // требуется явный вызов конструктора: vector<T> x(10);
                           //       неявный вызов: vector<T> y = 10 (неправильно);
explicit vector (size_type size, const T& value = T (),
                 const A& a = A ()); // заводятся сразу несколько элементов со
            // значениями, которые даются их конструкторами по умолчанию.
            // Если второй параметр отсутствует, конструктор умолчания в T
            // обязателен. Есть и другие виды конструкторов.
            // Имеются также методы, связанные с итераторами:
    // iterator begin (); const_iterator begin () const;
    // iterator   end (); const_iterator   end () const;
```



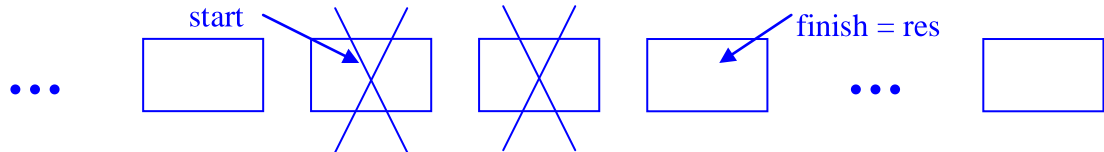

```
iterator insert (iterator i, const T& value) {...}      // вставка перед элементом
```




```
iterator insert (iterator i, size_type number, const T & value){...}
                        // вставка нескольких одинаковых элементов перед элементом
iterator erase (iterator i) { ... return (i); }// уничтожение заданного элемента и
                            //  выдача итератора элемента, следующего за удалённым
```



```
iterator erase (iterator start, iterator finish)         // уничтожение диапазона
{ ... return (finish);}//[start,finish) и выдача следующего за последним удалённым
```




```
bool empty            () const {...}              //истина, если контейнер пуст
size_type size        () const {...}              //выдача текущего размера
void clear            () {erase (begin(), end());}//уничтожение всех элементов, при
  // этом память не освобождается, так как деструктор самого вектора не вызывается
void push_back (const T&value) {insert(end(),value);} //вставка в конец контейнера
void pop_back         () {erase (end() - 1);     }//уничтожение последнего элемента

reference operator [](size_type i) { return * (begin () + i); }
                                               // reference – аналог & в Си++
reference front () { return * begin (); }      // содержимое первого элемента
reference  back () { return *(end () - 1); }   // содержимое последнего элемента
reference at (size_type i) { ... }             // содержимое элемента с номером i
```

<u>При работе с функцией at( )используется перехватчик исключительной ситуации:</u>

```
try                  { ... v.at (i}; ... }
catch (out_of_range) {        ...        }
```

<u>Обход вектора прямым и обратным итератором:</u>

```
      int main ()
      { vector<int> v (100, 5); // 100 элементов, инициированных значением 5
        vector<int>::const_iterator p = v.begin ();
        vector<int>::const_reverse_iterator q = v.rbegin ();      ...
        while (p != v.end  ())      { cout << * p << ' '; ++ p; } ...
        while (q != v.rend ())      { cout << * q << ' '; ++ q; } ...
        return 0;
      }
```

<u>Учебная программа, иллюстрирующая особенности работы с векторами</u>

```
#include <vector>
using namespace std;
typedef vector<int> Container;
typedef Container::size_type Cst;
void f (Container& v, int i1, int i2) {
  try { for (Cst i = 0; i < 10; i++)
        { // здесь значение индекса i проверять не надо: вектор создаётся заново
          v.push_back (i);     // Элементы: 0, 1, 2, ..., 9.
        }
        v.at (i1) = v.at (i2);          // проверка правильности индексов i1 и i2
                                        // выполняется внутри функции at ()
        cout << v.size ();              // Размер контейнера для данной точки
        Container::iterator p = v.begin ();
        p += 2;              // Для векторов это можно, для других – advance (p, 2)
        v.insert (p, 100);   // Элементы: 0, 1, 100, 2, ..., 9.   p теряет значение
        sort (v.begin (), v.end ());    // Сортировка диапазона
        for (Cst i = 0; i < v.size (); i++)
          { // здесь значение индекса i уже проверено, можно пользоваться
  // непосредственно индексацией v [i], даже если число элементов в цикле меняется
            cout << v[i];
          }
      }
  catch (out_of_range) { ... } // реакция на ошибочный индекс
}
int main () { Container v; f (v, 5, 12); }
```

Списки, строящиеся на основе стандартного контейнера `list`, (уровень разрешённого итератора – двунаправленный):

```
#include <list>        // определено для всех контейнеров STL (vector, list, ...)
using namespace std;   // все стандартные контейнеры определены в стандартном
                       // пространстве именования (std)
template<class T, class A = allocator<T>> class list;
list& operator = (const list <T, A> & obj);
list (const list <T, A> & obj);  // конструктор копирования
// инициализация списка копированием элементов из [first, last)
// It - итератор для чтения
list (It first, It last, const A& = A());
// Конструкторы, которые могут вызываться с одним параметром, во избежание
// случайного преобразования объявлены как явные (explicit). Это означает,
// что конструктор может вызываться только явно.
// (list<int>l = 10 - ошибка, попытка неявного преобразования 10 в list<int>)
explicit list (const A& = A ()); // явный вызов конструктора: list<T> x(10);
explicit list (size_type size, const T& value = T (),
                 const A& a = A ()); // заводятся сразу несколько элементов со
            // значениями, которые даются их конструкторами по умолчанию.
            // Если второй параметр отсутствует, конструктор умолчания в T
            // обязателен. Есть и другие виды конструкторов.
            // Имеются также методы, связанные с итераторами:
    // iterator begin (), end (); const_iterator begin () const, end () const;
```

l.size()

list l

past-the-end-element

элемент ⇄ элемент ⇄ элемент ⇄ элемент

l.rend()

l.begin()

l.rbegin()

l.end()

```
void push_front(const T& value){ insert (begin (), value); } // вставка в начало
void  pop_front            (){ erase  (begin ());}//уничтожение первого элемента
```

Контейнеры в библиотеке STL построены таким образом, что фрагмент учебной программы для работы с векторами, показанный ранее, может легко быть приспособлен для работы со списками. Для этого надо всюду заменить слово *vector* словом *list*, а также внимательно переработать те операторы, в которых существенно используются свойства итераторов произвольного доступа, использованные в программе (были выделены шрифтом с подчеркиванием). Например, выдача данных в поток записывается не так, как для векторов (с помощью операции индексации), а иначе:

```
for (int i = 0; i < v.size (); i++)      cout << v[i]; // итерация по вектору
                                   //          не годится для работы со списком
for (p = v.begin (); p != v.end (); ++p) cout << * p;  // итерация по списку
                                   //        работоспособно также и для вектора
```

Схема классической системы программирования

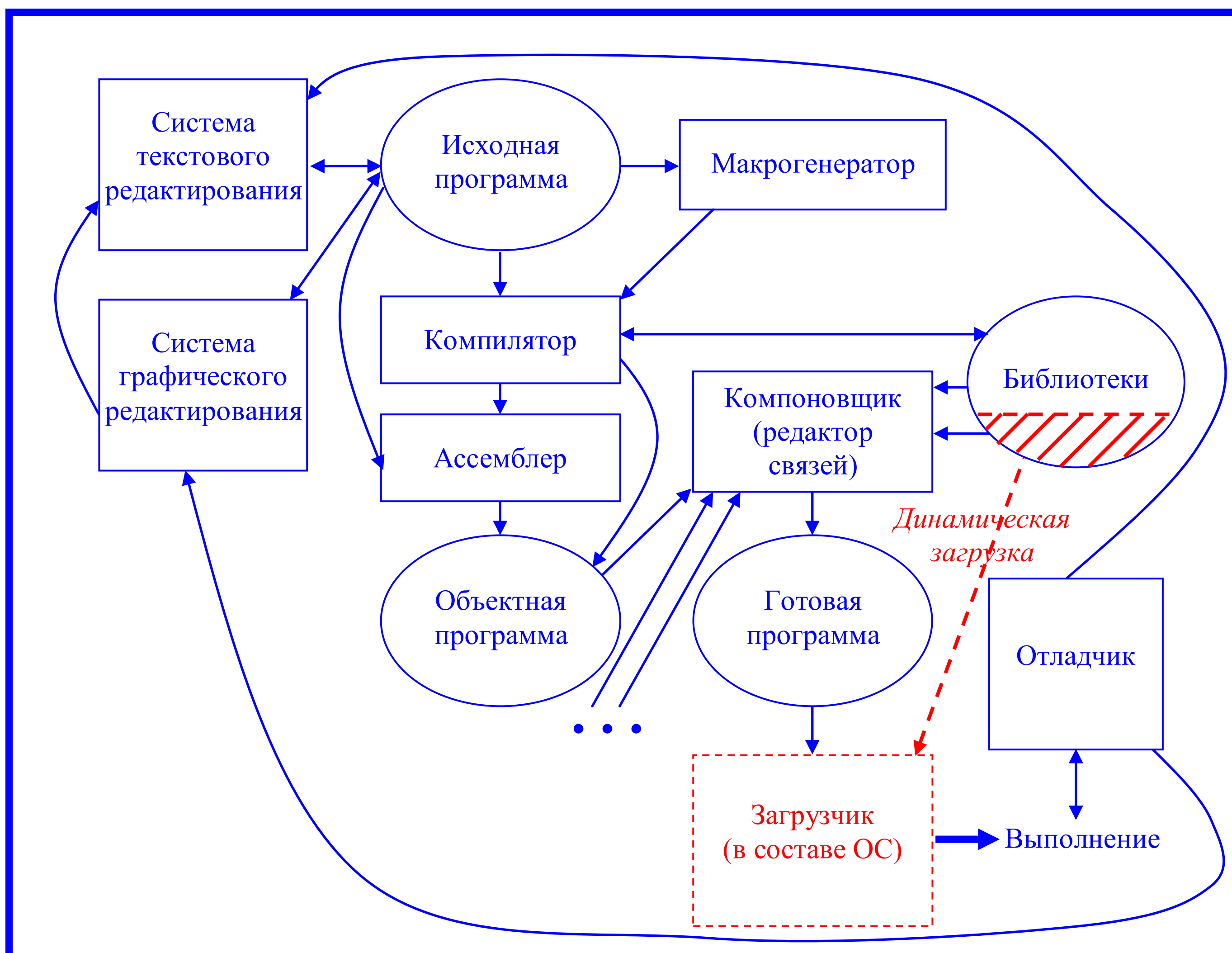


Схема работы компилятора языка программирования
(сплошные стрелки указывают порядок работы составных частей компилятора, пунктирные линии отображают потоки информации).

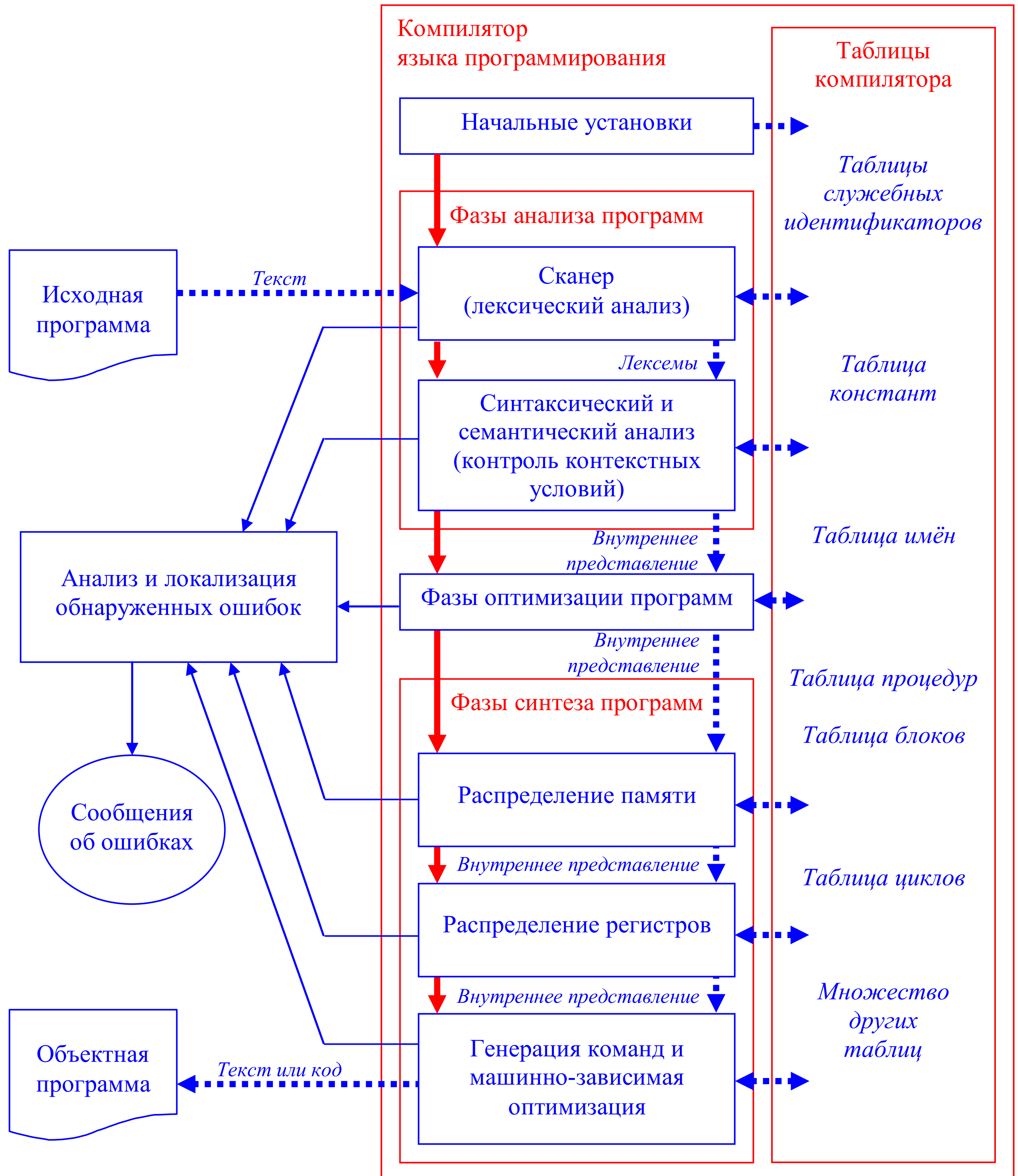


Схема работы однопроходного компилятора языка программирования

Описание модельного языка. Правила грамматики:

```
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    →  C | IC | IR
N    →  R | NR
C    →  a | b | ... | z | A | B | ... | Z
R    →  0 | 1 | 2 | ... | 9
```

Замечания:

a) запись вида {α} означает итерацию цепочки α (повторение ее 0 или более раз), то есть в порождаемой цепочке в этом месте может находиться либо ε, либо α, либо αα, либо ααα и т.д.

b) запись вида [α | β] означает, что в порождаемой цепочке в этом месте может находиться либо α, либо β.

c) P – цель грамматики; символ ⊥ – маркер конца текста программы.

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 операций отношения должны быть целочисленными.
5. 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 ⊥ >}. Вложенные комментарии запрещены.

Идентификаторы **true**, **false**, **read**, **write** и другие, упомянутые среди правил грамматики, – служебные слова (их нельзя переопределять, как стандартные идентификаторы Паскаля). Эти идентификаторы не имеют никакого отношения к буквам, из которых они составлены.

Сохраняется правило языка Паскаль о разделителях между идентификаторами, числами и служебными словами.

Действия, выполняемые лексическим анализатором (сканером):

1. **GetS** – ввод очередного символа исходной программы.
2. **clear** – инициализация буфера ввода символов лексемы.
3. **add** – добавление очередного символа к лексеме в буфере.
4. **get_token** – поиск лексем в таблицах служебных слов (*TW*), ограничителей и знаков операций (*TD*).
5. **get_object** – поиск объектов в таблице имён *TI*, либо в таблице констант *TC*.
6. **new Ident** – создание нового объекта "Идентификатор".
7. **new Number** – создание нового объекта "Константа".
8. **put_object** – занесение информации о вновь созданных объектах (константах или идентификаторах) в таблицу констант *TC*, либо в таблицу имён *TI*.
9. **new Token** – создание новой лексемы при вводе константы или идентификатора (указатель на объект в этой лексеме может быть новым, а может извлекаться из соответствующей таблицы)

Состояния лексического анализатора:

1. **H** – начальное состояние
2. **Comment** – ввод комментария (примечания)
3. **NOperator** – ввод знака операции отрицания
4. **Operator** – ввод односимвольного знака операции
5. **LOperator** – ввод двухсимвольного знака операции
6. **Identifier** – ввод идентификатора
7. **Literal** – ввод целочисленной константы
8. **Error** – ошибка

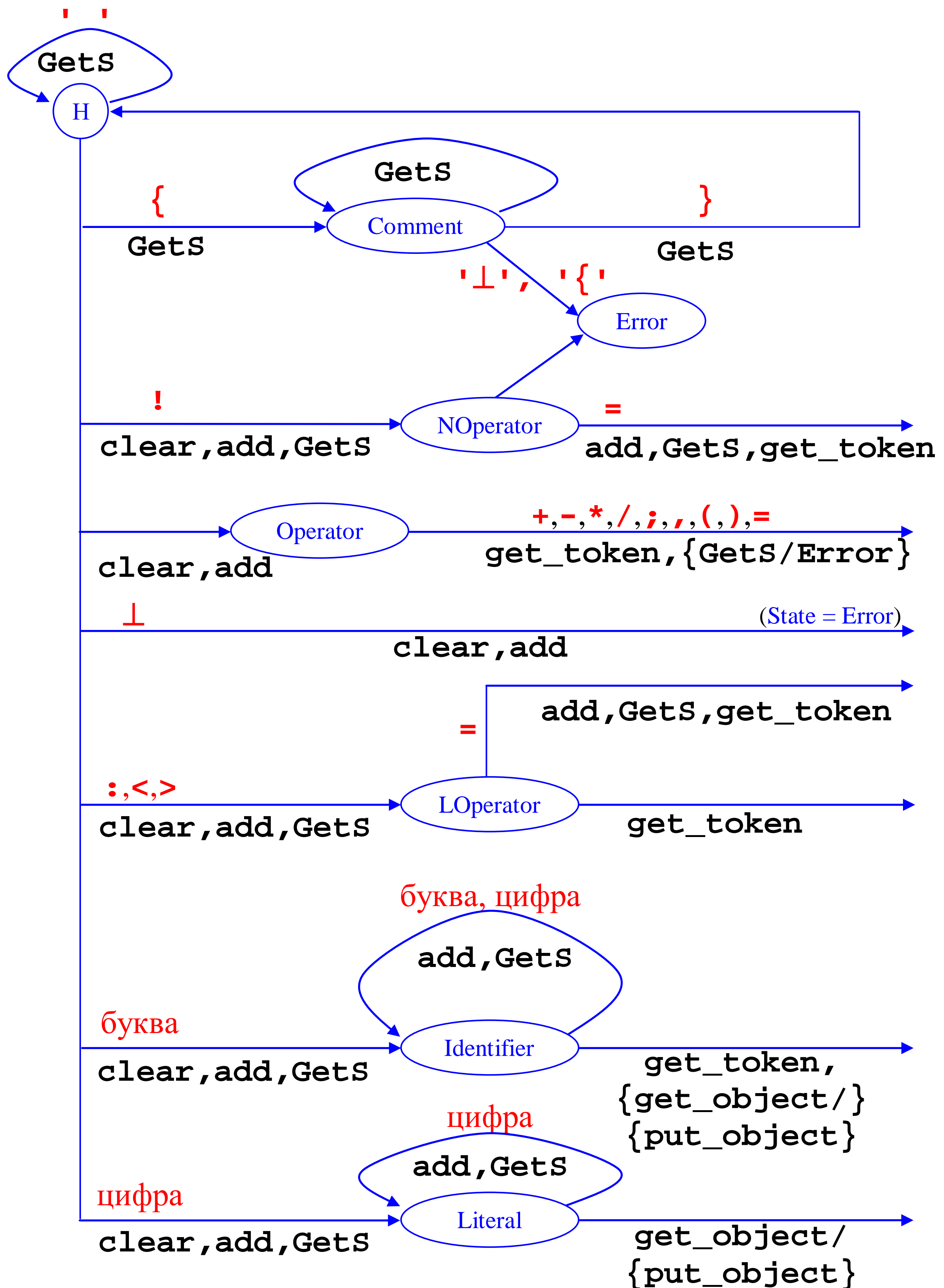


Диаграмма состояний лексического анализатора (сканера) модельного языка.
*Все выходные стрелки предполагают последующий переход в начальное состояние (**H**).*
*После чтения конца файла осуществляется переход в состояние **Error** без выдачи сообщения об ошибке (ошибка выдается лишь при повторной попытке чтения).*
Непомеченные дуги соответствуют переходам по символам, не надписанным ни над одной дугой, исходящей из этой же вершины.

Действия, выполняемые при семантическом контроле контекстных условий:

1. **reset** – очистка стека
2. **push** – занесение нового элемента в стек имен или типов
3. **pop** – чтение текущего элемента из стека
4. **decl** – занесение в таблицу информации о новом идентификаторе
5. **check_op** – проверка совпадения типов двух операндов бинарной операции
6. **check_not** – проверка типа операнда унарной операции отрицания
7. **check_id** – контроль наличия описания идентификатора
8. **eq_type** – сравнение типов двух операндов из стека
9. **check_id_in_read** – контроль наличия описания идентификатора в операторе чтения

Грамматика модельного языка с семантическими действиями:

P → **program** D1 ; B⊥

D1 → **var** D { , D}

D → < reset () > I < push (name) > {, I < push (name) >}:
[**int** < decl ("**int**") > | **bool** < decl ("**bool**") >]

B → **begin** S { ; S} **end**

S → I < check_id () > := E < eq_type () >
if E < eq_type ("**bool**") > **then** S **else** S |
while E < eq_type ("**bool**") > **do** S |
B |
read (I <check_id_in_read ()>) |
write (E)

E → E1 | E1 [= | < | > | <= | >= | !=] < push (type) > E1 < check_op () >

E1 → T {[+ | – | **or**] < push (type) > T < check_op () >}

T → F {[* | / | **and**] < push (type) > F <check_op () >}

F → I < check_id () > | N < push (type) > | L < push (type) > |
not F < check_not () > | (E)

L → **true** | **false**

I → C | IC | IR

N → R | NR

C → a | b | ... | z | A | B | ... | Z

R → 0 | 1 | 2 | ... | 9

Действия, выполняемые при генерации внутреннего представления программы:

1. **put** – занесение элемента в массив ПОЛИЗ, варианты:
   нет параметров: резервирование места в массиве ПОЛИЗ
   один параметр: запись элемента в конец массива ПОЛИЗ
   два параметра: запись элемента в указанное место массива
2. **place** – определение номера свободного элемента в массиве ПОЛИЗ

Некоторые правила грамматики модельного языка с действиями по генерации ПОЛИЗ:

F	→	I <check_id (); Put (**I**)> | N <Push (type = **int**); Put (**N**)> |		
		L <Push (type = **bool**); Put (**L**)> | **not** F <check_not ()> | (E)		
S	→	I <check_id (); Put (&I)> **:=** E <eq_type (); Put (“**:=**”)>		
S	→	**if** E <eq_type (**bool**); Place (lab1); Put (); Put (“!F”)>		
		then	S	<Place (lab2); Put (); Put (“!”); Put (Place, lab1)>
		else	S	<Put (Place, lab2)>
S	→	**while** <Place (lab1)> E <eq_type (**bool**); Place (lab2); Put (); Put (“!F”)>		
		do S <Put (lab1); Put (“!”); Put (Place, lab2)>		

```
#include <iostream.h>
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <typeinfo>
using namespace std;

//=================================================================

//////////////////////  Отладочные режимы  ////////////////////////

//=================================================================

static unsigned int Dump = 0;
#define DUMP_ANY (unsigned int) - 1
#define DUMP_BUF (unsigned int)          1
#define DUMP_REC (unsigned int)          2
#define DUMP_LEX (unsigned int)          4
#define DUMP_PLZ (unsigned int)          8
#define DUMP_PTA (unsigned int)         16
#define DUMP_PTC (unsigned int)         32
#define DUMP_PTI (unsigned int)         64
#define DUMP_PTL (unsigned int)        128
#define DUMP_SIM (unsigned int)        256

//=================================================================

/////////////////////  Список лексем языка  ///////////////////////

//=================================================================

enum type_of_lex {          LEX_NULL,   LEX_PROGRAM,
   LEX_VAR,    LEX_BOOL,   LEX_INT,    LEX_FALSE,   LEX_TRUE,
   LEX_BEGIN,  LEX_END,    LEX_ASSIGN,
   LEX_IF,     LEX_THEN,   LEX_ELSE,   LEX_WHILE,
   LEX_DO,     LEX_READ,   LEX_WRITE,
   LEX_AND,    LEX_NOT,    LEX_OR,     LEX_LT,
   LEX_LE,     LEX_EQ,     LEX_NE,     LEX_GE,      LEX_GT,
   LEX_DIV,    LEX_PLUS,   LEX_MINUS,  LEX_MULT,    LEX_LPAREN, LEX_RPAREN,
   LEX_COMMA,  LEX_COLON,  LEX_SEMICOLON,
   LEX_ID,     LEX_NUM,    LEX_FIN,
   PLZ_GO,                 PLZ_FGO     };
```

```
//======================================================================

////////////////  Таблицы внешних представлений  ////////////////////

//======================================================================

static char *        KeyWords [] = { "",          "program",   "begin",   "end",     "var",     "int",      "bool",
                                     "true",      "false",
                                     "if",        "then",      "else",    "do",      "while",   "read",
                                     "write",     "not",       "or",      "and",     " go_to",  " go_if_not",     0};
static type_of_lex  LKeyWords [] = { LEX_NULL,   LEX_PROGRAM, LEX_BEGIN, LEX_END,   LEX_VAR,    LEX_INT,     LEX_BOOL,
                                     LEX_TRUE,   LEX_FALSE,
                                     LEX_IF,     LEX_THEN,    LEX_ELSE,  LEX_DO,    LEX_WHILE,  LEX_READ,
                                     LEX_WRITE,  LEX_NOT,     LEX_OR,    LEX_AND,   PLZ_GO,     PLZ_FGO,     LEX_NULL};
static char *      Delimiters [] = { "",         "@",         ",",       ":",       ":=",       ";",
                                     "<",        "<=",        "=",       "!=",      ">=",       ">",
                                     "+",        "-",         "*",       "/",       "(",        ")",               0};
static type_of_lex LDelimiters []= { LEX_NULL,   LEX_FIN,     LEX_COMMA, LEX_COLON, LEX_ASSIGN, LEX_SEMICOLON,
                                     LEX_LT,     LEX_LE,      LEX_EQ,    LEX_NE,    LEX_GE,     LEX_GT,
                                     LEX_PLUS,   LEX_MINUS,   LEX_MULT,  LEX_DIV,   LEX_LPAREN, LEX_RPAREN,  LEX_NULL};

//======================================================================

/////////////////// Интерфейсы используемых классов ////////////////

//======================================================================

//======================================================================

/////////////////////////  Класс buf  //// (для сборки лексем) //////

//======================================================================

class buf {    char * b; int size;  int top;
      public:  buf (int max_size = 260);
             ~ buf                  ();
               void   clear         ();
               void   add(const char c);
               char * get_string    () const;
               int    int_buf       () const;
               friend ostream & operator << (ostream &s, const buf & b);
};
```

```
                        class Token;
                        class ProgramObject;
template <class Object> class ObjectTable;

                        ostream & operator << (ostream & s, const Token * t);
template <class Object> ostream & operator << (ostream & s, const ObjectTable<Object> & l);
                        ostream & operator << (ostream & s, const ProgramObject * t);

//===================================================================

///////////////////////  Класс таблиц лексем  /////////////////////

//===================================================================

class TokenTable { Token ** p; char ** c; int size;
      public:  TokenTable (int max_size, char * data [], type_of_lex t []);
             ~ TokenTable ();
               Token *        operator ()       (int k);
               char  *        operator []       (int k);
               int     get_size                       () const;
               int     get_index       (const Token * l) const;
               int     get_index   (const type_of_lex l) const;
               Token * get_token         (const buf & b) const;
               void    put_obj (ProgramObject *t, int i);
        friend ostream & operator << (ostream & s, const TokenTable & t);
};

//===================================================================

////////////////////////  Класс лексем  /////////////////////////////

//===================================================================

class Token          { type_of_lex type;   ProgramObject * value;
      public:  Token (const type_of_lex t, ProgramObject * v = 0);
               type_of_lex get_type                   () const;
               void        set_type(const type_of_lex t);
               void        set_value (ProgramObject * v);
           ProgramObject * get_value                  () const;
        friend ostream & operator << (ostream & s, const Token * t);
};
```

```
//====================================================================

/////////////////  Класс таблиц объектов программы  ////////////////

//====================================================================

template <class Object> class ObjectTable { int size; public: Object ** p; int free;
      public:  ObjectTable         (int max_size);
             ~ ObjectTable                    ();
               Object *        operator[] (int k);
               Object *put_obj (Object *t    = 0);
               Object *put_obj (Object *t, int i);
               int   get_place                 () const;
               int   get_index        (Token * l) const;
               int   get_index    (type_of_lex t) const;
               int   get_index            (int v) const;
               Object * get_object(const buf & b) const;
        friend ostream & operator << (ostream &s, const ObjectTable & t);
};
```

```
//====================================================================

////////////////  Базовый класс объектов программы  ////////////////

//====================================================================

class ProgramObject { protected: type_of_lex type; int value;
      public:  type_of_lex  get_type                  () const;
               void         set_type     (type_of_lex t);
               int          get_value                 () const;
               void         set_value            (int v);
       virtual void         exec              (int & i) const = 0;
       virtual bool         is_object    (const buf & b) const = 0;
       virtual ostream &    print          (ostream & s) const = 0;
};
```

```
//==================================================================

////////////// Производные классы объектов программы  //////////////

//==================================================================

class Ident:public ProgramObject { char * name; bool declare; bool assign;
      public:  Ident                     (const buf & b);
               bool   get_assign                      () const;
               void   set_assign                      ();
               bool   get_declare                     () const;
               void   set_declare                     ();
               char * get_name                        () const;
               bool   is_object          (const buf & b) const;
               ostream & print             (ostream & s) const;
               void exec                         (int &) const;
};
//==================================================================

class Number:public ProgramObject {
      public:  Number                    (const buf & b);
               bool   is_object          (const buf & b) const;
               ostream & print             (ostream & s) const;
               void exec                       (int & i) const;
};
//==================================================================

class Label:public ProgramObject {
      public:  Label                             (int n);
               bool   is_object          (const buf & b) const;
               ostream & print             (ostream & s) const;
               void exec                       (int & i) const;
};
//==================================================================

class Address: public ProgramObject {
      public:  Address                           (int n);
               bool   is_object          (const buf & b) const;
               ostream & print             (ostream & s) const;
               void exec                       (int & i) const;
};

//==================================================================
```

```
class Operation:public ProgramObject { char * sign;
          protected: Operation (char * str, type_of_lex t);
             public:  bool   is_object            (const buf & b) const;
                   ostream & print                  (ostream & s) const;           virtual void exec (int & i) const = 0;
};
class    TrueObject: public Operation { public:    TrueObject (char * str, type_of_lex t); void exec (int &)   const;  };
class   FalseObject: public Operation { public:   FalseObject (char * str, type_of_lex t); void exec (int &)   const;  };
class     NotObject: public Operation { public:     NotObject (char * str, type_of_lex t); void exec (int &)   const;  };
class      OrObject: public Operation { public:      OrObject (char * str, type_of_lex t); void exec (int &)   const;  };
class     AndObject: public Operation { public:     AndObject (char * str, type_of_lex t); void exec (int &)   const;  };
class      EqObject: public Operation { public:      EqObject (char * str, type_of_lex t); void exec (int &)   const;  };
class      LtObject: public Operation { public:      LtObject (char * str, type_of_lex t); void exec (int &)   const;  };
class      GtObject: public Operation { public:      GtObject (char * str, type_of_lex t); void exec (int &)   const;  };
class      LeObject: public Operation { public:      LeObject (char * str, type_of_lex t); void exec (int &)   const;  };
class      GeObject: public Operation { public:      GeObject (char * str, type_of_lex t); void exec (int &)   const;  };
class      NeObject: public Operation { public:      NeObject (char * str, type_of_lex t); void exec (int &)   const;  };
class    PlusObject: public Operation { public:    PlusObject (char * str, type_of_lex t); void exec (int &)   const;  };
class   MinusObject: public Operation { public:   MinusObject (char * str, type_of_lex t); void exec (int &)   const;  };
class    MultObject: public Operation { public:    MultObject (char * str, type_of_lex t); void exec (int &)   const;  };
class     DivObject: public Operation { public:     DivObject (char * str, type_of_lex t); void exec (int &)   const;  };
class  AssignObject: public Operation { public:  AssignObject (char * str, type_of_lex t); void exec (int &)   const;  };
class    GoToObject: public Operation { public:    GoToObject (char * str, type_of_lex t); void exec (int & i) const;  };
class GoIfNotObject: public Operation { public: GoIfNotObject (char * str, type_of_lex t); void exec (int & i) const;  };
class   WriteObject: public Operation { public:   WriteObject (char * str, type_of_lex t); void exec (int &)   const;  };
class    ReadObject: public Operation { public:    ReadObject (char * str, type_of_lex t); void exec (int &)   const;  };

//=====================================================================

/////////////////////////  Класс Scanner  ///////////////////////////

//=====================================================================

class Scanner { enum State { H, Comment, NOperator, Operator, LOperator, Identifier, Literal, Error };
                     State FA_State;     FILE * fp; char c;               buf b;
void GetS                           ();  // Процедура ввода очередного символа программы

//////////////////// Основные процедуры сканера ////////////////////

public:            Scanner                   (char *);  // Конструктор класса
                  ~Scanner                         ();  // Деструктор класса
        Ident    *  CreateIdentObject  (const buf & b);     Number   *  CreateNumberObject (const buf & b);
        Token    *  get_lex                        ();  // Процедура ввода очередной лексемы
};
```

```
//=====================================================================

/////////////////////////  Класс Parser  ///////////////////////////

//=====================================================================

class Parser { Token * curr_lex;  Scanner scan; type_of_lex c_type;
void GetL                            ();  // Процедура ввода очередной лексемы и установки внутренних переменных

/////////////////// Процедуры рекурсивного спуска //////////////////

void P (); void D1 (); void D (); void B (); void S (); void E (); void E1 (); void T (); void F ();

////////////////// Процедуры семантического контроля ///////////////

void decl             (type_of_lex type) const;
void check_op                        () const;  void check_not                        () const;
void eq_type                         () const;  void eq_type          (type_of_lex type) const;
void check_id                        () const;  void check_id_in_read                 () const;

/////////////////// Вспомогательные процедуры //////////////////////

Address * CreateAddressObject (Token * l);
Label   * CreateLabelObject   (int label);


/////////// Основные процедуры синтаксического анализа ////////////

public: Parser            (char *program);  // Конструктор класса
       ~Parser                        ();  // Деструктор класса
        void Analyze                  ();  // Основная процедура анализа
};

//=====================================================================

///////////////////////  Класс Simulator  //////////////////////////

//=====================================================================

class Simulator { ProgramObject * curr_obj;
      public:           void Simulate  ();  // Основная процедура интерпретатора
                           ~ Simulator ();  // Деструктор класса
};
```

```
//======================================================================

////////////////////////// Шаблоны стеков ////////////////////////////

//======================================================================

template <class T, int max_size> class Stack { T s [max_size]; int top;
public: Stack         ()       {                        reset ();     }
        void reset    ()       {                         top = 0;     }
        void push  (T i)       { if (!is_full  ()) { s [top ++] = i;  }
                                 else throw "Стек переполнен";        }
        T    pop      ()       { if (!is_empty ()) { return s [--top];}
                                 else throw "Стек исчерпан";          }
        bool is_empty () const { return top <= 0;                     }
        bool is_full  () const { return top >= max_size;              }
};

//======================================================================

////////////////////////     Стеки     ////////////////////////////////

//======================================================================

    Stack     <Token *, 100>  Names;
    Stack <type_of_lex, 100>  Types;
    Stack         <int, 100> Values;

//======================================================================

////////////////  Таблицы внутренних представлений  ////////////////

//======================================================================

    TokenTable  TW  (sizeof   (KeyWords) / sizeof   (KeyWords [0]), KeyWords,   LKeyWords);
    TokenTable  TD  (sizeof (Delimiters) / sizeof (Delimiters [0]), Delimiters, LDelimiters);
    ObjectTable<Ident>          TI                            (100);
    ObjectTable<Address>        TA (sizeof (TI) / sizeof (TI [0]));
    ObjectTable<Number>         TC                            (100);
    ObjectTable<Label>          TL                            (100);
    ObjectTable<Operation>      TO                             (20); // Таблица для 20 операционных объектов
    ObjectTable<ProgramObject> PLZ                           (1000);
    int                         ind_GO,       ind_FGO;
```

```
//=====================================================================

/////////////////// Реализация используемых классов ////////////////

//////////// Отладочные операторы выдачи лексем и объектов //////////

//=====================================================================

ostream & operator << (ostream & s, const Token * t)
{ ProgramObject * p; int i;
  s << "(Тип = ";
  s.width (2); s << t -> type << ")";
  if  ((p = t -> get_value ()) != 0) { s << " "; p -> print (s); }
  else { for (i = 0; i < sizeof  (KeyWords) / sizeof   (KeyWords [0]); i ++)
             if (  LKeyWords [i] == t -> type) { s << " Слово " <<   KeyWords [i] << endl; return s; }
         for (i = 0; i < sizeof(Delimiters) / sizeof (Delimiters [0]); i ++)
             if (LDelimiters [i] == t -> type) { s << " Знак  " << Delimiters [i] << endl; return s; }
         s << endl;
       }
  return s;
}
template <class Object> ostream & operator << (ostream & s, const ObjectTable<Object> & l)
{ for (int i = 0; i < l.free; i ++) { cout.width (4); cout << i << ": " << l.p [i];}
  return s;
}
ostream & operator << (ostream & s, const ProgramObject * t) { t -> print (s); return s; }

//=====================================================================

//////////////////////////  Класс buf  ////////////////////////////

//=====================================================================

                buf::buf (int max_size)
                       { b = new char  [size = max_size]; top = 0; clear(); }
                buf::~ buf      ()       { delete b;                          }
         void   buf::clear      ()       {
              if (Dump & DUMP_BUF && top) { cout << "Буфер: " << b << endl; }
                                           memset (b, '\0', size); top = 0; }
         void   buf::add        (  const char c) { b [top ++] = c;          }
         int    buf::int_buf    () const { return atoi (b);                 }
         char * buf::get_string () const { return b;                        }
     ostream &operator << (ostream &s, const buf & b) { s << "Буфер = " << b.b << endl; return s;       }
```

```
//====================================================================

//////////////////////  Класс таблиц лексем  //////////////////////

//====================================================================

       TokenTable::~ TokenTable                         () { delete []  p; }
Token* TokenTable::    operator ()                 (int k) { return p [k]; }
char * TokenTable::    operator []                 (int k) { return c [k]; }
void   TokenTable::    put_obj (ProgramObject * t, int i) { p [i] -> set_value (t); }
       TokenTable::  TokenTable (int max_size, char * data [], type_of_lex t [])
{ p = new Token * [(size = max_size) + 1]; c = data;
  for (int i  = 0; i  < size; i ++)   { p [i] = 0; if (!data [i])  break;
                                        p [i] = new Token (t [i]);
                                      }
}
int TokenTable::get_size () const { return size; }
int TokenTable::get_index (const type_of_lex l) const
{ Token ** q = p; Token * t; int i = 0;
  while (* q)
      { t = * q ++;
        if (t -> get_type () == l) break;
        i ++;
      }
  return i;
}
int TokenTable::get_index (const Token * l) const
{ Token ** q = p; int i = 0;
  while (* q != l) { q ++; i ++; }
  return i;
}
Token * TokenTable::get_token (const buf & b) const
{ Token ** q = p; char ** s = c; Token * t;
  while (* q)
      { t = * q ++;
        if (! strcmp (b.get_string (), * s ++)) return t;
      }
  return 0;
}
ostream & operator << (ostream & s, const TokenTable & l)
{ for (int i = 0; i < l.size; i ++) cout << l.p [i];
  return s;
}
```

```
//=====================================================================

///////////////////////////  Класс лексем  /////////////////////////

//=====================================================================

                Token::Token  (const type_of_lex t, ProgramObject * v) {       value = v; type = t; }
  type_of_lex   Token::get_type                  () const               { return           type;     }
         void   Token::set_type     (type_of_lex t)                     {                  type = t; }
ProgramObject * Token::get_value                 () const               { return value;              }
         void   Token::set_value(ProgramObject * v)                     {       value = v;           }

//=====================================================================

/////////////////  Класс таблиц объектов программы  /////////////////

//=====================================================================

template <class Object>          ObjectTable<Object>::ObjectTable<Object> (int max_size)
                                                                   { p = new Object * [(size = max_size) + 1]; free = 0; }
template <class Object> Object * ObjectTable<Object>::put_obj (Object * t, int i){  p [i]       = t; return t; }
template <class Object> Object * ObjectTable<Object>::put_obj (Object * t)        {  p [free ++] = t; return t; }
template <class Object> int      ObjectTable<Object>::get_place               () const { return free; }
template <class Object> Object * ObjectTable<Object>::operator []        (int k)       { return p[k]; }
template <class Object>          ObjectTable<Object>:: ~ ObjectTable<Object>   ()       {  delete[] p; }

template <class Object> int      ObjectTable<Object>::get_index (Token * l) const
{ Object ** q = p;
  for (int i = 0; i < free; i ++) { if (* q ++ == l -> get_value ())    return i; } return - 1;
};
template <class Object> int ObjectTable<Object>::get_index (int v) const
{ Object ** q = p;
  for (int i = 0; i < free; i ++) { if ((* q ++) -> get_value () == v)  return i; } return - 1;
};
template <class Object> int ObjectTable<Object>::get_index (type_of_lex t) const
{ Object ** q = p;
  for (int i = 0; i < free; i ++) { if ((*(* q ++)).get_type () == t)   return i; } return - 1;
};

template <class Object> Object * ObjectTable<Object>::get_object (const buf & b) const
{ Object ** q = p; Object * t;
  for (int i = 0; i < free; i ++) { t = * q ++; if (t -> is_object (b)) return t; } return   0;
}
```

```
//============================================================

///////////////  Базовый класс объектов программы  ///////////////

//============================================================

type_of_lex ProgramObject::get_type             () const { return  type;      }
       void ProgramObject::set_type (type_of_lex t)       { type  =    t;      }
        int ProgramObject::get_value            () const { return value;      }
       void ProgramObject::set_value        (int v)       { value =    v;      }

//============================================================

/////// Конструкторы производных классов объектов программы  ////////

//============================================================

        Ident::   Ident   (const buf & b) { char * str = b.get_string (); name = new char [strlen (str) + 1];
                                                                   declare = assign = false; strcpy (name, str); }
       Number::Number     (const buf & b)           { value = b.int_buf ();  }
        Label::Label                        (int n) { value = n;             }
      Address::Address                      (int n) { value = n;             }
    Operation::Operation (char * str, type_of_lex t) { type  = t; sign = new char [strlen (str)+1]; strcpy (sign, str); }
   TrueObject::   TrueObject (char * str, type_of_lex t): Operation (str, t) {}
  FalseObject::  FalseObject (char * str, type_of_lex t): Operation (str, t) {}
    NotObject::    NotObject (char * str, type_of_lex t): Operation (str, t) {}
     OrObject::     OrObject (char * str, type_of_lex t): Operation (str, t) {}
    AndObject::    AndObject (char * str, type_of_lex t): Operation (str, t) {}
     EqObject::     EqObject (char * str, type_of_lex t): Operation (str, t) {}
     LtObject::     LtObject (char * str, type_of_lex t): Operation (str, t) {}
     GtObject::     GtObject (char * str, type_of_lex t): Operation (str, t) {}
     LeObject::     LeObject (char * str, type_of_lex t): Operation (str, t) {}
     GeObject::     GeObject (char * str, type_of_lex t): Operation (str, t) {}
     NeObject::     NeObject (char * str, type_of_lex t): Operation (str, t) {}
   PlusObject::   PlusObject (char * str, type_of_lex t): Operation (str, t) {}
  MinusObject::  MinusObject (char * str, type_of_lex t): Operation (str, t) {}
   MultObject::   MultObject (char * str, type_of_lex t): Operation (str, t) {}
    DivObject::    DivObject (char * str, type_of_lex t): Operation (str, t) {}
 AssignObject:: AssignObject (char * str, type_of_lex t): Operation (str, t) {}
   GoToObject::   GoToObject (char * str, type_of_lex t): Operation (str, t) {}
GoIfNotObject::GoIfNotObject (char * str, type_of_lex t): Operation (str, t) {}
  WriteObject::  WriteObject (char * str, type_of_lex t): Operation (str, t) {}
   ReadObject::   ReadObject (char * str, type_of_lex t): Operation (str, t) {}
```

```
//====================================================================

//////// Реализация производных классов объектов программы  /////////

//====================================================================

   bool       Ident::get_assign          () const { return assign;           }
   void       Ident::set_assign          ()       { assign  = true;          }
   bool       Ident::get_declare         () const { return declare;          }
   void       Ident::set_declare         ()       { declare = true;          }
   char *     Ident::get_name            () const { return name;             }
   bool       Ident::is_object (const buf & b) const
                   { return ! strcmp (b.get_string (), name);                }
ostream &     Ident::print     (ostream & s) const
                   { s << "Имя      "<< name << endl; return s;              }

//====================================================================

   bool      Number::is_object (const buf & b) const
                   { return b.int_buf () == value;                           }
ostream &    Number::print     (ostream & s) const
                   { s << "Число  = "; s.width (9); s << value << endl; return s;}

//====================================================================

   bool       Label::is_object (const buf & b) const { return true;          }
ostream &     Label::print     (ostream & s) const
                   { s << "Метка  = "; s.width (4); s << value << endl; return s;}

//====================================================================

   bool     Address::is_object (const buf & b) const { return true;          }
ostream &   Address::print     (ostream & s) const
                   { s << "Адрес  = ";
                     s.width (4); s << value << "  " << TI [value]; return s; }

//====================================================================

   bool   Operation::is_object (const buf & b) const
                   { return ! strcmp (b.get_string (), sign);                }
ostream & Operation::print     (ostream & s) const
                   { s << "Объект   ";
                     s /*.width (4); s << type << " " */ << sign << endl; return s; }
```

```
//=======================================/////// Класс Scanner ///////===============================================
        Scanner:: Scanner (char * program) { fp  = fopen  (program, "r");          GetS (); FA_State = H; }
        Scanner::~Scanner ()                {      fclose (fp); }   void Scanner::GetS () { c = fgetc (fp); }

Ident * Scanner::CreateIdentObject  (const buf & b)
    { Ident  * I = TI.get_object(b); return I == 0 ? TI.put_obj (new Ident  (b)) : I; }
Number* Scanner::CreateNumberObject (const buf & b)
    { Number * N = TC.get_object(b); return N == 0 ? TC.put_obj (new Number (b)) : N; }

Token * Scanner::get_lex ()                    { Token * res;
  for (;;) { switch (FA_State) {
         case H:          if (isspace (c))                                       GetS ();
                     else if (isalpha (c))  {                b.clear (); b.add (c); GetS (); FA_State = Identifier;        }
                     else if (isdigit (c))  {                b.clear (); b.add (c); GetS (); FA_State = Literal;           }
                     else if (c =='@')      {                b.clear (); b.add (c);          FA_State = Error;
                     else if (c =='{')      {                                        GetS (); FA_State = Comment;           }
                                                                                              return TD.get_token (b);      }
                     else if (c ==':'||c =='<'||c =='>') { b.clear (); b.add (c); GetS (); FA_State = LOperator;         }
                     else if (c =='!')      {                b.clear (); b.add (c); GetS (); FA_State = NOperator;         }
                     else                   {                b.clear (); b.add (c);          FA_State =  Operator;         }
                     break;
         case Identifier: if (isalnum (c) ) {                            b.add (c); GetS ();                                }
                     else { FA_State = H; if ((res = TW.get_token (b)) != 0)                 return res;
                                          else                    return new Token (LEX_ID,  CreateIdentObject  (b)); }
                     break;
        case Literal:     if (isdigit (c))  {                            b.add (c); GetS ();                                }
                     else { FA_State = H;                         return new Token (LEX_NUM, CreateNumberObject (b)); }
                     break;
        case Comment:                      if (c == '}') {                          GetS (); FA_State = H;                 }
                                      else if (c == '@') {                                   FA_State = Error;    throw c; }
                                      else if (c == '{')                                                          throw c;
                                      else                                          GetS ();
                     break;
        case LOperator:   FA_State = H; if (c == '=') {                 b.add (c); GetS ();                                }
                                                                                     return TD.get_token (b);
        case NOperator:   FA_State = H; if (c == '=') {                 b.add (c); GetS (); return TD.get_token (b);       }
                     else                                                                                          throw '!';
        case Operator:    FA_State = H; if ((res = TD.get_token (b)) != 0) {        GetS (); return res;                    }
                     else                                                                                          throw c;
        case Error:                                                                                                throw c;
       } //end switch
     } // end for
}
```

```
//===================================================================
/////////////////////////  Класс Parser  ///////////////////////////
//===================================================================
/////////////////// Вспомогательные процедуры //////////////////////

Address * Parser::CreateAddressObject (Token * l)
{ int    Ident_index = TI.get_index (l);
  int      Adr_index = TA.get_index (Ident_index);
  return   Adr_index >= 0 ? TA [  Adr_index] : TA.put_obj (new Address (Ident_index));
}
Label   * Parser::CreateLabelObject   (int label)
{ int    Label_index = TL.get_index (label);
  return Label_index >= 0 ? TL [Label_index] : TL.put_obj (new   Label (label));
}

//////////// Основные процедуры синтаксического анализа ////////////

     Parser:: Parser (char * program): scan (program) {}
     Parser::~Parser               ()                  {}
void Parser::GetL ()
{ curr_lex = scan.get_lex ();        if (Dump & DUMP_LEX) cout << curr_lex;
  c_type = curr_lex -> get_type ();
}
void  Parser::Analyze ()
{                                                                                              GetL (); P  ();
}
void Parser::P ()
{ if (Dump & DUMP_REC) cout << "P  (): entry\n";
  if (c_type != LEX_PROGRAM)     throw curr_lex;                                               GetL (); D1 ();
  if (c_type != LEX_SEMICOLON)   throw curr_lex;                                               GetL (); B  ();
  if (c_type != LEX_FIN)         throw curr_lex;
  if (Dump & DUMP_REC) cout << "P  (): exit\n";
}
void Parser::D1()
{ if (Dump & DUMP_REC) cout << "D1 (): entry\n";
  if (c_type != LEX_VAR)         throw curr_lex;
  do {                                                                                         GetL (); D  ();
     } while (c_type == LEX_COMMA);
  if (Dump & DUMP_REC) cout << "D1 (): exit\n";
}
```

```
void Parser::D () {
  if (Dump & DUMP_REC) cout << "D  (): entry\n";
  Names.reset ();
  if (c_type != LEX_ID)          throw curr_lex;
                                              Names.push (curr_lex);                      GetL ();
  while (c_type == LEX_COMMA)  {                                                          GetL ();
                                              if (c_type != LEX_ID)     throw curr_lex;
                                              Names.push (curr_lex);                      GetL ();
        }
  if (c_type != LEX_COLON)       throw curr_lex;
                                                                                          GetL ();
  if (c_type == LEX_INT || c_type == LEX_BOOL) {       decl (c_type);                     GetL (); }
  else                           throw curr_lex;
  if (Dump & DUMP_REC) cout << "D  (): exit\n";
}

void Parser::B () {
  if (Dump & DUMP_REC) cout << "B  (): entry\n";
  if (c_type != LEX_BEGIN)       throw curr_lex;
  do {                                                                                    GetL (); S  ();
     } while (c_type == LEX_SEMICOLON);
  if (c_type != LEX_END)         throw curr_lex;                                          GetL ();
  if (Dump & DUMP_REC) cout << "B  (): exit\n";
}

void Parser::E () {
  if (Dump & DUMP_REC) cout << "E  (): entry\n";
                                                                                                   E1 ();
  if (c_type == LEX_EQ || c_type == LEX_LT || c_type == LEX_GT ||
      c_type == LEX_LE || c_type == LEX_NE || c_type == LEX_GE)
       {                                      Types.push (c_type);                        GetL (); E1 (); check_op ();}
  if (Dump & DUMP_REC) cout << "E  (): exit\n";
}

void Parser::E1 () {
  if (Dump & DUMP_REC) cout << "E1 (): entry\n";
                                                                                                   T  ();
  while (c_type == LEX_OR || c_type == LEX_MINUS || c_type == LEX_PLUS)
       {                                      Types.push (c_type);                        GetL (); T  (); check_op ();}
  if (Dump & DUMP_REC) cout << "E1 (): exit\n";
}
```

```
void Parser::T () {
  if (Dump & DUMP_REC) cout << "T  (): entry\n";
                                                                                                            F  ();
  while (c_type == LEX_MULT || c_type == LEX_DIV || c_type == LEX_AND)
       {                                       Types.push (c_type);                         GetL (); F  (); check_op ();}
  if (Dump & DUMP_REC) cout << "T  (): exit\n";
}
void Parser::F () { ProgramObject * Oper = curr_lex -> get_value ();
  if (Dump & DUMP_REC) cout << "F  (): entry\n";
       if (c_type == LEX_ID)    {              check_id  ();   PLZ.put_obj (Oper);
                                                               delete curr_lex;             GetL (); }
  else if (c_type == LEX_NUM)   { Types.push (LEX_INT);        PLZ.put_obj (Oper);
                                                               delete curr_lex;             GetL (); }
  else if (c_type == LEX_TRUE)  { Types.push (LEX_BOOL);       PLZ.put_obj (Oper);          GetL (); }
  else if (c_type == LEX_FALSE) { Types.push (LEX_BOOL);       PLZ.put_obj (Oper);          GetL (); }
  else if (c_type == LEX_NOT)   { GetL (); F (); check_not (); PLZ.put_obj (Oper);                   }
  else if (c_type == LEX_LPAREN){ GetL (); E (); if (c_type == LEX_RPAREN)                  GetL ();
                                                 else                              throw curr_lex; }
  else throw curr_lex;
  if (Dump & DUMP_REC) cout << "F  (): exit\n";
}
void Parser::S () { int lab1, lab2; ProgramObject * Oper;
  if (Dump & DUMP_REC) cout << "S  (): entry\n";
       if (c_type == LEX_ID)
       { check_id ();                          PLZ.put_obj (CreateAddressObject (curr_lex));
         delete curr_lex;                      GetL ();     Oper = curr_lex -> get_value();
         if (c_type == LEX_ASSIGN)
             { GetL (); E (); eq_type ();          PLZ.put_obj (Oper);
             }
         else throw curr_lex;
       } // assign-end
  else if (c_type == LEX_WHILE)
       { GetL (); lab1 = PLZ.get_place ();
                  E (); eq_type (LEX_BOOL); lab2 = PLZ.get_place (); PLZ.put_obj ();
         PLZ.put_obj (TO [ind_FGO]);
         if (c_type == LEX_DO)
             { GetL (); S ();  PLZ.put_obj (CreateLabelObject (lab1));
               PLZ.put_obj (TO [ind_GO]);
               PLZ.put_obj (CreateLabelObject (PLZ.get_place ()), lab2);
             }
         else throw curr_lex;
       } // end while
```

```
  else if (c_type == LEX_IF)
       { GetL (); E (); eq_type (LEX_BOOL); lab1 = PLZ.get_place (); PLZ.put_obj ();
         PLZ.put_obj (TO [ind_FGO]);
         if (c_type == LEX_THEN)
             { GetL (); S ();                lab2 = PLZ.get_place (); PLZ.put_obj ();
               PLZ.put_obj (TO [ind_GO]);
               PLZ.put_obj (CreateLabelObject (PLZ.get_place ()), lab1);
               if (c_type == LEX_ELSE)
                   { GetL (); S (); PLZ.put_obj (CreateLabelObject (PLZ.get_place ()), lab2);
                   }
               else throw curr_lex;
             }
         else throw curr_lex;
       } // end if
  else if (c_type == LEX_READ)
       { Oper = curr_lex -> get_value ();
         GetL (); if (c_type == LEX_LPAREN)
             { GetL ();
               if (c_type == LEX_ID)
                  { check_id_in_read ();          PLZ.put_obj (CreateAddressObject (curr_lex));
                    delete curr_lex;              GetL ();                              }
               else throw curr_lex;
               if (c_type == LEX_RPAREN)         { GetL(); PLZ.put_obj (Oper);  }
               else throw curr_lex;
             }
         else throw curr_lex;
       } // end read
  else if (c_type == LEX_WRITE)
       { Oper = curr_lex -> get_value ();
         GetL (); if (c_type == LEX_LPAREN)
             { GetL (); E ();                     Types.pop ();
               if (c_type == LEX_RPAREN)         { GetL(); PLZ.put_obj (Oper); }
               else throw curr_lex;
             }
         else throw curr_lex;
       } // end write
  else B ();
  if (Dump & DUMP_REC) cout << "S  (): exit\n";
}

void Parser::eq_type                  () const
{ if (Types.pop () != Types.pop ())            throw "Несоответствие типов в присваивании :=";
}
```

```
void Parser::eq_type (type_of_lex token) const
{ if (Types.pop () != token)                    throw "Требуется логическое выражение";
}
void Parser::decl (type_of_lex type) const
{ while (! Names.is_empty ())
       { Token * Ident_lex = Names.pop ();
         Ident * t = dynamic_cast<Ident*> (Ident_lex -> get_value ());  delete Ident_lex;
         if (t -> get_declare ())               throw "Повторное описание";
         else { t -> set_declare ();  t -> set_type (type); }
       }
}

void Parser::check_op () const
//  t       - тип операнда, требуемый для данной операции
//  r       - тип формируемого операцией результата
// (начальные значения t и r соответствуют операциям отношения)
//  t1 и t2 - реальные типы операндов
{ type_of_lex t = LEX_INT, r = LEX_BOOL;
  type_of_lex t2 = Types.pop ();
  type_of_lex op = Types.pop ();
  type_of_lex t1 = Types.pop ();
  if (op == LEX_PLUS || op == LEX_MINUS || op == LEX_MULT || op == LEX_DIV) r  = LEX_INT;
  if (op == LEX_OR   || op == LEX_AND)                                     t  = LEX_BOOL;
  if (t1 == t2  &&  t1 == t)                                               Types.push (r);
  else throw "Неверные типы в двуместной операции";
  PLZ.put_obj (TO [TO.get_index (op)]);
}

void Parser::check_not        () const
{ if (Types.pop () != LEX_BOOL)                                 throw "Неверный тип в операции отрицания";
  else Types.push (LEX_BOOL);
}

void Parser::check_id         () const
{ Ident * t = dynamic_cast<Ident*> (curr_lex -> get_value ());
  if (t -> get_declare ())     Types.push (t -> get_type  ());  else throw "Отсутствует описание";
}

void Parser::check_id_in_read () const
{ Ident * t = dynamic_cast<Ident*> (curr_lex -> get_value ());
  if (! t -> get_declare ())                                    throw "Не описан идентификатор";
}
```

```
//=====================================================================

///////////// Операционные функции программных объектов /////////////

//=====================================================================


void         Ident::exec (int &) const
                  { if (get_assign ()) Values.push (get_value ());
                    else throw "PLZ: неопределенное значение у имени";    }
void        Number::exec (int &) const { Values.push (get_value ());     }
void       Address::exec (int &) const { Values.push (get_value ());     }
void         Label::exec (int &) const { Values.push (get_value ());     }
void    TrueObject::exec (int &) const { Values.push (1);                }
void   FalseObject::exec (int &) const { Values.push (0);                }
void     NotObject::exec (int &) const { Values.push (1 - Values.pop ());}
void      OrObject::exec (int &) const { Values.push (Values.pop () || Values.pop ());           }
void     AndObject::exec (int &) const { Values.push (Values.pop () && Values.pop ());           }
void      EqObject::exec (int &) const { Values.push (Values.pop () == Values.pop ());           }
void      LtObject::exec (int &) const { Values.push (Values.pop () >= Values.pop ());           }
void      GtObject::exec (int &) const { Values.push (Values.pop () <= Values.pop ());           }
void      LeObject::exec (int &) const { Values.push (Values.pop () >  Values.pop ());           }
void      GeObject::exec (int &) const { Values.push (Values.pop () <  Values.pop ());           }
void      NeObject::exec (int &) const { Values.push (Values.pop () != Values.pop ());           }
void    PlusObject::exec (int &) const { Values.push (Values.pop () +  Values.pop ());           }
void   MinusObject::exec (int &) const { int k = Values.pop (); Values.push (Values.pop () - k); }
void    MultObject::exec (int &) const { Values.push (Values.pop () *  Values.pop ());           }
void     DivObject::exec (int &) const
                  { int k = Values.pop ();
                    if (k)  Values.push (Values.pop () / k);
                    else throw "PLZ: деление на нуль";                    }
void  AssignObject::exec (int &) const
                  { int k = Values.pop (); Ident * t = TI [Values.pop ()];
                    t -> set_value (k);   t -> set_assign ();            }
void    GoToObject::exec (int & i) const { i = Values.pop () - 1;        }
void GoIfNotObject::exec (int & i) const
                  { int k = Values.pop (); if(!Values.pop ()) i = k - 1; }
void   WriteObject::exec (int &) const
                  { cout << "Writing=>" << Values.pop () << endl;        }
```

```
void    ReadObject::exec (int &) const
                  { int k = 0; Ident * t = TI [Values.pop ()];
                    if (t -> get_type () == LEX_INT)
                          { cout << "Введите значение для "
                                 << t -> get_name () << " =>" << flush; scanf ("%d", & k);
                          }
                    else  { char m [80];
                            for (int l = 0; l < 3; l ++)
                              { cout << "Введите логическое значение как true или false для "
                                     << t -> get_name () << " =>" << flush; scanf ("%s", m);
                                     if (! strcmp (m,  "true")) { k = 1; break; }
                                else if (! strcmp (m, "false")) { k = 0; break; }
                                cout << "Ошибка при вводе: true/false" << endl;
                              }
                          }
                     t -> set_value (k); t -> set_assign ();
                  }

//====================================================================

///////////////////////  Класс Simulator  ////////////////////////////

//====================================================================

void Simulator::Simulate ()
{ int size;
  Values.reset ();
  size = PLZ.get_place ();
  for (int index = 0; index < size; index ++)
     { curr_obj = PLZ [index];                if (Dump & DUMP_SIM) { cout.width (4); cout << index << ": " << curr_obj; }
       if (curr_obj) curr_obj -> exec (index);
     }
}

Simulator::~ Simulator () {}
```

```
//====================================================================

////////////////////////  Функция main  ////////////////////////////

//====================================================================


int main (int argc, char ** argv)
{ bool res = false;

//           Установка отладочных режимов работы интерпретатора

  Dump = /* * /+ DUMP_BUF /* */
         /* * /+ DUMP_REC /* */
         /* */ + DUMP_LEX /* */
         /* */ + DUMP_PLZ /* */
         /* */ + DUMP_PTA /* */
         /* */ + DUMP_PTC /* */
         /* */ + DUMP_PTI /* */
         /* */ + DUMP_PTL /* */
         /* */ + DUMP_SIM /* */
               + 0;

//       Создание операционных объектов и построение индексов для объектов, обрабатываемых индивидуально

  Token * T; char * ExtRep; type_of_lex type; int size = TW.get_size () - 1;
  for (int i = 0; i < size; i ++)
    { T = TW (i);  ExtRep = TW [i]; type = T -> get_type ();
      switch (type)
       { case LEX_TRUE:  TW.put_obj (TO.put_obj (new    TrueObject (ExtRep,type)), i); break;
         case LEX_FALSE: TW.put_obj (TO.put_obj (new   FalseObject (ExtRep,type)), i); break;
         case LEX_NOT:   TW.put_obj (TO.put_obj (new     NotObject (ExtRep,type)), i); break;
         case LEX_OR:    TW.put_obj (TO.put_obj (new      OrObject (ExtRep,type)), i); break;
         case LEX_AND:   TW.put_obj (TO.put_obj (new     AndObject (ExtRep,type)), i); break;
         case PLZ_GO:    TW.put_obj (TO.put_obj (new    GoToObject (ExtRep,type)), i); break;
         case PLZ_FGO:   TW.put_obj (TO.put_obj (new GoIfNotObject (ExtRep,type)), i); break;
         case LEX_WRITE: TW.put_obj (TO.put_obj (new   WriteObject (ExtRep,type)), i); break;
         case LEX_READ:  TW.put_obj (TO.put_obj (new    ReadObject (ExtRep,type)), i); break;
       }
    }
  ind_GO  = TO.get_index (PLZ_GO);
  ind_FGO = TO.get_index (PLZ_FGO);
```

```
  size = TD.get_size () - 1;

  for (i = 0; i < size; i ++)
    { T = TD (i);  ExtRep = TD [i]; type = T -> get_type ();
      switch (type)
       { case LEX_EQ:    TD.put_obj (TO.put_obj (new      EqObject (ExtRep,type)), i); break;
         case LEX_LT:    TD.put_obj (TO.put_obj (new      LtObject (ExtRep,type)), i); break;
         case LEX_GT:    TD.put_obj (TO.put_obj (new      GtObject (ExtRep,type)), i); break;
         case LEX_LE:    TD.put_obj (TO.put_obj (new      LeObject (ExtRep,type)), i); break;
         case LEX_GE:    TD.put_obj (TO.put_obj (new      GeObject (ExtRep,type)), i); break;
         case LEX_NE:    TD.put_obj (TO.put_obj (new      NeObject (ExtRep,type)), i); break;
         case LEX_PLUS:  TD.put_obj (TO.put_obj (new    PlusObject (ExtRep,type)), i); break;
         case LEX_MINUS: TD.put_obj (TO.put_obj (new   MinusObject (ExtRep,type)), i); break;
         case LEX_MULT:  TD.put_obj (TO.put_obj (new    MultObject (ExtRep,type)), i); break;
         case LEX_DIV:   TD.put_obj (TO.put_obj (new     DivObject (ExtRep,type)), i); break;
         case LEX_ASSIGN:TD.put_obj (TO.put_obj (new  AssignObject (ExtRep,type)), i); break;
       }
    }

//            Проведение синтаксического анализа указанной программы

  if (Dump & DUMP_LEX)       {cout << "Программа в виде последовательности лексем:::"         << endl << endl;          }

  try { Parser * M = new Parser ("program.txt");                M -> Analyze ();     delete M;          res     = true;}

  catch (char c)             {cout << "Неверный символ при лексическом анализе: "               << c << endl; return 1;}
  catch (ProgramObject * l)  {cout << "Неверный программный объект при синтаксическом анализе: " << l << endl; return 1;}
  catch (Token * t)          {cout << "Неверная лексема при синтаксическом анализе: "           << t << endl; return 1;}
  catch (const char * source){cout << source                                                      << endl; return 1;}
  catch (...)                {cout << "Непонятная ситуация при анализе входного текста."          << endl; return 1;}

  if (Dump & DUMP_LEX)       {cout << "res = " << (res ? "true " : "false")                       << endl;          }

//            Отладочная выдача созданных при синтаксическом анализе таблиц

  if (Dump & DUMP_PTI)       {cout << "Таблица имен:::::::::::::::::::::::::::::::::"                << endl     << TI;}
  if (Dump & DUMP_PTA)       {cout << "Таблица адресов::::::::::::::::::::::::::::::"                << endl     << TA;}
  if (Dump & DUMP_PTC)       {cout << "Таблица констант:::::::::::::::::::::::::::::"                << endl     << TC;}
  if (Dump & DUMP_PTL)       {cout << "Таблица меток::::::::::::::::::::::::::::::::"                << endl     << TL;}
  if (Dump & DUMP_PLZ)       {cout << "Программа в инверсной польской записи:::::::"                << endl     <<PLZ;}
```

```
// Создание объекта для проведения интерпретации и интерпретация внутреннего представления

  if (Dump & DUMP_SIM)       {cout << "Начало работы:::::::::::::::::::::::::::::::::"                 << endl;          }
  try { Simulator * S = new Simulator ();
        S -> Simulate ();

//                Уничтожение ненужных объектов, собранных в таблицы

        size = TA.get_place (); for (i = 0; i < size; i ++) delete TA [i];
        size = TC.get_place (); for (i = 0; i < size; i ++) delete TC [i];
        size = TI.get_place (); for (i = 0; i < size; i ++) delete TI [i];
        size = TL.get_place (); for (i = 0; i < size; i ++) delete TL [i];
        size = TO.get_place (); for (i = 0; i < size; i ++) delete TO [i];
        delete S;
      }
  catch (const char * source){cout << source                                                    << endl; return 1;}
  catch (...)                {cout << "Непонятная ситуация в интерпретаторе."                  << endl; return 1;}

  if (Dump & DUMP_ANY)       {cout << "Конец работы:::::::::::::::::::::::::::::::::::"                 << endl;          }
  return 0;
}
```