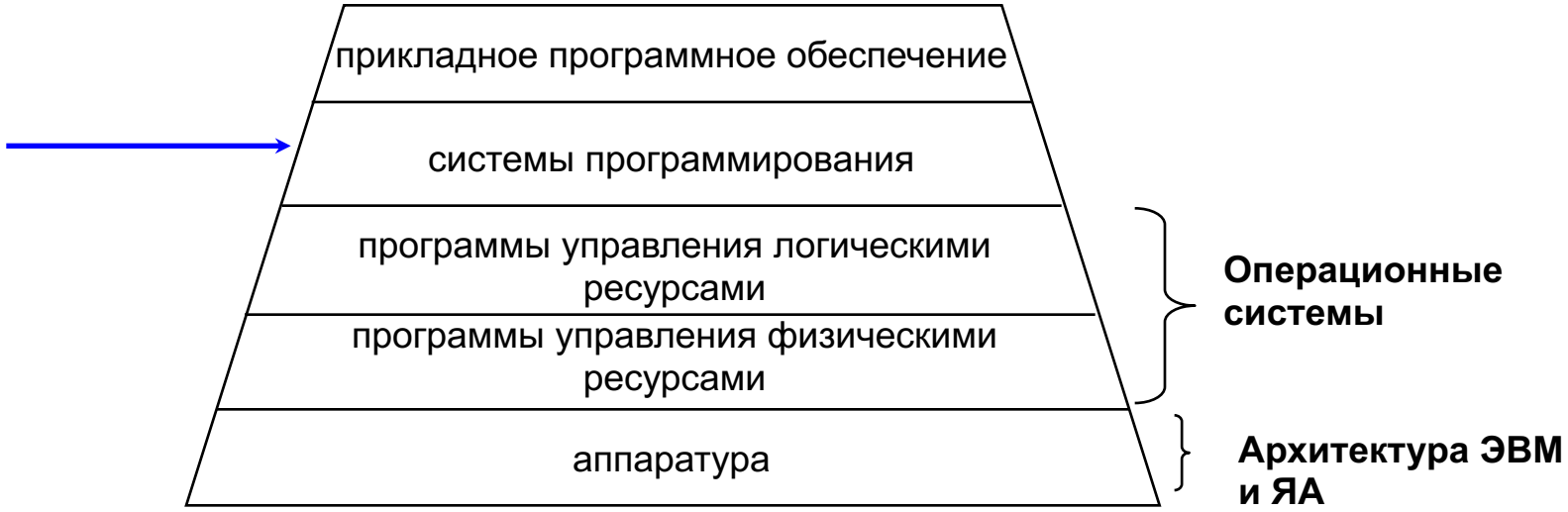
С И С Т Е М Ы П Р О Г Р А М М И Р О В А Н И Я



Иерархия вычислительной системы

С И С Т Е М Ы П Р О Г Р А М М И Р О В А Н И Я

Развитие СП:

- программирование в машинных кодах
- автокоды, языки ассемблера
- трансляторы с языков высокого уровня
- визуальные средства автоматизации и проектирования

Определение: *системой программирования называется комплекс программных средств (инструментов, библиотек) , предназначенных для поддержки разработки программного продукта на протяжении всего жизненного цикла этого продукта*

С И С Т Е М Ы П Р О Г Р А М М И Р О В А Н И Я

- Основные понятия, назначение, структура и функционирование СП
- Принципы ООП на примере языка C++ и СП, поддерживающие ООП
- Элементы теории трансляции

Коллоквиум по курсу – Письменная работа (по окончании курса)

Баллы учитываются на экзамене

Экзамен в июне (письменный) задачи по всем темам курса

Список основной литературы

1. И. А. Волкова, А. В. Иванов, Л. Е. Карпов. Основы объектно-ориентированного программирования. Язык программирования C++. Учебное пособие для студентов 2 курса. – М.: Издательский отдел факультета ВМК МГУ, 2011.

2. И. А. Волкова, А. А. Вылиток, Т. В. Руденко. Формальные грамматики и языки. Элементы теории трансляции (3-е издание). – М.: Изд-во МГУ, 2009

3. И. А. Волкова, И. Г. Головин, Л. Е. Карпов. Системы программирования (Учебное пособие). – М.: Издательский отдел факультета ВМиК МГУ, 2009.

4. И. А. Волкова, А. А. Вылиток, Л. Е. Карпов. Сборник задач и упражнений по языку C++ (Учебное пособие для студентов 2 курса). – М.: Издательский отдел факультета ВМК МГУ, 2013.

Список дополнительной литературы

1. А. Ахо, Р. Сети, Дж. Ульман. Компиляторы. — М.: Изд. дом «Вильямс», 2001. (Шифр в библиотеке МГУ: 5ВГ66 А-955)
2. А. В. Гордеев, А. Ю. Молчанов. Системное программное обеспечение. — СПб.: Питер, 2001
3. Г. Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ (zip), 2-е издание. — М. СПб.: «Издательство Бином» — «Невский диалект»,1998.
4. А. Элиенс. Принципы объектно-ориентированной разработки программ, 2-е издание. — М.: Издательский дом «Вильямс», 2002.
5. Дж. Макгрегор, Д. Сайкс. Тестирование объектно-ориентированного программного обеспечения. Практическое пособие. — М.: «DiaSoft», 2002.
6. Б. Страуструп. Язык программирования C++. Специальное издание. — М.: Издательство «БИНОМ», 2001.
7. Б. Страуструп. Программирование: принципы и практика использования C++.: Пер. с англ. – М. ООО «И.Д.Вильямс», 2011. – 1248 с.
8. Г. Шилдт. Самоучитель C++. 3-е изд. — СПб: БХВ-Петербург, 2002.

Интернет-ресурсы

Материалы по курсу можно найти на сайте:

http://cmcmsu.info/2course/

4-й семестр: Системы программирования

Язык C++

C++ позволяет справиться с возрастающей сложностью программ (в отличие от C).

Автор – Бьёрн Страуструп.

Стандарты (комитета по стандартизации ANSI) – 1998, 2011, 2014, 2017 … .

C++:

- лучше C,
- поддерживает абстракции данных,
- поддерживает объектно-ориентированное программирование (ООП).

Парадигмы программирования

Все программы состоят из кода и данных и каким-либо образом концептуально организованы вокруг своего кода и\или данных.

Основные парадигмы (технологии) программирования определяют способ построения программ:

- **процедурно-ориентированная** (программа – это ряд последовательно выполняемых операций, причём код воздействует на данные, например в программах на С),
- **объектно-ориентированная** (программа состоит из объектов – программных сущностей, объединяющих в себе код и данные, взаимодействующих друг с другом через определенные интерфейсы, при этом доступ к коду и данным объекта осуществляется только через сам объект, т.е. данные определяют выполняемый код),
- функциональная (программа – набор математических функций, а ее выполнение – вычисление некоторой главной функции) ,
- логическая (программа – набор логических высказываний, а ее выполнение – доказательство или опровержение заданного высказывания) .

Объектная парадигма

Объект = состояние + поведение

Поведение = посылка сообщений себе и другим объектам.

Для каждого вида сообщения существуют «обработчики», которые могут модифицировать состояние объекта и посылать сообщения другим объектам.

Пример

запись X= X + Y

трактуется в объектной парадигме так:

объекту X посылается сообщение

«складывайся с объектом Y и измени свое состояние на новое – результат сложения»

Объекты с одинаковым поведением и возможными состояниями объединяются в *классы*

Классы образуют *иерархии*

Иерархия – это способ борьбы со сложностью систем

Иерархия «часть/целое»

В ООП это называется структурой объектов

Иерархия «общее/частное»

- Транспортное средство
  - грузовое ТС
    - самосвал
    - бензовоз
  - пассажирское ТС
    - автобус
    - легковой автомобиль

В ООП это называется структурой классов

Объектно-ориентированное программирование

это метод программирования, основанный на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определенного класса, а классы являются членами определенной иерархии наследования

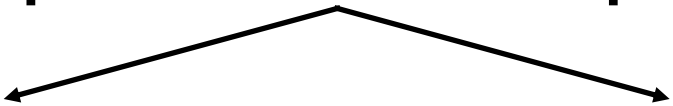
Постулаты ООП

Абстракция - центральное понятие ООП

Абстракция позволяет программисту справиться со сложностями решаемых им задач.

Мощный способ создания абстракций –

иерархическая классификация



типовая (**структура классов**)

структурная (**структура объектов**)

Основные механизмы (постулаты) ООП:

-**инкапсуляция**,

-**наследование**,

-**полиморфизм**.

ИНКАПСУЛЯЦИЯ

Инкапсуляция - механизм,

- связывающий вместе код и данные, которыми он манипулирует;

- защищающий их от произвольного доступа со стороны другого кода, внешнего по отношению к рассматриваемому.

Доступ к коду и данным жестко контролируется интерфейсом.

Основой инкапсуляции является **класс**.

Класс - это механизм (пользовательский тип данных) для создания объектов.

Объект класса - переменная типа класс или экземпляр класса.

Любой объект характеризуется **состоянием** (значениями полей данных) и **поведением** (операциями над объектами, задаваемыми определенными в классе функциями, которые называют **методами** класса).

НАСЛЕДОВАНИЕ

Наследование - механизм, с помощью которого один объект (**производного класса**) приобретает свойства другого объекта (**родительского, базового класса**).

Наследование позволяет объекту производного класса наследовать от своего родителя общие атрибуты, а для себя определять только те характеристики, которые делают его уникальным внутри класса.

Производный класс конкретизирует, в общем случае **расширяет** базовый класс.

Наследование поддерживает концепцию иерархической классификации.

Новый класс не обязательно описывать, начиная с нуля, что существенно упрощает работу программиста.

ПОЛИМОРФИЗМ

Полиморфизм - механизм, позволяющий использовать один и тот же интерфейс для общего класса действий.

В общем случае концепция полиморфизма выражается с помощью фразы "один интерфейс - много методов".

Выбор конкретного действия (метода) применительно к конкретной ситуации возлагается на компилятор. Программисту же достаточно запомнить и применять один интерфейс, вместо нескольких, что также упрощает его работу.

Различаются следующие виды полиморфизма:

-**статический** (на этапе компиляции, с помощью перегрузки функций),

-**динамический** (во время выполнения программы, реализуется с помощью виртуальных функций) и

-**параметрический** (на этапе компиляции, с использованием механизма шаблонов).

Декомпозиция задачи

При программировании в объектно-ориентированном стиле на первое место выходит **проектирование** решения задачи, т.е определение того, какие классы и объекты будут использоваться в программе, каковы их свойства и способы взаимодействия.

Как правило, при этом необходимо произвести декомпозицию задачи.

Декомпозиция – научный метод, использующий структуру задачи и позволяющий разбить решение одной большой задачи на решения серии меньших задач, возможно взаимосвязанных, но более простых.

Процедурно- и объектно-ориентированная декомпозиция задачи

Процедурно-ориентированная декомпозиция — вычленение из алгоритма решения задачи модулей, выполняющих некоторое самостоятельное действие (оформление некоторых действий в виде отдельных функций и процедур).

Объектно-ориентированная декомпозиция – выделение элементов, принадлежащих различным абстракциям проблемной области (вычленение объектов проблемной области и определение их свойств).

Примеры: автобаза, студент, ВУЗ, стек.

Принципы объектно-ориентированной декомпозиции задачи

1. Выделяемые элементы не следует делать слишком мелкими — это усложнит процедуры их координации и взаимодействия.

2. Удобно при выделении элементов представлять их в виде черного ящика, внутренне устройство которого неизвестно, но определены выполняемые им действия и важные для внешнего использования «входы» и «выходы» (набор функций для получения/выдачи информации или изменения состояния элемента , т. н. ***интерфейс*** выделенного элемента).

3. Компоненты, в рамках одного выделенного элемента должны быть концептуально взаимосвязаны.

4. Для удобства и простоты использования выделенных элементов их интерфейс следует стремиться минимизировать.

Синтаксис класса

class имя_класса {
[private:]
закрытые члены класса (функции, типы и поля-данные)
public:
открытые члены класса (функции, типы и поля-данные)
protected:
защищенные члены класса
} список_объектов;

Описание объектов – экземпляров класса:
имя_класса список объектов;
// служ. слово class не требуется

Классы C++ отличаются от структур C++ **только** правилами определения **по умолчанию**
-**прав доступа** к первой области доступа членов класса и
-**вида наследования**:
для **структур** – **public**,
для **классов** – **private**.

Члены класса

- Члены-данные;
- Члены-функции (методы);
- Члены-типы – вложенные пользовательские типы,

Правила доступа к членам класса и поиска их имен единообразны для всех членов класса и не зависят от их вида.

Ex.:
```
class X {
    double t;    // Данное
public:
    void f ( );  // метод
    int a;       // данное
    enum { e1, e2, e3 } g;
private:
    struct inner { // вложенный класс
        int i, j;
        void g ( );
    };
    inner c;
};
...
X x;  x.a = 0; x.g = X::e1;
```

Действия над объектами классов

Над объектами класса можно производить следующие действия:

- присваивать объекты одного и того же класса (при этом производится почленное копирование членов-данных),
- получать адрес объекта с помощью операции &,
- передавать объект в качестве формального параметра в функцию,
- возвращать объект в качестве результата работы функции.
- осуществлять доступ к элементам объекта с помощью операции ‘.’, а если используется указатель на объект, то с помощью операции ‘->’.
- вызывать методы класса, определяющие поведение объекта.

Пример класса

```
...
class A {
  int a;
public:
  void set_a (int n);
  int get_a ( ) const { return a; }      // Константные методы класса
                                         // не изменяют состояние своего объекта
};

void A::set_a (int n) {
        a = n;
}

int main () {
        A obj1, obj2;
        obj1.set_a(5);
        obj2.set_a(10);
        cout << obj1.get_a ( ) << '\n';
        cout << obj2.get_a ( ) << endl;
        return 0;
}
```

АТД (абстрактный тип данных)

АТД называют тип данных с полностью скрытой (инкапсулированной) структурой, а работа с переменными такого типа происходит только через специальные, предназначенные для этого функции.

В C++ АТД реализуется с помощью классов (структур), в которых нет открытых членов-данных.

Класс A из предыдущего примера является абстрактным типом данных.

О терминологии

Оператор (statement) – действие, задаваемое некоторой конструкцией языка.

Операция (operator, для обозначения операций языка: +, *, =, и др.) – используются в выражениях.

Определение (описание) переменной (definition) – при этом отводится память, производится инициализация, определение возможно только 1 раз.

Объявление переменной (declaration) – дает информацию компилятору о том, что эта переменная где-то в программе описана.

Для преобразования типов используются два термина – **преобразование** (conversion) и **приведение** (cast).

Некоторые отличия C++ от C

- Введен логический тип ***bool*** и константы логического типа ***true*** и ***false*.**

- В C++ отсутствуют типы по умолчанию (например, обязательно int main () {…}).

- Локальные переменные можно описывать в любом месте программы, в частности внутри цикла for. Главное, чтобы они были описаны до их первого использования.

  По стандарту C++ переменная, описанная внутри цикла for, локализуется в теле этого цикла.

- В C++ переработана стандартная библиотека.

  В частности, в стандартной библиотеке C++ файл заголовков ввода/вывода назвается **<iostream>,** введены классы, соответствующие стандартным (консольным) потокам ввода – класс **istream** – и вывода – класс **ostream**, а также объекты **cin** (класса istream) и **cout** и **cerr** (класса ostream).

  Через эти объекты доступны операции ввода **>>** из стандартного потока ввода (например, cin >> x ;), и вывода **<<** в стандартный поток вывода (например, cout << ”string” << S << ‘\n’;), при использовании которых не надо указывать никакие форматирующие элементы.

Некоторые отличия C++ от C

- Функция main() не может быть вызвана рекурсивно в C++ (в C можно).

- В C++ нет отдельных пространств имен для тегов структур, перечислений.

```
typedef
    struct name {
        int a;
        char c;
     }  name;     /* допустимо в C, но не в C++ */
```

- Имя структуры становится полноценным именем типа

```
struct typename {
        int a;
        char c;
} ; // описали тип name

name  Obj; // описали объект типа name
struct name Obj1; // так тоже можно, в стиле C
```

Некоторые отличия C++ от C

- Символьные константы вида ’a’, ’1’ типа char в C++, но int в C.

- Пример программы, имеющей различный смысл в C и C++

```
int main () {
    return  6  //* комментарий */ 2
                ;  /* возвращает результат 3 в C, но 6 в C++
                   */
 }
```

Работа с динамической памятью

***int** *p,*m;*

p = ***new int** ;* или

p = ***new (nothrow) int** ;* или

p = ***new int** (1);* или

m = ***new int** [10];* – для массива из 10 элементов;

массивы, создаваемые в динамической памяти инициализировать нельзя;

…….

***delete** p;* или

***delete** [ ] m;* – для удаления массива;

Значения параметров функции по умолчанию

Пример:

```
void f (int a,  int b = 0, int c =1);
```

Обращения к функции:

```
f(3)        // a = 3, b = 0, c = 1;
f(3, 4)     // a = 3, b = 4, c = 1;
f(3, 4, 5)// a = 3, b = 4, c = 5.
```

Пространства имен

Пространства имен вводятся только на уровне файла, но не внутри блока.

```
namespace  std  {
        // объявления, определения
   }
Ex:    std::cout << std::endl;

namespace  NS  {
        char  name [ 10 ] ;
        namespace  SP  {
                int  var = 3;
        }
}
Ex:    ... NS::name ...;      NS::SP::var += 2;
```

```
#include <iostream>
using namespace std;

using NS::name;
```

Указатель this

Иногда для реализации того или иного метода возникает необходимость иметь указатель на «свой» объект, от имени которого производится вызов данного метода.

В C++ введено ключевое слово **this**, обозначающее «указатель на себя», которое можно трактовать как неявный параметр любого метода класса:

<имя класса> * **const this**;

***this** – сам объект.

Таким образом, любой метод класса имеет на один (первый) параметр больше, чем указано явно.

this, участвующий в описании функции, перегружающей **операцию**, всегда указывает на **самый левый** (в выражении с этой операцией) операнд операции.

В реальности поле **this** не существует (не расходуется память), и при сборке программы вместо **this** подставляется соответствующий адрес объекта.

Специальные методы класса

Конструктор – метод класса, который

- имеет имя, в точности совпадающее с именем самого класса;
- не имеет типа возвращаемого значения;
- **всегда** вызывается при создании объекта (сразу после отведения памяти под объект в соответствии с его описанием).

Деструктор – метод класса, который

- имеет имя, совпадающее с именем класса, перед первым символом которого приписывается символ **~** ;
- не имеет типа возвращаемого значения и параметров;
- **всегда** вызывается при уничтожении объекта (перед освобождением памяти, отведенной под объект).

Специальные методы класса

```
class A {
    ......
    public:
            A ( );          // конструктор умолчания
            A (A & y);   // A (const A & y); конструктор копирования (КК)
[explicit]   A (int x);     // конструктор преобразования; explicit запрещает
                            // компилятору неявное  преобразование   int  в  A
            A (int x, int y);
          // A (int x = 0, int y = 0);       // заменяет 1-ый, 3-ий и 4-ый
                                  // конструкторы
          ~A ();         // деструктор
          ......
};
Int main () {
    A a1,  a2 (10), a3 = a2;
    A a4 = 5,  a5 = A(7);  // Err!, т.к. временный объект не может быть
                                  // параметром для неконстантной ссылки в КК
                      // О.К., если будет      A (const A & y)
    A *a6 = new A (1);
}
```

Правила автоматической генерации специальных методов класса

- Если в классе **явно не описан никакой конструктор**, то конструктор умолчания генерируется автоматически с пустым телом в **public** области.

- Если в классе явно не описан конструктор копирования, то он **всегда генерируется автоматически** в **public** области с телом, реализующим почленное копирование значений полей-данных параметра конструктора в значения соответствующих полей-данных создаваемого объекта.

- Если в классе явно не описан деструктор, то он **всегда генерируется автоматически** с пустым телом в **public** области.

Класс Box

```
class Box {
    int  l;          // length – длина
    int w;           // width – ширина
    int h;           // height – высота
public:
    int volume () const { return   l * w * h ; }
    Box (int a, int b, int c ) { l = a; w = b; h = c; }
    Box (int s) { l = w = h = s; }
    Box ( ) { w = h = 1; l = 2; }
    int get_l ( ) const { return l; }
    int get_w ( ) const { return w; }
    int get_h ( ) const { return h; }
};
```

Автоматически сгенерированные конструктор копирования и операция присваивания:

```
Box (const Box & a) {  l = a.l;    w = a.w;    h = a.h; }
```

```
Box &  operator = ( const Box & a) { l = a.l;  w = a.w;  h = a.h;  return * this; }
```

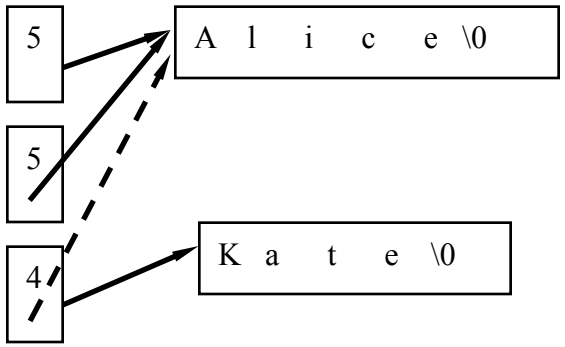
Конструктор копирования и операцию присваивания можно определить в классе явно вместо автоматически генерируемых.

Неплоский класс string

```
class string {
      char * p;    // здесь потребуется динамическая память,
      int size;
public:
      string (const  char * str);
      string (const string & a);
      ~string ( ) { delete [ ] p; }
       string & operator= (const string & a);
    ...
};

string :: string (const char * str) {
    p = new char [ ( size = strlen (str) ) + 1];
    strcpy (p, str);
}

string :: string (const string & a) {
    p = new char [ (size = a.size) + 1];
    strcpy (p, a.p);
}
```

Пример использования класса string

```
void f {
        string  s1 (“Alice”);   s1

        string  s2 = s1;        s2

        string  s3 (“Kate”);    s3
        ...
        s3 = s1;
}
```



{... s1...s2 {...s3...}...s1...s2}

Переопределение операции присваивания

```
string & string :: operator = (const string & a) {

    if  (this == & a)
            return * this;      //  если  a = a
    delete [ ] p;
    p = new char [ (size = a.size) + 1];
    strcpy (p, a.p);
    return * this;
}
```

При этом запись s1 = s2 эквивалентна s1.**operator**= (s2);

Композиция (строгая агрегация) объектов

```
class Point {
        int x;
        int y;
public:
        Point ( );
        Point ( int, int );
        ...
};
```

```
class Z {
        Point p;
        int z;
public:
        Z ( int  c ) { z = c; };
        ...
};
```

```
Z * z = new Z (1);          // Point ( );  Z(1);
delete z;                   // ~Z();   ~Point();
```

Использование **списка инициализации** при **описании** конструктора:

```
Z :: Z ( int c ) :  p (1, 2)  { z = c; }  или
Z :: Z ( int c ) :  p (1, 2),  z (c)  { }
```

Ссылки 1

Ссылочный тип данных задается так: <тип> &

Ссылка (reference) – переменная ссылочного типа.

Единственная **операция над ссылками** – **инициализация** (установление связи с инициализатором) при создании, при этом ссылка обозначает (именует) тот же **адрес** памяти, что и ее инициализатор (L-value выражение).

После описания и обязательной инициализации ссылку можно использовать точно так же, как и соответствующий ей инициализатор.

Фактически ссылка является синонимом своего инициализатора.

Ссылочный тип данных в C++ используется в следующих случаях:

а) **Описание переменных-ссылок** (локальных или глобальных).
Например,

```
int i = 5;
int & one_more_i = i;   //ссылка обязательно должна быть инициализирована
                        // one_more_i – синоним имени i ; &i ≡ &one_more_i;
i = one_more_i + 1;
one_more_i = i + 1;
cout << i << one_more_i;  //напечатается  7 7
```

Ссылки 2

b) **Передача параметров в функции по ссылке**.

Инициализация формального параметра ссылки происходит в момент передачи фактического параметра (L-value выражения), и далее все действия, выполняемые с параметром-ссылкой, выполняются с соответствующим фактическим параметром.

Пример:
```
void  swap (int & x, int & y) {
        int t = x;
        x = y;
        y = t;
    }
```
Пример обращения к функции swap:
```
int a = 5, b = 6;
swap (a, b);
```

c) **Возвращение результата работы функции в виде ссылки** - для более эффективной реализации функции - т.к. не надо создавать временную копию возвращаемого объекта – и в том случае, когда возвращаемое значение должно быть L-value-выражением.

Инициализация возвращаемой ссылки происходит при работе оператора **return**, операндом которого должно быть L-value выражение. **Не следует возвращать ссылку на локальный объект функции**, который перестает существовать при выходе из функции.

Пример:
```
int &  f( ) {
        int * p = new int(5);
        return *p;
    }
```
Пример обращения к функции f:
```
int & x = f();
```

Ссылки 3

d) Использование ссылок – членов-данных класса.

Инициализация поля-ссылки класса обязательно происходит через список инициализации конструктора, вызываемого при создании объекта.

Пример:

```
class A {
        int x;
    public:
        int & r;
        A( ) : r (x) {
                x = 3;
        }
                        A(const A &); // требуется явно описать !
                        A & operator= (const A&); // явно описать !
        ...
};
int main () {
        A a;
        ...
}
```

Ссылки 4
Константные ссылки

е) Использование **ссылок на константу**– формальных параметров функций (для эффективности реализации в случае объектов классов).

Инициализация параметра – ссылки на константу происходит во время передачи фактического параметра, который, в частности, может быть **временным объектом**, сформированным компилятором для фактического параметра-**константы**.

Пример:

```
struct A {
    int a;
    A( int  t = 0) { a = t; }
};
int f (const int & n, const A & ob)  {
    return n+ob.a;

}
int main () {
    cout << f (3, 5) << endl;
...
}
```

Временные объекты

Временные объекты создаются в рамках выражений (в частности, инициализирующих), где их можно модифицировать (применять неконстантные методы, менять значения членов-данных).

В общем случае «живут» временные объекты до окончания вычисления соответствующих выражений.

НО! Если инициализировать ссылку на константу временным объектом (в частности, передавать временный объект в качестве параметра для формального параметра – ссылки на константу), время его жизни продлевается до конца жизни соответствующей ссылочной переменной.

НЕЛЬЗЯ инициализировать неконстантную ссылку временным объектом (в частности, неконстантные ссылки – формальные параметры).

```
Пример:     struct  A {
            A (int);
            A (const A &);
            };
…           const A & r = A (1);        // если здесь и в КК убрать const,
A a1 = A (2);               // все эти конструкции будут
A a2 = 3;    …              //  ошибочными
```

Важно! Компилятор ВСЕГДА сначала проверяет синтаксическую и семантическую (контекстные условия) правильность, а затем оптимизирует!

Порядок выполнения конструкторов и деструкторов

При вызове **конструктора** класса выполняются:

1) конструкторы базовых классов (если есть наследование),
2) конструкторы умолчания всех вложенных объектов в порядке их описания в классе,
3) собственный конструктор (все поля класса уже проинициализированы, следовательно, их можно использовать).

Деструкторы выполняются в обратном порядке:

1) собственный деструктор (при этом поля класса ещё не очищены, следовательно, доступны для использования),
2) автоматически вызываются деструкторы для всех вложенных объектов в порядке, обратном порядку их описания в классе,
3) деструкторы базовых классов (если есть наследование).

Вызов конструктора копирования

1) явно,
2) в случае инициализации объектом того же типа:

```
Box a (1, 2, 3);
Box b = a;  // a – параметр конструктора копирования,
```

3) в случае инициализации временным объектом:

```
Box c = Box (3, 4, 5);
// сначала создается временный объект и вызывается
// обычный конструктор, а затем работает конструктор
// копирования для инициализации объекта c; если компилятор
// оптимизирующий,  вызывается только обычный
// конструктор с указанными параметрами;
```

4) при передаче параметров функции по значению (для инициализации локального объекта);

5) при возвращении результата работы функции в виде объекта;

6) при генерации исключения-объекта.

Вызов других конструкторов

- явно,
- при создании объекта (при обработке описания объекта),
- при создании объекта в динамической памяти (по new), при этом сначала в «куче» отводится необходимая память, а затем работает соответствующий конструктор,
- при композиции объектов наряду с собственным конструктором вызывается конструктор объекта – члена класса,
- при создании объекта производного класса также вызывается конструктор и базового класса,
- при автоматическом приведении типа с помощью конструктора преобразования.

Вызов деструктора

1) явно,

2) при свертке стека – при выходе из блока описания объекта, в частности при обработке исключений, завершении работы функции;
3) при уничтожении временных объектов – сразу, как только завершается конструкция, в которой они использовались;
4) при выполнении операции delete для указателя на объект (инициализация указателя – с помощью операции new), при этом сначала работает деструктор, а затем освобождается память.
5) при завершении работы программы при удалении глобальных/статических объектов.

Конструкторы вызываются в порядке определения объектов в блоке. При выходе из блока для всех автоматических объектов вызываются деструкторы, в порядке, противоположном порядку выполнения конструкторов.

Друзья класса

Друг класса – это функция, не являющаяся членом этого класса, но имеющая доступ к его **private** и **protected** членам.

Своих друзей класс объявляет сам в любой зоне описания класса с помощью служебного слова **friend**.

Функция-друг может быть описана внутри класса.

Если функций, имена которых совпадают с объявленной в классе функцией-другом, несколько, то другом считается только та, у которой в точности совпадает прототип.

Другом класса может быть:

- обычная функция: **friend void** f (...);
- функция-член другого класса: **friend void** Y::f (..);
- весь класс: **friend class** Y;

Свойства друзей класса

Дружба не обладает ни наследуемостью, ни транзитивностью.

Примеры:

```
class A {
    friend class B;
    int a;
};
```

```
class B {
        friend class C;
       };
```

```
class C {
    void f (A* p) {
        p -> a++;  // ошибка, нет доступа к закрытым членам класса A
    }
};
```

```
class D: public B {
    void f (A* p) {
        p -> a++; // ошибка, нет доступа к закрытым членам класса A

    }
};
```

Использование функций - друзей класса

```
class X {
        int a;
        friend void fff ( X *, int); // здесь нет this !
    public:
        void mmm (int);
};

void fff ( X * p, int i) {
        p -> a = i;
}

void X::mmm (int i) {
        a = i;
}

void f () {
        X obj;
        fff (&obj, 10);
        obj.mmm (10);
}
```

Преимущества использования друзей класса

1. Эффективность реализации (можно обходить ограничения доступа, предназначенные для обычных пользователей класса).

2. Функция-друг нескольких классов позволяет упростить интерфейс этих классов.

3. Функция-друг допускает преобразование своего первого параметра-объекта, а метод класса – нет.

Перегрузка операций

- Для перегрузки встроенных операций C++ используется ключевое слово **operator**.

- Перегружать операцию можно с помощью

  - метода класса,

  - внешней функции, в частности, функции-друга (что менее эффективно).

- Нельзя перегружать:

  ‘.’ , ‘::’ , ‘?:’ , ‘.*’ , sizeof, и typeid

Пример 1

```
class complex {
        double re, im;
    public:
        complex (double r = 0, double i = 0)  {       re = r;
                                                      im = i;
        }
        complex operator+ (const complex & a) {
            complex temp (re + a.re, im + a.im);
            return temp;
        }  ...
     // operator double () { return re; } – функция преобразования
};
int main () {
    complex x (1, 2), y (5, 8), z;
    double t = 7.5;
    z = x + y;  // O.K.:   x.operator+ (y);
    z = z + t;  // O.K.:    z.operator+ (complex (t)); если есть ф-я преобр., то
                    // неоднозначность: '+' для  double  или перегруженный
    z = t + x;  // Er.! – т.к. первый операнд по умолчанию – типа complex.
}
```

Пример 2

```
class complex {
        double re, im;
    public:
        complex (double r = 0, double i = 0) {
                re = r;
                m = i;
        }
        friend complex operator+ (const complex & a, const complex & b);
        ...
};
complex operator+ (const complex & a, const complex & b) {
                complex temp (a.re + b.re, a.im + b.im);
                return temp;
}
int main () {
        complex x (1, 2), y (5, 8), z;
        double t = 7.5;
        z = x + y;  // O.K. – operator+ (x, y);
        z = z + t;   // O.K. – operator+ (z, complex (t));
        z = t + x;   // O.K. – operator+ (complex (t), x);
}
```

Пример 3

```
class complex {
        double re, im;
    public:
        friend complex operator * (const complex & a,  double b);
        ...
};
complex operator * (const complex & a, double b) {
                    complex temp (a.re * b,  a.im * b);
                    return temp;
}
int main () {
    complex x (1, 2),  z;
    double t = 7.5;
    z = x * t;       // O.K. – operator* (x, t);
    z = t * x;       // Er.! т.к. нет функции преобразования x --> double, но
                     // если бы была, была бы неоднозначность:
                     // * - из double или из complex
}
```

В таких случаях обычно определяют еще одного друга с прототипом:

```
        complex operator * (double b,  const complex & a);
```

Замечания

- n-местные операции перегружаются
  a) методом с (n-1) параметром,
  b) внешней функцией с n параметрами;

- в любом случае сохраняется приоритет, ассоциативность и местность операций;

- операции

  = , [ ] , ( ) и ->

  можно перегрузить **только** нестатическими методами класса, что гарантирует, что первым операндом будет сам объект, к которому операция применяется

Особенности перегрузки операций ++ и --

complex x;

<u>префиксная ++:</u> ++ x; эквивалентно записи x.**operator** ++ ();

```
complex & operator ++ () {
    re = re + 1;
    im = im + 1;
    return *this;
}
```

<u>постфиксная ++:</u> x ++; эквивалентно записи x.**operator** ++ (0);

```
complex operator ++ (int) {
    complex c = * this;
    re = re + 1;
    im = im + 1;
    return  c;
}
```

Перегрузка операции →

Операцию → перегружают **методом класса**, объекты которого играют роль «умных» указателей на объекты другого класса.

Операцию → можно считать постфиксной *унарной*, поскольку преобразование объекта класса в указатель не зависит от конкретного поля, на которое он указывает.

Пример:

```
struct T { int f;};
class Tptr {
        T* pt;
public:
        Tptr () { pt = new T; }
        T* operator →() {
                return pt;
        }
        ~Tptr () { delete pt; }
};
```

Метод ***operator →()*** *обязан возвращать либо указатель, либо объект класса, для которого также перегружена операция* →. Последним в цепочке перегруженных операций → должен быть метод, возвращающий указатель на объект некоторого класса.

Пример перегрузки операции «( )» и операции вывода «<<»

```
class Matrix {
        double M [ 3 ] [ 3 ];
public:
        Matrix ();
        double & operator ( ) (int i, int j) const {
                return M [ i ] [ j ];
        }
     friend ostream & operator << (ostream & s, const Matrix & a)
    {           for (int i = 0;  i < 3 ;  i ++) {
                        for (int j = 0; j < 3; j ++)
                                s  <<  a (i,  j)  <<  ' ';
                        s  <<  endl;
                }
                return s;
    }
};
```

Перегрузка функций

О перегрузке можно говорить только для функций из одной области видимости!

Алгоритм поиска и выбора функции:

1. Выбираются только те перегруженные (одноименные) функции, для которых фактические параметры соответствуют формальным по количеству и типу (приводятся с помощью каких-либо преобразований).

2. Для каждого параметра функции (отдельно и по очереди) строится множество функций, оптимально отождествляемых по этому параметру (best matching).

3. Находится пересечение этих множеств:

   - если это ровно одна функция – она и является искомой,

   - если множество пусто или содержит более одной функции, генерируется сообщение об ошибке.

Пример 1

class X { **public**: X(**int**);...};

class Y {<нет конструктора с параметром типа **int**>...};

```
void f (X, int);          // 1-й параметр ‘да’ , 2-й  ‘да’
void f (X, double);       // 1-й параметр ‘да’ , 2-й  ‘нет’
void f (Y, double);       //отбрасывается на 1-м шаге

void g () {... f (1,1); ...}
```

Т.к. в пересечении множеств, построенных для каждого параметра, одна функция f (X, **int**) – вызов разрешим.

Пример 2

```
struct X { X (int);...};

   void f (X, int);  // 1 –й ‘нет’   2-й ‘да’
   void f (int, X);  // 1 –й ‘да’    2-й ‘нет’

   void g () {... f (1,1); ...}
```

Т.к. пересечение множеств, построенных для каждого параметра, пусто – вызов неразрешим.

Пример 3

```
void f (char);
void f (double);

void g () {... f (1); ...}   // ?
```

Не всегда просто выполнить шаг 2 алгоритма, поэтому стандартом языка C++ закреплены правила сопоставления формальных и фактических параметров при выборе одной из перегруженных функций.

Правила для шага 2 алгоритма выбора перегруженной функции

а) Точное отождествление.

б) Отождествление при помощи расширений.

в) Отождествление с помощью стандартных преобразований.

г) Отождествление с помощью преобразований, определенных пользователем.

д) Отождествление по **...** .

a) Точное отождествление

- точное совпадение,

- совпадение с точностью до **typedef**,

- тривиальные преобразования:

T[ ] <--> T *,
T <--> T&,
T --> **const** T, // в одну сторону!
T(...) <--> (T*)(...) .

Пример:

```
void f (float);
void f (double);
void f (int);

void g () {...      f (1.0);      // f (double)
                    f (1.0F);     // f (float)
                    f (1);        // f (int);        ...
}
```

б) Отождествление при помощи расширений

- Целочисленные расширения:

  char, **short** (**signed** и **unsigned**), **enum**, **bool** --> **int** (**unsigned int**, если не все значения могут быть представлены типом **int** – тип **unsigned short** не всегда помещается в **int**);

- Вещественное расширение: **float --> double**

Пример:

```
void f (int);

void f (double);
```

```
void g () {
        short aa = 1;
        float ff = 1.0;
        f (ff);          // f (double)
        f (aa);          // f (int)
}
```

в) Отождествление с помощью стандартных преобразований

- все оставшиеся стандартные целочисленные и вещественные преобразования, которые могут выполняться неявно, а также преобразование объекта производного класса к объекту однозначного доступного базового класса

- преобразования указателей:

0 --> любой указатель,

любой указатель –> **void***,

derived* --> base* - для однозначного доступного базового класса;

Пример:

```
void f (char);
void f (double);

void g () { ... f (0);      // неоднозначность, т.к.
                            // преобр. int --> char  и
                            // int --> double равноправны
}
```

г) Отождествление с помощью пользовательских преобразований

- с помощью конструкторов преобразования

- с помощью функций преобразования

Пример:

```
struct S {
        S (long);                   // long --> S
        operator int ();            // S --> int        ...
};

void f (long);          void g (S);             void h (const S&);
void f (char*);         void g (char*);         void h (char*);

void ex (S &a) {
    f (a);  // O.K.  f ( (long) ( a.operator int()) );   т.е. f (long)     -   на шаге г).
    g (1);  // O.K.  g ( S ( (long) 1) );                 т.е. g (S)        -   на шаге г).
    g (0);  // O.K.  g ( (char*) 0);                      т.е. g (char*)    -   на шаге в)!!!
    h (1);  // O.K.  h ( S ( (long) 1) );                 т.е. h (const S&) -   на шаге г).
}
```

Замечание 1

Пользовательские преобразования применяются **неявно** только в том случае, если они **однозначны**!

Пример:

```
class Boolean {
        int b;
    public:
        Boolean operator+ (Boolean);
        Boolean (int i) { b = i != 0;}
        operator int () { return b; }
...
};
void g () {
        Boolean b (1), c (0); // O.K.
        int k;
        c = b + 1; // Er.!  т.к. может интерпретироваться двояко:
                   // b.operator int () +1 – целочисленный '+' или
                   // b.operator+ (Boolean (1)) – Boolean '+'
        k = b + 1; // Er.!     -- " --
}
```

Замечание 2

Допускается не более **одного пользовательского** преобразования для обработки одного вызова для одного параметра!

Пример:

```
class X { ... public: operator int (); ... };
class Y { ... public: operator X (); ... };

void f () {
        Y a;
        int b;
        b = a; // Er.! , т.к. требуется   a.operator X ().operator int ()
        ...
}
```

Но! **явно** можно делать любые преобразования, явное преобразование сильнее неявного.

д) Отождествление по **...**

Пример 1:

```
class Real {
    public:
        Real (double);
        ...
};

void f (int, Real);
void f (int, ...);          // можно и без ‘,’

void g () {
        f (1,1);            // O.K.  f (int, Real);
        f (1, “Anna”);  // O.K.  f (int, ...);
}
```

Пример 2:

Многоточие может приводить к неоднозначности:

```
void f (int);
void f (int ...);

void g () {...
    f (1);  // Er.! т.к. отождествление по
            // первому параметру дает
            // обе функции.
}
```

Одиночное наследование

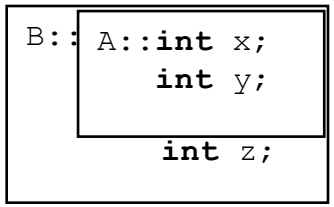
Конструкторы, деструкторы и operator= не наследуются

class < имя derived-cl > : < способ наследования > < имя base-cl > {...};

Пример:

```
struct A { int x; int y; };          struct B : A { int z; };
                                     class C : protected A { int z; };
        .........
A a;                                 A * pa;
B b;                                 C  c,  * pc = &c;
b.x = 1;                             pc -> z;   // ошибка: доступ к закрытому полю
b.y = 2;                             pc -> x;   // ошибка: доступ к защищённому полю
b.z = 3;                             pa = ( A * ) pc;
a = b;                               pa -> x;   // правильно: поле A::x – открытое
```

```
A a, *pa;
B b, *pb;
pb = &b;
pa = pb;
pb = ( B* ) pa;
```



Сокрытие имён (hiding)

```
struct A {
        int f ( int x , int y);
        int g ();
        int h;
};
struct B : public A {
        int x;
        void f ( int x );
        void h ( int x );
};
.......
A  a,  *pa;
B  b,  *pb;
pb = &b;
pb -> f (1);                // вызывается B::f(1)
pb -> g ();                 // вызывается A::g()
pb -> h = 1;                // Err.! функция h(int) – не L-value выражение
pa = pb;   pa -> f (1);     // Err.! функция A::f(1) имеет 2 параметра
pb = &a;                    // Err.!  расширяющее присваивание // pb = (B*)&a
pb -> f (1);                // Возможна Err, если в f(1) используется x  из B
```

Видимость и доступность имен

Пример

```
int x;
void f (int a) { cout << “ :: f " << a << endl; }

class A {
    int x;
public:
    void f (int a) { cout << “ A:: f " << a << endl; }
};
class B : public A {
public:
    void f (int a) { cout << “ B:: f " << a << endl; }
    void g ();
};

void B::g() {
    f(1);           // вызов  B::f(1)
    A::f(1);
    ::f(1);         // вызов глобальной void f(int)
    //x = 2;        //ошибка!!! – осущ. доступ к закрытому члену класса A
}
```

Вызов конструкторов базового и производного классов. Пример.

```
class A { ... };

class B : public A {
public:
    B ();
    B (const B &); // есть явно описанный конструктор копирования
    ...
};
class C : public A {
public:
    // нет явно описанного конструктора копирования
    ...
};
int   main ( ) {
    B b1;            //  A (),   B ()
    B b2 = b1;       //  A (),   B (const B &)
    C c1;            //  A (),   C ()
    C c2 = c1;       //  A (const A &),   C (const C &)
    ...
}
```

Классы student и student5

```
class student {
    char * name;
    int year;
    double est;
public:
    student ( char* n, int y, double e);
    void print () const;
     ~student ();
};
```

```
class student5: public student {
        char * diplom;
        char * tutor;
public:
        student5 ( char* n, double e, char* d, char* t);
        void print () const;
        // эта print  скрывает print из базового класса
        ~student5 ();
};
```

```
student5:: student5 ( char* n, double e, char* d, char* t) : student (n, 5, e) {
    diplom = new char [strlen (d) + 1];
    strcpy (diplom, d);
    tutor = new char [strlen (t) + 1];
    strcpy (tutor, t);
}
```

```
student5 :: ~student5 () {

    delete [ ] diplom;
    delete [ ] tutor;
}
```

```
void  student5 :: print () const {
        student :: print ();    // name, year, est
                cout << diplom << endl;
        cout << tutor << endl:
}
```

Использование классов student и student5

```
void f ( ) {
    student  s (“Kate”, 2, 4.18),     * ps = & s;
    student5  gs (“Moris”, 3.96, “DIP”, “Nick”), * pgs = & gs;

    ps -> print();              // student :: print ();
    pgs -> print();             // student5 :: print ();

    ps = pgs;       // base = derived – допустимо с преобразованием по
                    // умолчанию.
    ps -> print();  // student :: print () – функция выбирается статически
                    // по типу указателя.
```

Виртуальные методы

Метод называется **виртуальным**, если при его объявлении в классе используется квалификатор **virtual**.

Класс называется **полиморфным**, если содержит хотя бы один виртуальный метод.

Объект полиморфного класса называют **полиморфным** объектом.

Чтобы динамически выбирать функцию print () по типу объекта, на который ссылается указатель, переделаем наши классы т.о.:

```
class student {…
public:
    ...
    virtual void print ( ) const ;
};
class student5 : public student {…
public:
    ...
    [virtual] void print ( ) const ;
};
```

Тогда: ps = pgs;
ps -> print(); // student5 :: print () – ф-я выбирается динамически по типу
// объекта, чей адрес в данный момент хранится в указателе

Виртуальные деструкторы

Совет: для полиморфных классов делайте деструкторы виртуальными!

```
void f () {
    student * ps = new  student5 (“Moris”, 3.96, “DIP”, “Nick”);
            …
    delete ps; // вызовется ~student, и не вся память очистится
}
```

Но если:

```
    virtual ~student ();    и
    [virtual] ~student5 ();
```

то вызовется ~student5(), т.к. сработает динамический полиморфизм.

Механизм виртуальных функций (механизм динамического полиморфизма)

1. !Виртуальность функции, описанной с использованием служебного слова **virtual** не работает сама по себе, она начинает работать, когда появляется класс, производный от данного, с функцией с **таким же прототипом**.

2. Виртуальные функции выбираются по типу объекта, на который ссылается указатель (или ссылка).

3. У виртуальных функций должны быть одинаковые прототипы. Исключение составляют функции с одинаковым именем и списком формальных параметров, у которых тип результата есть указатель или ссылка на себя (т.е. соответственно на базовый и производный класс).

4. Если виртуальные функции отличаются только типом результата (кроме случая выше), генерируется ошибка.

5. Для виртуальных функций, описанных с использованием служебного слова **virtual,** с разными прототипами работает только механизм сокрытия имен, замещения не происходит

Абстрактные классы

Абстрактным называется класс, содержащий хотя бы одну **чистую виртуальную** функцию.

Чистая виртуальная функция имеет вид: **virtual** тип_рез имя (сп_фп) **= 0**;

Пример:

```
class shape {
public:
        virtual double area () = 0;
};
class rectangle: public shape {
        double height, width;
public:
        double area () {
                return    height * width;
        }
};
class circle: public shape {
        double radius;
public:
        double area () {
                return    3.14 * radius * radius;
        }
};
```

```
#define N 100
....
shape* p [ N ];
double total_area = 0;
          ....
for (int i =0; i < N; i++)
          total_area += p[i] -> area();
          ....
```

Интерфейсы

Интерфейсами называют абстрактные классы, которые

- не содержат нестатических полей-данных, и
- все их методы являются открытыми чистыми виртуальными функциями.

Реализация виртуальных функций

```
class A {
    int a;
public:
    virtual void f ();
    virtual void g (int);
    virtual void h (double);
};
class B : public A {
public:
    int b;
    void g (int);
    virtual void m (B*);
};
```

```
class C : public B {
public:
        int c;
        void h (double);
        virtual void n (C*);
};
```

Тогда C c; ~

a
pvtbl
b
c

vtbl для c ~

&A:: f
&B:: g
&C:: h
&B:: m
&C:: n

```
C c;
A *p = &c;
p -> g (2);   ~   (* ( p -> pvtbl [1]) ) (p, 2); // p = this
```

Виртуальные функции. Пример 1.

```
class X {
public:
    void g ( ) {
        cout << "X::g\n";
        h ( );
    }
    virtual void f() {
        g ( );
        h ( );
    }
    virtual void h ( ) {
        cout << "X::h\n";
    }
};
```

```
class Y : public X {
public:
    void g ( ) {
        cout << "Y::g\n";
        h ( );
    }
        virtual void f ( ) {
        g ( );
        h ( );
    }
    virtual void h ( ) {
         cout << "Y::h\n";
    }
};
```

```
int main () {
        Y b;
        X  *px = &b;
        px -> f();          //  Y::g    Y::h    Y::h
        px -> g();          //  X::g    Y::h
        return 0;
}
```

Виртуальные функции. Пример 2.

```
struct A {
    virtual int f (int x, int y) {
        cout << "A : :f (int, int) \n";
        return x + y;
    }
    virtual void f ( int x ) {
        cout << "A :: f( ) \n";
    }
};

struct B :  A {
    void f ( int x ) {
        cout << "B :: f( ) \n";
    }
};

struct C :  B {
        virtual int f (int x, int y) {
        cout << "C :: f (int, int) \n";
        return x + y;
    }
};
```

```
int main () {
        B b,  *pb = &b;
        C c;
        A  * pa = &b;
        pa -> f (1);      // B::f();
                pa -> f (1, 2);  // A::f(int, int)
    //pb -> f (1, 2);  // Err.! Эта f не видна

        A & ra = c;
        ra.f(1,1);        // C::f(int, int)

                B & rb = c;
    //rb.f(0,0);   // Err.! Эта f не видна
         return 0;
}
```

Виртуальные функции. Пример 3.

```
struct B {
    virtual B& f () { cout << " f ( ) from B\n"; return *this;}
    virtual void g (int x, int y = 7) { cout << "B::g\n"; }
};

struct D : B {
    virtual D& f ( ) { cout << " f ( ) from D\n"; return *this; }
    virtual void g (int x, int y) { cout << "D::g y = " << y << endl; }
};
    int main () {
        D  d;
        B  b1,  *pb = &d;
        pb -> f();              // f ( ) from D
        pb -> g(1);             // D::g  y = 7
        pb -> g(1,2);           // D::g  y = 2
        return 0;
    }
```

Если значение параметра y по умолчанию будет в функции g класса D, а в базовом не будет, компилятор на вызов pb - > g(1) выдаст ошибку.

Множественное наследование

class A { **...** };
class B { **...** };
class C : **public** A, **protected** B { **...** };

!!! Спецификатор доступа распространяется только на один базовый класс; для других базовых классов начинает действовать принцип умолчания.

!!! Класс не может появляться как непосредственно базовый дважды:

class C : **public** A, **public** A { ... }; - Er.!

но может быть более одного раза непрямым базовым классом:

class L { **public**: **int** n; **...** };

class A : **public** L { ... };

class B : **public** L { ... };

class C : **public** A, **public** B { ... **void** f (); ... };

Здесь **решетка смежности** такая: L <-- A <-- C --> B --> L .

При этом может возникнуть неоднозначность из-за «многократного» базового класса.

О доступе к членам производного класса

void C::f () { ... n = 5; ...} // Er.! – неясно, чье n, но

void C::f () { ...A::n = 5; ...} // O.K.! , либо B::n = 5;

Имя класса в операции разрешения видимости (A или B) – это указание, в каком классе в решетке смежности искать заданное имя.

О преобразовании указателей

Указатель на объект производного класса может быть неявно преобразован к указателю на объект базового класса, только если этот базовый класс является **однозначным** и **доступным** !!!

Продолжение предыдущего примера:

```
void g ( ) {
        C* pc = new C;
        L* pl = pc;          // Er.! – L  не является однозначным,
        pl = (L*) pc;        // Er.! – явное преобразование не помогает,
                              // но возможно:
        pl = (L*) (A*) pc;  // либо pl = (L*) (B*) pc; O.K.!
```

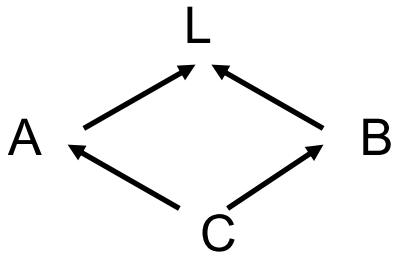
Базовый класс считается **доступным** в некоторой области видимости, если доступны его public-члены.

```
class B   { public:  int a; ... };
class D : private B { ... };

void g () {
        D* pd = new D;
        B* pb = pd;      // Er.! – в g() public-члены B, унаследованные
                         // D,  недоступны, такое преобразование
                         // может осуществлять только
                         // функция-член D, либо друзья D.
}
```

Виртуальные базовые классы

```
class L   { public:  int  n ;  ... };
class A :  virtual public  L  { ... };
class B :  virtual public  L  { ... };
class C : public A, public B  { ... void f (); ... };
```

Теперь решетка смежности будет такой:



и теперь допустимо:

```
void C :: f () { ... n = 5; ...} // О.К.! – n в одном экземпляре

void g () {
        C* pc = new C;
        L* pl = pc;          // O.K.! – появилась однозначность.
}
```

Правила выбора имен в производном классе.

1 шаг: контроль **однозначности** (т.е. проверяется, определено ли анализируемое имя в одном базовом классе или в нескольких); при этом контекст не привлекается, совместное использование (в одном из базовых классов) допускается.

2 шаг: если однозначно определенное имя есть имя перегруженной функции, то пытаются **разрешить** анализируемый вызов (т.е. найти best-maching).

3 шаг: если предыдущие шаги завершились успешно, то проводится контроль **доступа**.

Неоднозначность из-за совпадающих имен в различных базовых классах.

```
class A {
   public:
        int a;
        void (*b) ( );
        void f ( );
        void g ( );  ...
};
```

```
class B {
        int a;
        void b ( );
        void h (char);
public:
        void f ( );
        int g;
        void h ( );
        void h (int);  ...
};
```

```
class C : public A, public B  { ... };
```

Пример.

```
void gg (C* pc) {

        pc --> a = 1;       // Er.! – A::a  или  B::a

        pc --> b();         // Er.! – нет однозначности

        pc --> f ();        // Er.! – нет однозначности

        pc --> g ();        // Er.! – нет однозначности,
                            // контекст не привлекается!

        pc --> g = 1;       // Er.! – нет однозначности,
                            // контекст не привлекается!

        pc --> h ();        // O.K.!

        pc --> h (1);       // O.K.!

        pc --> h  ('a');    // Er.! – доступ в последнюю очередь

        pc --> A::a = 1;    // O.K.! – т.е. снимаем неоднозначность
                            // с помощью  операции «::»
}
```

Статические члены класса.

- Статические члены-данные и члены-функции описываются в классе с квалификатором **static.**

- Статические члены-данные существуют в одном экземпляре и доступны для всех объектов данного класса.

- Статические члены класса существуют **независимо от конкретных экземпляров класса**, поэтому обращаться к ним можно еще до размещения в памяти первого объекта этого класса, а также изменять, используя, например, имя константного объекта класса.

- **Необходимо** предусмотреть выделение памяти под каждый статический член-данное класса (т.е. описать его вне класса с возможной инициализацией), т.к. при описании самого класса или его экземпляров память под статические члены-данные не выделяется.

- Доступ к статическим членам класса (наряду с обычным способом) можно осуществлять через имя класса (без указания имени соответствующего экземпляра) и оператор разрешения области видимости «**::**».

Пример.

```
class A {
public:
        static int  x;
        static void f (char c);
};

int A::x; // !!! – размещение статического объекта в памяти

 void g() {
        …
        A::x = 10;
        …
         A::f ('a');
        …
}
```

Особенности использования статических методов класса

- Статических методы класса используются, в основном, для работы с глобальными объектами или статическими полями данных соответствующего класса.

- Статические методы класса не могут пользоваться нестатическими членами-данными класса.

- Статические методы класса не могут пользоваться указателем this , т.е. использовать объект, от имени которого происходи обращение к функции.

- Статические методы класса не могут быть виртуальными и константными (inline - могут).

Средства обработки ошибок. Исключения в C++

Обработка **исключительных ситуаций** в C++ организуется с помощью ключевых слов **try**, **catch** и **throw**.

Операторы программы, при выполнении которых необходимо обеспечить обработку исключений, выделяются в **try-catch** - блок.

Если ошибка произошла внутри **try**-блока (в частности, в вызываемых из **try**-блока функциях), то соответствующее **исключение** должно генерироваться с помощью оператора **throw**, а перехватываться и обрабатываться в теле одного из обработчиков **catch,** которые располагаются непосредственно за **try**-блоком.

Исключение - объект некоторого типа, в частности, встроенного.

Операторы, находящиеся после места генерации ошибки в **try**-блоке, игнорируются, а после обработки исключения управление передается первому оператору, находящемуся за обработчиками исключений. **try-catch**-блоки могут быть вложенными.

Общий синтаксис **try-catch** блока:

try {
….. **throw** исключение; …..
}
catch (type) {---/*__throw__;*/}
catch (type arg) {---/*__throw__;*/}
…
catch (…) {---/*__throw__;*/}

Перехват исключений

С каждым try-блоком может быть связано несколько операторов **catch**. Они просматриваются по очереди сверху вниз.

Какой именно обработчик **catch** будет использоваться, зависит от типа сгенерированного исключения.

Выбирается первый обработчик с типом параметра, **совпадающим** с типом исключения. **Ловушки с базовым типом** (или с указателем или ссылкой на базовый тип) **перехватывают все исключения с производным типом** (или его адресом), т.е. производные типы должны стоять раньше базовых типов.

Если исключение перехвачено каким-либо обработчиком **catch**, аргумент arg получает его значение, которое затем можно использовать в теле обработчика. Если доступ к самому исключению не нужен, то в операторе **catch** можно указывать только его тип.

Существует специальный вид обработчика, перехватывающего любые исключения - **catch (…){---}**. Естественно, он должен находиться в конце последовательности операторов **catch**.

Если для сгенерированного исключения в текущем **try**-блоке нет подходящего обработчика, оно перехватывается объемлющим try-блоком (main()→f()→g()→h()).

Если же подходящего обработчика так и не удалось найти, может произойти **ненормальное (аварийное)** завершение программы. При этом вызывается стандартная библиотечная функция **terminate ()**, которая в свою очередь вызывает функцию **abort ()**, чего лучше избегать.

Пример

```
class A {
public:
        A () {cout << “Constructor of A\n”;}
       ~A () {cout << “Destructor of A\n”;}
};
class Error {};
class Error_of_A : public Error {};
void f () {
        A a;
        throw 1;
        cout << “This message is never printed” << endl;
}
int main () {
        try {
                f ();
                throw   Error_of_A();
        }
        catch (int) { cerr << “Catch of int\n”; }
        catch (Error_of_A) { cerr << “Catch  of   Error_of_A \n”; }
        catch (Error) { cerr << “Catch  of   Error\n”; }
        return 0;
}
```

Результат работы программы на предыдущем слайде.

Constructor of A
Destructor of A // т.к. в f обработчика нет, поиск идет дальше,
// но при выходе из f вызывается деструктор
// локальных объектов.
Catch of int

Если поменять строки внутри try, получим:

Catch of Error_of_A

Если закомментировать строку

// **catch** (Error_of_A) { cerr << “Catch of Error_of_A \n”; },

получим

Catch of Error

Пример использования классов исключений

class MathEr {...**virtual void** ErrProcess();...};

class Overflow : **public** Math Er {... **void** ErrProcess();...};

class ZeroDivide : **public** Math Er {... **void** ErrProcess();...};

...

Через параметры конструктора исключения можно передавать любую нужную информацию.

Если использовать виртуальные функции, можно после **try**-блока задать единственный обработчик **catch**, имеющий параметр типа базового класса, но перехватывающий и обрабатывающий любые исключения:

```
try {    ...
}
catch (MathEr & m) {...  m. ErrProcess();  ...}
```

Организованная таким образом обработка исключений позволяет легко модифицировать программы.

Исключения, генерируемые в функциях

В заголовке функции можно указать типы исключений (через запятую), которые может генерировать функция (эту возможность удобно использовать при описании библиотечных функций):

*тип_рез имя_функции (список_арг) [**const**] **throw** (список_типов) { ... }*

Если список типов **пустой**, то функция не может генерировать **никаких** исключений.

Если же функция все-таки сгенерировала недекларированное исключение, вызывается библиотечная функция ***unexpected ()*** работающая аналогично функции *terminate().*

Использование аппарата исключений – единственный безопасный способ нейтрализовать ошибки в конструкторах и деструкторах, поскольку они не возвращают никакого значения, и нет другой возможности отследить результат их работы.

Если деструктор, вызванный во время свертки стека, попытается завершить свою работу при помощи исключения, то система вызовет функцию terminate(), что крайне нежелательно. Отсюда важное требование к деструктору: ни одно из исключений, которое могло бы появиться в процессе работы деструктора, не должно покинуть его пределы.

Действия, выполняемые с момента генерации исключения до завершения его обработки

- При генерации исключения (***throw*** *X*) создается объект-исключение – копия X (работает конструктор копирования). С этой копией будет работать выбранный далее обработчик; она существует до тех пор, пока обработка исключения не будет завершена.

- Для всех других объектов ***try***-блока, созданных к этому моменту, перед выходом из ***try***-блока освобождается память; при этом для объектов – экземпляров классов вызывается деструктор. То же делается и для уже созданных подобъектов: членов класса – объектов другого класса и баз. Этот процесс называют «раскруткой» («сверткой») стека.

- Если в списке обработчиков ***catch*** этого ***try***-блока найден подходящий, то выполняются его операторы; затем выполнение программы продолжается с оператора, расположенного за последним обработчиком этого **try**-блока.

- Если в списке данного **try**-блока не нашлось подходящего обработчика, то поиск продолжается в динамически объемлющих ***try***-блоках (при этом процесс свертки стека продолжается).

- Если подходящего обработчика так и не нашлось, то вызывается функция *terminate()* и выполнение программы прекращается.

Механизм RTTI (Run-Time Type Identification)

Механизм RTTI состоит из трех частей:

1. операция **dynamic_cast**
   (в основном предназначена для получения указателя на объект производного класса при наличии указателя на объект полиморфного базового класса);

2. операция **typeid**
   (служит для идентификации точного типа объекта при наличии указателя на полиморфный базовый класс);

3. структура **type_info**
   (позволяет получить дополнительную информацию, ассоциированную с типом).

Для использования RTTI в программу следует включить заголовок <typeinfo>.

(1) Операция ***dynamic_cast*** реализует приведение типов (**указателей или ссылок**) полиморфных классов в динамическом режиме.

Синтаксис использования операции *dynamic_cast* :

dynamic_cast **<** *целевой тип* **>** ***(выражение)***

Если даны **два полиморфных класса B и D** (причем D – производный от B), то *dynamic_cast* всегда может привести D* к B*.

Также *dynamic_cast* может привести B* к D*, но только в том случае, если объект, на который указывает указатель, действительно является объектом типа D (либо производным от него)!

При неудачной попытке приведения типов результатом выполнения *dynamic_cast* является 0, если в операции использовались указатели.

Если же в операции использовались ссылки, генерируется исключение типа ***bad_cast***.

Пример: пусть Base – полиморфный класс, а Derived – класс, производный от Base.

```
Base * bp,  b_ob;
Derived * dp, d_ob;

bp = & d_ob;
dp = dynamic_cast  <Derived *> (bp);
if (dp)
     cout << «Приведение типов прошло успешно»;

bp = &b_ob;
dp = dynamic_cast  <Derived *> (bp);
if (!dp)
     cout << «Приведения типов не произошло»;
```

(2)-(3) Информацию о типе объекта можно получить с помощью операции ***typeid***.

Синтаксис использования операции ***typeid*:**

typeid (выражение) или
typeid (имя_типа)

Операция *typeid* возвращает ссылку на **объект класса *type_info***, представляющий либо тип объекта, обозначенного заданным выражением, либо непосредственно заданный тип.

В классе *type_info* определены следующие открытые члены:

bool operator == (**const** type_info & объект); // для сравнения типов
bool operator != (const type_info & объект); // для сравнения типов
bool before (const type_info & объект); // для внутреннего использования
const char * name ( ); //возвращает указатель на имя типа

Оператор ***typeid*** наиболее полезен, если в качестве аргумента задать указатель полиморфного базового класса, т.к. с его помощью во время выполнения программы можно определить тип реального объекта, на который он указывает. То же относится и к ссылкам.

typeid часто применяется к разыменованным указателям (*typeid (* p)*). Если указатель на полиморфный класс p == NULL, то будет сгенерировано исключение типа ***bad_typeid***.

Пример.

```
class Base {
        virtual void f ( ) {...};
};
class Derived1: public Base {...
};
class Derived2: public Base {...
};
int main ( ) {
        int i;
        Base *p, b_ob;
        Derived1 ob1;
        Derived2 ob2;
        cout << «Тип i - » << typeid ( i ) . name ( ) << endl;
        p = & b_ob;
        cout << “p указывает на объект типа ” << typeid ( * p ) . name ( ) << endl;
        p = & ob1;
        cout << “p указывает на объект типа ” << typeid ( * p ) . name ( ) << endl;
        p = & ob2;
        cout << “p указывает на объект типа ” << typed ( * p ) . name ( ) << endl;
        if ( typeid (ob1) == typeid (ob2) )
                cout << “Тип объектов ob1 и ob2 одинаков\n”;
        else
                cout << “Тип объектов ob1 и ob2 не одинаков\n”;
        return 0;
}
```

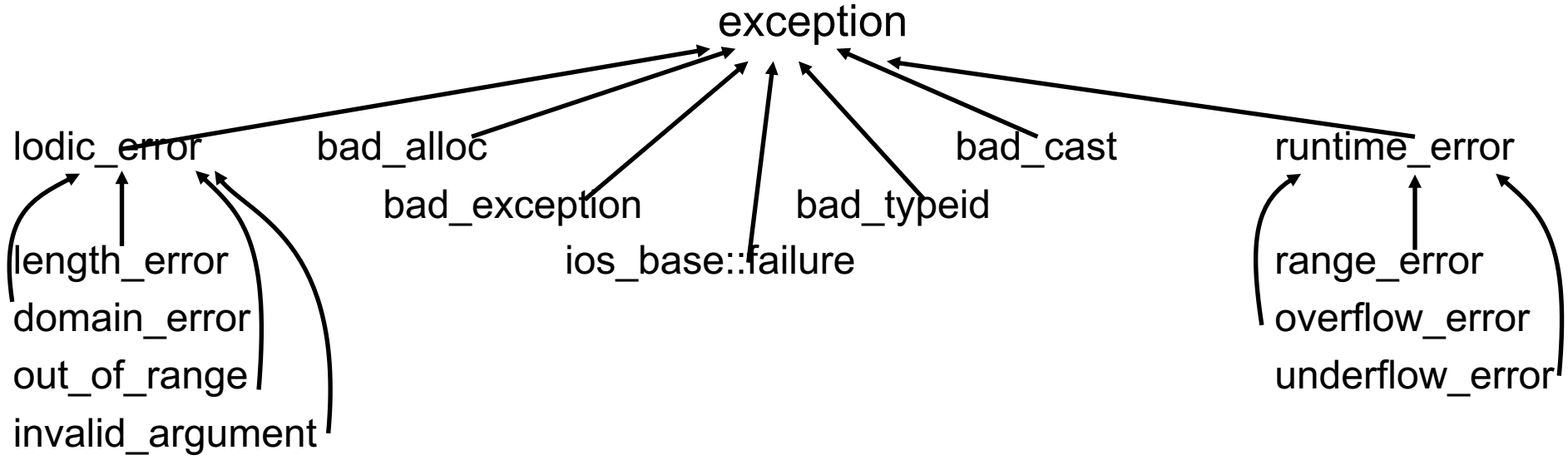
Стандартные исключения

Текст по генерации стандартных исключений вставляется компилятором.

Стандартные исключения объединены в иерархию классов, в вершине которой находится стандартный абстрактный библиотечный класс ***exception***, описанный в <stdexcept> :

```
class exception {
public:
        exception () throw ();
        exception (const exception &) throw ();
        exception & operator=(const exception &) throw ();
        virtual ~exception () throw ();
        virtual const char * what () const throw();
       ...
};
```

Иерархия классов стандартных исключений



Из этих классов исключений мы рассматриваем только исключения

bad_cast и **bad_typeid**, генерируемые соответственно при неверной работе операций dynamic_cast и typeid, и расположенные в файле <typeinfo>,

out_of_range, генерируемое методом at () контейнеров STL, и расположенное в файле <stdexcept> ,

bad_alloc, генерируемое операцией **new** при невозможности выделения динамической памяти и расположенное в файле <new>.

Чтобы операция new при ошибке выделения динамической памяти возвращала 0, надо использовать следующую ее форму:

T * p = **new** (nothrow) T;

Пример использования стандартных исключений

```
void f () {
        try {   ...
                // использование стандартной библиотеки
        }
        catch (exception & e) {
                cout << “Стандартное исключение” << e.what() << ‘\n’;
        }
        catch (...) {
                cout << “Другое исключение” << ‘\n’;
                ...
        }
}
```

Иерархию классов стандартной библиотеки можно брать за основу для своих исключений.

Шаблоны

1. Механизм шаблонов реализует в C++ **параметрический полиморфизм.**

2. Шаблон представляет собой предварительное описание функции или класса, конкретное представление которых зависит от параметров шаблона.

3. Для описания шаблонов используется ключевое слово **template**, вслед за которым указываются аргументы (параметры шаблона), заключенные в угловые скобки.

4. Параметры шаблона перечисляются через запятую, и могут быть:

а) объектами следующих типов:

- **целочисленного,**
- **перечислимого,**
- **указательного (в том числе указатели на члены класса),**
- **ссылочного;**

б) именами типов (перед именем типа надо указывать ключевое слово **class** или **typename**).

5. Параметры-объекты являются **константами**, их нельзя изменять внутри шаблона.

Шаблоны функций

template < список_параметров_шаблона >
тип_рез-та имя_функции (список_аргументов_функции) { /*...*/ }

Обращение к функции-шаблону: имя_функции < список_фактич._пар._шаблона >
(список_фактич_аргументов_функции);

Пример:

```
template < class T > // функция суммирования элементов массива
T  sum ( T  array[ ],  int size ) {
        T  res = 0;
        for ( int i = 0; i < size; i++ )    res += array[ i ];
        return res;
}
```

Использование шаблона для массивов типа **int** [10] :

```
int  iarray [10];
int  i_sum;
//...
i_sum = sum < int > ( iarray, 10 );
```

Можно задать аргумент **size** в виде параметра шаблона:

```
template < class  T, int  size >
T  sum ( T  array [ ] ) { /* ... */ }
```

Тогда вызов sum будет таким:

```
i_sum = sum < int, 10 > ( iarray );
```

Неявное определение параметра-типа шаблона

Пример 1

```
class complex
{...  public:
            complex ( double r = 0, double i = 0 );
            operator double ();        .....
};
template < class T >
T  f ( T& x, T& y )  {
            return x > y ? x : y;
}
double f ( double x, double y ){
            return x > y ? -x : -y;
}
int main ( ) {
            complex a ( 2 , 5 ), b ( 2 , 7 ), c;
            double x = 3.5, y = 1.1;
            int  i,  j = 8,  k = 10;

            c = f ( a , b );        // f < complex > (a , b)
            x = f ( a , y );        // f (a , y)
            i = f ( j , k );        // f  < int > (j , k)
            return 0;
}
```

Пример 2

```
template < class T >
T max (T & x, T & y) {
    return x > y ? x : y;
}

int main ( ) {
    double x = 1.5,  y = 2.8,  z;
    int  i = 5, j = 12,  k;
    char * s1 = "abft";
    char * s2 = "abxde", * s3;

    z = max ( x, y );                  // max <double>
    k = max < int > (i, j);            // max <int>
    //z = max (x, i);                  // Err! -  неоднозначный выбор параметров
    z = max < double > ( y, j );
    s3 = max  (s2, s1);                // max < char * >,
                                       // но происходит сравнение адресов
    return 0;
}
```

Пример 3

```
template <class T>  T  m1 (T a, T b) {
   cout << "m1_1\n";
   return a < b ? b : a;
}

template <class T, class B> T m1 (T a, T b, B c) {
  cout << "m1_2\n";
  c = 0;  return a < b ? b : a;
}
template <class T, class Z> T m1 (T a, Z b) {
  cout << "m1_3\n";
  return a < b ? b : a;
}
```

```
int m1 (int a, int b) {
        cout << "m1_4\n";
        return a < b ? b : a;
}

int m1 (int a, double b) {
        cout << "m1_5\n";
        return a;
}
```

```
int main () {
  int i;
  m1 <int> (2, 3);                  // m1_1
  m1 <int, int> (2, 3);             // m1_3
  m1 <int> (2, 3, i);               // m1_2
  m1 (1, 1);                        // m1_4
  m1 (1.3, 1);                      // m1_3
  m1 (1.3, 1.3);                    // m1_1
  return 0;
}
```

```
// Если убрать первый шаблон:

// m1_3
// m1_3
// m1_2
// m1_4
// m1_3
// m1_3
```

Алгоритм выбора оптимально отождествляемой функции с учетом шаблонов

- Для каждого шаблона, подходящего по набору формальных параметров, осуществляется формирование специализации, соответствующей списку фактических параметров.

- Если есть два шаблона функции и один из них более специализирован (т.е. каждый его допустимый набор фактических параметров также соответствует и второй специализации), то далее рассматривается только он.

- Осуществляется поиск оптимально отождествляемой функции из полученного набора функций, включая определения обычных функций, подходящие по количеству параметров. При этом если параметры некоторого шаблона функции были определены путем **выведения** по типам фактических параметров вызова функции, то при дальнейшем поиске оптимально отождествляемой функции к параметрам данной специализации шаблона нельзя применять никаких описанных выше преобразований, кроме преобразований **Точного** отождествления.

- Если обычная функция и специализация подходят одинаково хорошо, то выбирается обычная функция.

- Если полученное множество подходящих вариантов состоит из одной функции, то вызов разрешим. Если множество пусто или содержит более одной функции, то генерируется сообщение об ошибке.

Шаблоны классов

Шаблоны создаются для классов, имеющих общую логику работы.

Для определения шаблона класса перед ключевым словом **class** помещается **template**-квалификатор.

template <список_параметров_шаблона_типа> **class** имя_класса { /*...*/ };

Конкретный экземпляр шаблона класса (объект класса) можно создать так:

имя_класса <список фактич_парам> объект;

Для шаблонов класса никакие фактические параметры по умолчанию не выводятся.

Функции-члены класса-шаблона автоматически становятся функциями-шаблонами.

Шаблоны методов.

Можно описывать шаблонные методы в классах, не являющихся шаблонами.

Запрещено определять шаблоны для виртуальных методов, из-за возникающих больших накладных расходов на возможную перестройку таблиц виртуальных методов при компиляции.

Шаблонный класс stack

```
template <class T, int max_size >
class stack {
        T s [max_size];
        int top;
public:
        stack ( ) { top = 0;}
        void reset ( ) { top = 0;}
        void push (T i);
        T pop () ;
        bool is_empty ( ) { return top == 0;}
        bool is_full  () { return top == max_size;}
};
```

```
template <class T, int max_size >
void stack <T, max_size > :: push (T i) {
        if ( ! is_full () ) {
                s [top] = i;
                top ++;
        }
        else
                throw "stack_is_full";
}
```

```
template <class T, int max_size >
T stack <T, max_size > :: pop ( ) {
        if ( ! is_empty () ) {
                top --;
                return s [top];
        }
        else
              throw "stack_is_empty";
}
```

Виды отношений между классами

Часто при проектировании программ, разрабатываемых в объектно-ориентированном стиле, взаимосвязь используемых в них классов и объектов представляют в виде диаграмм UML (Unified Modeling Language)

Классы изображают в виде прямоугольника, состоящего из трех частей:

сверху – имя класса,

в середине – члены данные, возможно, с указанием типов,

внизу – прототипы методов класса.

Имена абстрактных классов и чистых виртуальных функций выделяются курсивом.

Перед описанием имени члена класса или метода можно указать спецификатор доступа с помощью значков + (**public**),
- (**private**) или
(**protected**).

Для статических членов класса после спецификатора доступа указывается символ $.

Большинство ООЯП поддерживают следующие отношения между классами:

- Ассоциация
- Наследование
- Агрегация
- Использование
- Инстанцирование

Ассоциация

Ассоциация – отношение, показывающее, что два класса концептуально взаимодействуют друг с другом.

Отношение ассоциации удобно представлять в виде ER-диаграмм (entity - relationships – сущность - связь), в основном используемым при разработке реляционных баз данных. Связи изображаются сплошными линиями без направления.

Виды связей, представляемых ER-диаграммами:

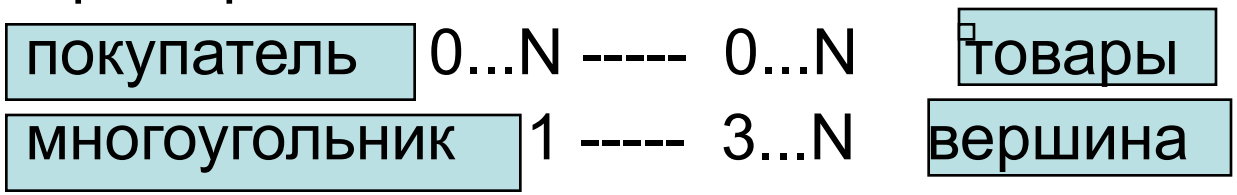
1—1

1—N

N—N

Различают обязательное и необязательное участие сущностей в установленных между ними связях.

Примеры:

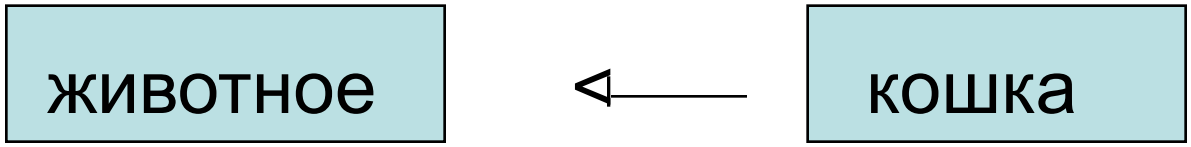


Наследование

Часть – общее (“ is a ”)

Отношение задается в виде стрелки с незакрашенным треугольником на конце, которая указывает на базовый класс.

Пример:



базовый класс
суперкласс

производный класс
подкласс

Агрегация

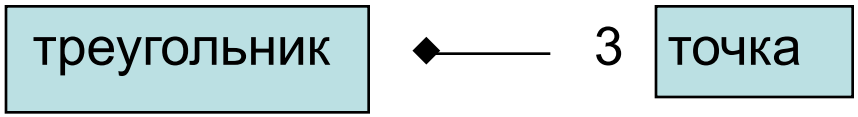
Часть – целое (“ has a ”).

Строгая агрегация – **композиция.**

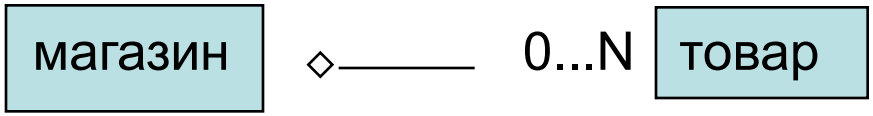
Нестрогая агрегация - **агрегация** (при этом один объект может быть включен в разные объекты одновременно).

Композиция обозначается стрелкой с закрашенным ромбом на конце, направленной на включающий класс, а агрегация – стрелкой с незакрашенным ромбом на конце.

Примеры:




```
class triangle {...
        point p1,p2,p3; ...
}
```




```
class shop {...
        goods * g; ...
}
```

Использование и Инстанцирование

Отношение **использования** возникает, когда

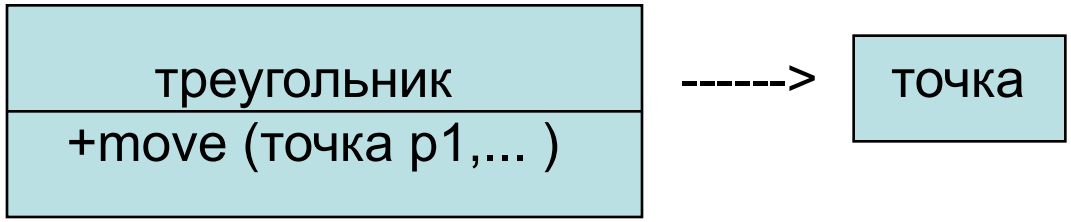
- в прототипе метода одного класса используется имя другого класса;
- в теле метода одного класса – локальный объект другого класса;
- в теле метода одного класса вызывается функция другого класса.

Использующий класс называют client,

а используемый supplier.

Отношение использования обозначается пунктирной стрелкой указывающей на класс supplier.

Пример:



Инстанцирование – связь между шаблоном класса и классом - результатом генерации по шаблону.

В UML инстанцирование обозначается стрелкой, идущей от шаблона класса к конкретной его реализации.

Стандартная библиотека шаблонов STL

STL (Standard Template Library) является частью стандарта C++.

Ядро STL состоит из четырех основных компонентов:

контейнеры,

итераторы,

алгоритмы,

распределители памяти.

Контейнеры

Контейнер – это класс, который предназначен для хранения объектов какого-либо типа.

Примеры известных ранее контейнеров:

- таблица идентификаторов,

- массив,

- дерево,

- список,

- ассоциативный список, например, список, хранящий фамилии людей и номера их телефонов, ключом которого является фамилия, если она уникальна(!).

Стандартные контейнеры STL

vector < T > - динамический массив
list < T > - линейный список
stack < T > - стек
queue < T > - очередь
deque < T > - двусторонняя очередь
priority_queue < T > - очередь с приоритетами
set < T > - множество
bitset < N > - множество битов (массив из N бит)
multiset < T > - набор элементов, возможно, одинаковых
map < key, val > - ассоциативный список
multimap < key, val > - ассоциативный список для хранения пар ключ/значение, где с каждым ключом может быть связано более одного значения.

Состав контейнеров

В каждом классе-контейнере определен набор методов для работы с контейнером, причем все контейнеры поддерживают **стандартный набор базовых операций.**

Базовая операция контейнера (базовый метод) – метод класса, имеющий во всех контейнерах одинаковое имя, одинаковый прототип и семантику (их примерно 15-20).

Например,

функция **push_back ()** помещает элемент в конец контейнера,

функция **size ()** выдает текущий размер контейнера.

Базовыми методами можно пользоваться одинаково независимо от того, в каком конкретно контейнере находятся элементы, можно также менять контейнеры, не меняя тела функций, работающих с ними.

Операции, которые не могут быть эффективно реализованы для всех контейнеров, не включаются в набор общих операций.

Например, обращение по индексу введено для контейнера **vector**, но не для **list.**

Типы, используемые в контейнерах

Каждый контейнер в своей **public** части содержит серию **typedef,** где введены стандартные имена типов, например:

value_type	- тип элемента,
allocator_type	- тип распределителя памяти,
size_type	- тип, используемый для индексации,
iterator	- итератор,
const_iterator	- константный итератор,
reverse_iterator	- обратный итератор**,**
const_reverse_iterator	- обратный константный итератор,
pointer	- указатель на элемент**,**
const_pointer	- указатель на константный элемент,
reference	- ссылка на элемент**,**
const_reference	- ссылка на константный элемент.

Эти имена определяются внутри каждого контейнера так, как это необходимо, т.е. скрывают реальные типы, что, например, позволяет писать программы с использованием контейнеров, ничего не зная о реальных типах.

В частности, можно составить код, который будет работать с любым контейнером.

Распределители памяти

Каждый контейнер имеет **распределитель памяти** (**allocator**), который используется при выделении памяти под элементы контейнера и предназначен для того, чтобы освободить пользователей контейнеров, от подробностей физической организации памяти.

Стандартная библиотека обеспечивает стандартный распределитель памяти, заданный стандартным шаблоном ***allocator*** из заголовочного файла **<memory>**, который выделяет память при помощи операции **new ( )** и по умолчанию используется всеми стандартными контейнерами.

Класс **allocator** обеспечивает стандартные способы выделения и перераспределения памяти, а также стандартные имена типов для указателей и ссылок.

Пользователь может задать свои распределители памяти, предоставляющие альтернативный доступ к памяти.

Стандартные контейнеры и алгоритмы получают память и обращаются к ней через средства, обеспечиваемые распределителем памяти.

Класс allocator

```
template <class T> class allocator {
public:
        typedef  T * pointer;
        typedef  T & reference;
        ……
        allocator ( ) throw ( );
        ……
        pointer allocate ( size_type  n );      // выделяет память для
                                                // n объектов типа T
        void deallocate ( pointer p,  size_type n );  // освобождает память,
                                                // отведенную под n объектов типа T
        void construct ( pointer  p,  const T & val );
                                                // инициализирует *p значением val
        void destroy ( pointer  p );            // вызывает деструктор для *p,
                                                // память при этом не освобождается.

        ……
};
```

Итераторы

Итератор – это класс, объекты которого выполняют такую же роль по отношению к контейнеру, как указатели по отношению к массиву. Указатель может использоваться в качестве средства доступа к элементам массива, а итератор - в качестве средства доступа к элементам контейнера.

Итераторы «склеивают» ядро STL в одну библиотеку.

Итераторы поддерживают абстрактную модель данных как последовательности объектов.

Понятия «нулевой итератор» не существует, а при организации циклов происходит сравнение с концом последовательности.

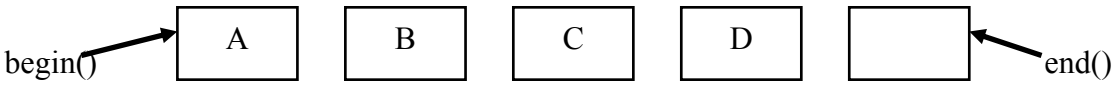
Каждый контейнер обеспечивает свои итераторы, поддерживающие стандартный набор итерационных операций со стандартными именами и смыслом.

Итераторные классы и функции находятся в заголовочном файле **< iterator >.**

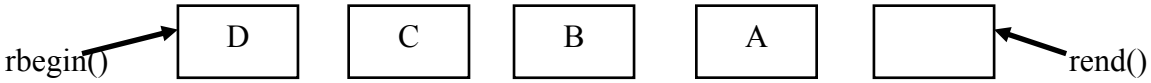
Методы контейнеров для нахождения значений итераторов концов последовательности элементов

- **iterator begin ( ); –** возвращает итератор, который указывает на первый элемент последовательности.
- **const_iterator begin ( ) const;**
- **iterator end ( ); –** возвращает итератор, который указывает на элемент, следующий за последним элементом последовательности.
- **const_iterator end ( ) const;**
- **reverse_iterator rbegin ( ); -** возвращает итератор, указывающий на первый элемент в обратной последовательности.
- **const_reverse_iterator rbegin ( ) const;**
- **reverse_iterator rend ( ); -** возвращает итератор, указывающий на элемент, следующий за последним в обратной последовательности.
- **const_reverse_iterator rend ( ) const;**

Прямые итераторы



Обратные итераторы



Операции над итераторами

Пусть p - объект типа итератор.

К каждому итератору можно применить, как минимум, три ключевые операции:

- *p – элемент, на который указывает итератор,
- p++ - переход к следующему элементу последовательности,
- == - операция сравнения.

Пример:

iterator p = v.begin(); – такое присваивание верно независимо от того, какой контейнер v .

Теперь *p – первый элемент контейнера v.

Замечание:

при проходе последовательности как прямым, так и обратным итератором переход к следующему элементу будет p++ (а не p-- !).

Не все виды итераторов поддерживают один и тот же набор операций.

Категории итераторов

В библиотеке STL введено **5 категорий итераторов:**

- 1. **Вывода** (output - запись в контейнер)
  (*p = , ++)

- 2. **Ввода** (input - считывание из контейнера)
  (= *p, →, ++, ==, !=)

- 3. **Однонаправленный** (forward)
  (*p =, =*p, →, ++ , ==, !=)

- 4. **Двунаправленный** (bidirectional)
  (*p=, =*p, →, ++,--, ==, !=) - list, map, set

- 5. **С произвольным доступом** (random_access)
  (*p=, =*p, →, ++,--, ==, !=, [ ], +, -, +=, -=, <, >, <=, >=) - vector,deque

Каждая последующая категория является более мощной, чем предыдущая.

Алгоритмы

Алгоритмы STL (их всего 60) - реализуют некоторые распространенные операции с контейнерами, которые не представлены функциями-членами каждого из контейнеров (например, просмотр, сортировка, поиск, удаление элементов…).

Каждый алгоритм выражается шаблоном функции или набором шаблонов функций.

Операции, реализуемые алгоритмами, являются универсальными для любого из контейнеров и поэтому определены вне этих контейнеров.

Реализация алгоритмов не использует имен никаких конкретных контейнеров, а все действия над контейнером производятся через универсальные имена итераторов.

Зная, как устроены алгоритмы, можно писать свои собственные алгоритмы обработки, которые не будут зависеть от контейнера.

Все стандартные алгоритмы находятся в пространстве имен std, а их объявления - в заголовочном файле **< algorithm >** .

Группы алгоритмов

1. **Немодифицирующие алгоритмы -** извлекают информацию из контейнера, но не модифицируют сам контейнер (ни элементы, ни порядок их расположения).

Примеры :

Find () – находит первое вхождение элемента с заданным значением

Count () – количество вхождений элемента с заданным значением

For_each () – применяется некоторая операция к каждому элементу (не связано с изменением)

2. Модифицирующие алгоритмы - изменяют содержимое контейнера. Либо сами элементы меняются, либо их порядок, либо их количество.

Примеры :

Transform () – применяется некоторая операция к каждому элементу (каждый элемент изменяется)

Reverse () – переставляет элементы в последовательности

Copy () – создает новый контейнер

3. Сортировка.

Примеры :

Sort () – простая сортировка .

Stable_sort () – сохраняет порядок следования одинаковых элементов (например, это бывает существенно при сортировке по нескольким ключам).

Merge () – склеивает две отсортированные последовательности.

Категории итераторов и алгоритмы

По соглашению, при описании алгоритмов, входящих в STL, используются стандартные имена формальных параметров-итераторов.

В зависимости от названия итератора в прототипе алгоритма, должен использоваться итератор уровня «не ниже чем». То есть по названию параметров шаблона можно понять, какого рода итератор нам нужен, то есть к какому контейнеру применим этот алгоритм.

Пример: шаблонная функция **find() с тремя параметрами (** итератор, с какого начинается поиск, каким заканчивается и искомый элемент).

Для реализации целей функции достаточно итератора ввода (из контейнера).

```
Template < class  InputInterator,  class T >
InputIterator find ( InputIterator first, InputIterator last, const T& value ) {
                while ( first != last  &&  * first  !=  value )
                            first ++;
                return first;
}
```

Однако категория итераторов не принимает участия в вычислениях. Этот механизм относится исключительно к компиляции.

typename

1. Ключевое слово **typename** используется при описании параметра-типа шаблона (наряду с ключевым словом **class**). Например,

```
template <typename T>
void f (T a) {...}
```

2. Если используемое в шаблоне имя типа зависит от параметров шаблона, **необходимо** использовать ключевое слово **typename**. Например,

```
template <class T>
void f (vector <T> & v)
{
    vector <T> :: iterator i = begin ();                    // Err!
    typename vector <T> :: iterator i = begin();   //O.K.
    ...
}
```

Пример шаблонной функции, использующей тип, вложенный в класс–параметр шаблона

```
struct X {
        enum { e1, e2, e3 } g;
        struct  inner {
                int i, j;
                void g ( ) { cout << “ggg\n”; }
        };
        inner c;
};
        template < class  T >
        void   f ( typename  T::inner   t )  {  t.g ( );  }

        int main ( ) {
                X  x;
                x.g = X::e1;
                x.c.g ( );
                X::inner   iii;
                iii.i = 7;
                f <X> ( iii );
                return 0;          }
```

Пример шаблонной функции для контейнеров STL

Поиск заданного элемента в контейнере, начиная с последнего (просмотр контейнера от конца к началу) обычно производится так:

```
template < class  C >
typename C :: const_iterator   find_last
      ( const  C & c, typename  C :: value_type  v )
{
            typename C :: const_iterator p = c.end ( );
            while (p != c.begin ( ) )
                  if ( * -- p == v )
                        return p;
            return  c.end ( );
}
```

Контейнер vector

```
template < class T , class A = allocator < T > > class vector {
.......
public:
// Типы – typedef …... - см. выше
// Итераторы …….. - см. выше
//
// Доступ к элементам
//
reference operator [ ] (size_type n);  // доступ без проверки диапазона
const_reference operator [ ] (size_type n) const;

reference at (size_type n); // доступ с проверкой диапазона (если индекс
// выходит за пределы диапазона, возбуждается исключение out_of_range)
const_reference at (size_type n) const;

reference front ( );  // первый элемент вектора
const_reference front ( ) const;

reference back ();  // последний элемент вектора
const_reference back ( ) const;
```

Контейнер vector

```
// Конструкторы, деструктор, operator=
//
explicit vector (const A&=A()); //создается вектор нулевой длины

explicit vector (size_type n; const T& value = T(); const A& = A());
// создается вектор из n элементов со значением value
// (или с "нулями" типа T, если второй параметр отсутствует;
// в этом случае конструктор умолчания в классе T обязателен)

template <class I> vector (I first, I last, const A& = A());
// инициализация вектора копированием элементов из [first, last),
// I - итератор для чтения

vector (const vector  < T, A > &  obj ); // конструктор копирования

vector& operator = (const vector  < T, A > & obj );

~vector();
```

Контейнер vector

```
//Некоторые функции-члены класса vector
//
iterator erase (iterator i ); // удаляет элемент, на который указывает данный
// итератор. Возвращает итератор элемента, следующего за удаленным.

iterator erase (iterator st, iterator fin); // удалению подлежат все элементы
// между st и fin, но fin не удаляется. Возвращает fin.

Iterator insert ( iterator i , const T& value = T()); // вставка некоторого
// значения value перед i. Возвращает итератор вставленного элемента).

void insert (iterator i , size_type n, const T&value); // вставка n копий
// элементов со значением value перед i.

void push_back ( const T&value ) ; // добавляет элемент в конец вектора

void pop_back () ; // удаляет последний элемент (не возвращает значение!)

size_type  size() const; // выдает количество элементов вектора

bool empty () const; // возвращает истину, если вызывающий вектор пуст

void clear(); //удаляет все элементы вектора          ....        }
```

Контейнер list

```
template < class T , class A = allocator < T > > class list {
.....
public:
// Типы
//      ……..
// Итераторы
//      ……..
//
// Доступ к элементам
//
reference front ();  // первый элемент списка

const_reference front () const;

reference back ();  // последний элемент списка

const_reference back () const;
```

Контейнер list

```
// Конструкторы, деструктор, operator=
//
explicit list (const A& = A());  // создается список нулевой длины

explicit list (size_type n; const T& value = T(); const A& = A());
// создается список из n элементов со значением value
// (или с "нулями" типа T, если второй параметр отсутствует

template <class I> list (I first, I last, const A& = A());
// инициализация списка копированием элементов из [first, last),
// I - итератор для чтения

list (const list  < T, A > & obj ); // конструктор копирования

list& operator = (const list  < T, A > & obj );

~list();
```

Контейнер list

```
//Некоторые функции-члены класса list
//
iterator erase (iterator i ); // удаляет элемент, на который указывает данный
// итератор. Возвращает итератор элемента, следующего за удаленным.

iterator erase (iterator st, iterator fin); // удалению подлежат все элементы
// между st и fin, но fin не удаляется. Возвращает fin.

Iterator insert ( iterator i , const T& value = T()); // вставка некоторого
// значения value перед i. Возвращает итератор вставленного элемента).

void insert (iterator i , size_type n, const T&value); // вставка n копий
// элементов со значением value перед i.

void push_back ( const T&value ) ; // добавляет элемент в конец списка
void push_front ( const T&value ) ; // добавляет элемент в начало списка
void pop_back ( ) ; // удаляет последний элемент (не возвращает значение!)
void pop_front ( ) ; // удаляет первый элемент списка
size_type  size( ) const; // выдает количество элементов списка
bool empty ( ) const; // возвращает истину, если вызывающий список пуст
void clear( );  // удаляет все элементы списка
…….
}
```

Пример использования STL

Функция, формирующая по заданному вектору целых чисел список из элементов вектора с четными значениями и распечатывающая его.

```
# include < iostream >
# include < vector >
# include < list >
using namespace std;

void g (vector <int> &   v,  list <int> &  lst) {
    int i;
    for (i = 0; i < v.size( ); i++)
          if ( ! ( v[ i ] % 2 ) )
                    lst.push_back( v[ i ] );
    list < int > :: const_iterator  p = lst.begin ();
    while ( p != lst.end ( ) ) {
          cout << *p << endl;
          p++;
    }
}
```

Достоинства STL - подхода

- Каждый контейнер обеспечивает стандартный интерфейс в виде набора операций, так что один контейнер может использоваться вместо другого, причем это не влечет существенного изменения кода.
- Дополнительная общность использования обеспечивается через стандартные итераторы.
- Каждый контейнер связан с распределителем памяти - аллокатором, который можно переопределить с тем, чтобы реализовать собственный механизм распределения памяти.
- Для каждого контейнера можно определить дополнительные итераторы и интерфейсы, что позволит оптимальным образом настроить его для решения конкретной задачи.
- Контейнеры по определению однородны, т.е. должны содержать элементы одного типа, но возможно создание разнородных контейнеров как контейнеров указателей на общий базовый класс.
- Алгоритмы, входящие в состав STL, предназначены для работы с содержимым контейнеров. Все алгоритмы представляют собой шаблонные функции, следовательно, их можно использовать для работы с любым контейнером.

Введение в C++11 (стандарт ISO/IEC 14882:2011)

Вне рассмотрения в рамках курса остаются нововведения для работы с шаблонами:

- ведение понятий лямбда-функций и выражений,
- внешние шаблоны,
- альтернативный синтаксис шаблонных функций,
- расширение возможностей использования угловых скобок в шаблонах,
- typedef для шаблонов,
- шаблоны с переменным числом аргументов,
- статическая диагностика,
- изменения в STL,
- регулярные выражения.

Не рассматриваются также новые понятия

- ✓ тривиального класса,
- ✓ класса с простым размещением,
- ✓ ***explicit*** перед функциями преобразования,
- ✓ новшества в ограничениях для ***union***,
- ✓ новые строковые литералы,
- ✓ новые символьные типы ***char16_t*** и ***char32_t*** для хранения UTF-16 и UTF-32 символов,
- ✓ некоторое другое...

Введение в C++11
(стандарт ISO/IEC 14882:2011)

Полностью новый стандарт поддерживают компиляторы g++ начиная с версии 4.7. …

Для компиляции программы в соответствии с новым стандартом в командной строке в качестве опции компилятору g++ надо указать:

-std=c++0x (или в новых версиях -std=c++11) :

g++ …… -std=c++0x
(или g++ …… -std=c++11)

rvalue- ссылки

В C++11 появился новый тип данных – **rvalue-ссылка**:

<тип> && <имя> = <временный объект>;

В C++11 можно использовать перегруженные функции для неконстантных временных объектов, обозначаемых посредством rvalue-ссылок.

Например: **class** A; … A a; …

```
void f (A & x);                    ~ f (a);
void f (A && y);                   ~ f ( A() );
…
A && rr1 = A();
// A && rr2 = a; // Err!
int && n = 1+2;
n++;
…
```

Семантика переноса (Move semantics).

При создании/уничтожении временных объектов **неплоских** классов, как правило, требуется выделение-освобождение динамической памяти, что может отнимать много времени.

Однако, можно оптимизировать работу с временными объектами неплоских классов, если не освобождать их динамическую память, а просто перенаправить указатель на нее в объект, который **копирует значение временного объекта неплоского класса** (посредством поверхностного копирования). При этом после копирования надо обнулить соответствующие указатели у временного объекта, чтобы его конструктор ее не зачистил.

Это возможно сделать с помощью перегруженных конструктора копирования и операции присваивания с параметрами – **rvalue-ссылками**. Их называют конструктором переноса (move constructor) и операцией переноса (move assignment operator).

При этом компилятор сам выбирает нужный метод класса, если его параметром является временный объект.

Семантика переноса (Move semantics).

Пример:

```
class Str {
        char * s;
        int len;
public:
        Str (const char * sss = NULL); // обычный конструктор неплоского класса
        Str (const Str &); // традиционный конструктор копирования
        Str (Str && x) {  // move constructor
                s = x.s;
                x.s = NULL; // !!!
                len = x.len;
         }
        Str & operator = (const Str & x); // обычная перегруженная операция =
        Str & operator = (Str && x) { // move assignment operator
                s = x.s;
                x.s = NULL; // !!!
                len = x.len;
                return *this;
         }
        ~Str(); // традиционный деструктор неплоского класса
        Str operator + ( Str  x);      ...
};
```

Семантика переноса (Move semantics).

Использование rvalue-ссылок в описании методов класса Str приведет к более эффективной работе, например, следующих фрагментов программы:

```
... Str a("abc"), b("def"), c;
    c = b+a; //  Str& operator= (Str &&);
...

Str f (Str a ) {
        Str b;  ... return a;
}
...  Str d = f (Str ("dd") ); ... //  Str (Str &&);
```

Обобщенные константные выражения .

Введено ключевое слово

constexpr,

которое указывает компилятору, что обозначаемое им выражение является константным, что в свою очередь позволяет компилятору вычислить его еще на этапе компиляции и использовать как константу.

Пример:

```
constexpr int give5 () {
        return 5;
}
int mas [give5 () + 7];        // создание массива из 12
                               // элемнтов, так можно в C++11.
```

Обобщенные константные выражения .

Однако, использование **constexpr** накладывает жесткие ограничения на функцию:

- она не может быть типа **void**;
- тело функции должно быть вида **return** *выражение*;
- *выражение* должно быть константой,
- функция, специфицированная **constexpr** не может вызываться до ее определения.

В константных выражениях можно использовать не только переменные целого типа, но и переменные других числовых типов, перед определением которых стоит **constexpr.**

Пример:

```
constexpr double a = 9.8;
constexpr double b = a/6;
```

Вывод типов.

Описание ***явно инициализируемой*** переменной может содержать ключевое слово **auto:** при этом типом созданной переменной будет тип инициализирующего выражения.

Пример:

Пусть ft(....) – шаблонная функция, которая возвращает значение шаблонного типа, тогда при описании

auto var1 = ft(....);

переменная var1 будет иметь соответствующий шаблонный тип.

Возможно также:

auto var2 = 5; // var2 имеет тип **int**

Вывод типов

Для определения типа выражения во время компиляции при описании переменных можно использовать ключевое слово **decltype**.

Пример:

```
int v1;
decltype (v1) v2 = 5; // тип переменной v2 такой же, как у v1.
```

Вывод типов наиболее интересен при работе с шаблонами, а также для уменьшения избыточности кода.

Пример: Вместо

```
for(vector <int>::const_iterator itr = myvec.cbegin(); itr != myvec.cend(); ++itr)
  ...
```

можно написать:

```
for(auto itr = myvec.cbegin(); itr != myvec.cend(); ++itr) ...
```

For-цикл по коллекции

Введена новая форма цикла for, позволяющая автоматически осуществлять перебор элементов коллекции (массивы и любые другие коллекции, для которых определены функции begin () и end()).

Пример:

```
int arr[5] = {1, 2, 3, 4, 5};
for (int &x : arr) {
        x *= 2;
} ...
```

При этом каждый элемент массива увеличится вдвое.

```
for (int x : arr) {
        cout << x << ‘ ‘;
} ...
```

Улучшение конструкторов объектов

В отличие от старого стандарта новый стандарт C++11 позволяет вызывать одни конструкторы класса (так называемые делегирующие конструкторы) из других, что в целом позволяет избежать дублирования кода.

Пример:

```
class A {
        int n;
public:
        A (int x) : n (x) { }
        A ( ) : A (14) { }
};
```

Улучшение конструкторов объектов

Стало возможно инициализировать члены-данные класса в области их объявления в классе.

Пример:

```
class A {
        int n = 14;
public:
        explicit A (int x) : n (x) { }
                 A ( ) { }
        };
```

Любой конструктор класса A будет инициализировать n значением 14, если сам не присвоит n другое значение.

Замечание: Если до конца проработал хотя бы один делегирующий конструктор, его объект уже считается **полностью созданным**. Однако, объекты производного класса начнут конструироваться только после выполнения всех конструкторов (основного и его делегирующих) базовых классов.

Явное замещение виртуальных функций и финальность

В C++11 добавлена возможность (с помощью спецификатора ***override***) отследить ситуации, когда виртуальная функция в базовом классе и в производных классах имеет разные прототипы, например, в результате случайной ошибки (что приводит к тому, что механизм виртуальности для такой функции работать не будет).

Кроме того, введен спецификатор ***final*,** который обозначает следующее:

- *в описании классов* - то, что они не могут быть базовыми для новых классов,

- *в описании виртуальных функций* - то, что возможные производные классы от рассматриваемого не могут иметь виртуальные функции, которые бы замещали финальные функции.

Замечание: спецификаторы ***override u final*** имеют специальные значения только в приведенных ниже ситуациях, в остальных случаях они могут использоваться как обычные идентификаторы.

Явное замещение виртуальных функций и финальность

Пример:

```
struct B {
   virtual void  some_func ();
   virtual void  f (int);
   virtual void g () const;
};

struct D1 : public B {
   virtual void  sone_func ()  override; // Err: нет такой функции в B
   virtual void  f (int) override;            // OK!
   virtual void  f (long) override;       // Err: несоответствие типа параметра
   virtual void  f (int) const  override;
                                    // Err: несоответствие квалификации функции
   virtual int  f (int) override;     // Err: несоответствие типа результата
   virtual void g () const final;   // OK!
   virtual void  g (long);            // OK: новая виртуальная функция
};
```

Явное замещение виртуальных функций и финальность

Пример:

```
struct D2 : D1 { // см. предыдущий слайд

        virtual void g () const;    // Err: замещение финальной функции
};

struct F final {
   int x,y;
};

struct D : F {     // Err: наследование от  финального класса
   int z;
};
```

Константа нулевого указателя

В C++ NULL – это эквивалент константы 0, что может привести к нежелательному результату при перегрузке функций:

```
void f (char *);
void f (int);
```

При обращении f (NULL) будет вызвана f (**int**); , что, вероятно, не совпадает с планами программиста.

В C++11 введено новое ключевое слово ***nullptr*** для описания константы нулевого указателя:

```
std::nullptr_t  nullptr;
```

где тип *nullptr_t* можно неявно конвертировать в тип любого указателя и сравнивать с любым указателем.

Неявная конверсия в целочисленный тип **недопустима**, за исключением ***bool*** (в целях совместимости).

Для обратной совместимости константа 0 также может использоваться в качестве нулевого указателя.

Константа нулевого указателя

Пример:

```
char * pc = nullptr;     // OK!

int * pi = nullptr;      // OK!

bool   b = nullptr;      // OK: b = false;

int  i = nullptr;        // Err!

f (nullptr);       // вызывается  f(char*)  а не  f(int).
```

Перечисления со строгой типизацией

В C++ :

- ✓перечислимый тип данных фактически совпадает с целым типом,
- ✓если перечисления заданы в одной области видимости, то имена их констант не могут совпадать.

В C++11 наряду с обычным перечислением предложен также способ задания перечислений, позволяющий избежать указанных недостатков. Для этого надо использовать объявление ***enum class*** (или, как синоним, ***enum struct***). Например,

```
enum class  E { V1,  V2,  V3 = 100,  V4 /*101*/};
```

Элементы такого перечисления нельзя неявно преобразовать в целые числа (выражение E:: V4 == 101 приведет к ошибке компиляции).

Перечисления со строгой типизацией

В C++11 тип констант перечислимого типа не обязательно ***int*** (только по умолчанию), его можно задать явно следующим образом:

```
enum class E2 : unsigned int { V1, V2 };
```

// значение E2:: V1 определено, а V1 – не определено.

Или:

```
enum  E3 : unsigned long { V1 = 1, V2 };
```

// в целях обеспечения обратной совместимости определены и значение E3:: V1 , и V1.

В C++11 возможно предварительное объявление перечислений, но только если указан размер перечисления (явно или неявно):

```
enum E1;                    // Err: низлежащий тип не определен
enum E2 : unsigned int;     // OK!
enum  class E3 ;            //  OK: низлежащий тип  int
enum class E4 : unsigned long;      // OK!
enum E2 : unsigned short;   // Err: E2 ранее объявлен
                            // с другим низлежащим типом.
```

sizeof для членов данных классов без создания объектов

В C++11 разрешено применять операцию **sizeof** к членам-данным классов независимо от объектов классов.

Пример:

```
struct A {
        some_type a;
};
...  sizeof (A::a) ... // OK!
```

Кроме того, в C++11 узаконен тип ***long long int*** .

Системы программирования
Интегрированные среды разработки
Системы контроля версий

Структура вычислительной системы

Система программирования (СП) — это комплекс программных инструментов и библиотек, который поддерживает **весь** технологический цикл создания программного продукта (ПП).

ПП — программа, оформленная, документированная и специфицированная таким образом, что ее можно использовать независимо, отчужденно от автора программы.

Этапы технологического цикла создания ПП

I. **Создание ПП.**

II. **Сопровождение:**

- попытка приспособить ПП к измененным целям,
- исправление ошибок, не выявленных на этапе I.

III. **Эксплуатация ПП.**

Создание ПП (1)

1. Анализ требований

Уточняются, формализуются и документируются требования заказчика к ПП, в результате создаются ***внешняя спецификация ПП***, т.е. характеристика программы с точки зрения заказчика.

Часть требований к ПП можно формально записать с помощью языков спецификаций (SDL,…), таблиц решений, функциональных диаграмм.

2. Проектирование

Выделяются отдельные модули, определяется их иерархия и сопряжение между ними. В результате создается общая ***схема иерархии*** и ***внешняя спецификация отдельных модулей***.

По внешним спецификациям разрабатывается внутренняя структура каждого модуля - выбирается алгоритм работы модуля и способ внутреннего представления данных.

Создание ПП (2)

3. Кодирование.

4. Компоновка и интеграция

5. Тестирование и отладка

Верификация (ПП работает согласно спецификации)

Валидация (ПП пригоден для использования)

Тестирование: ручное, автоматизированное; функциональное, интеграционное, модульное; регрессионное

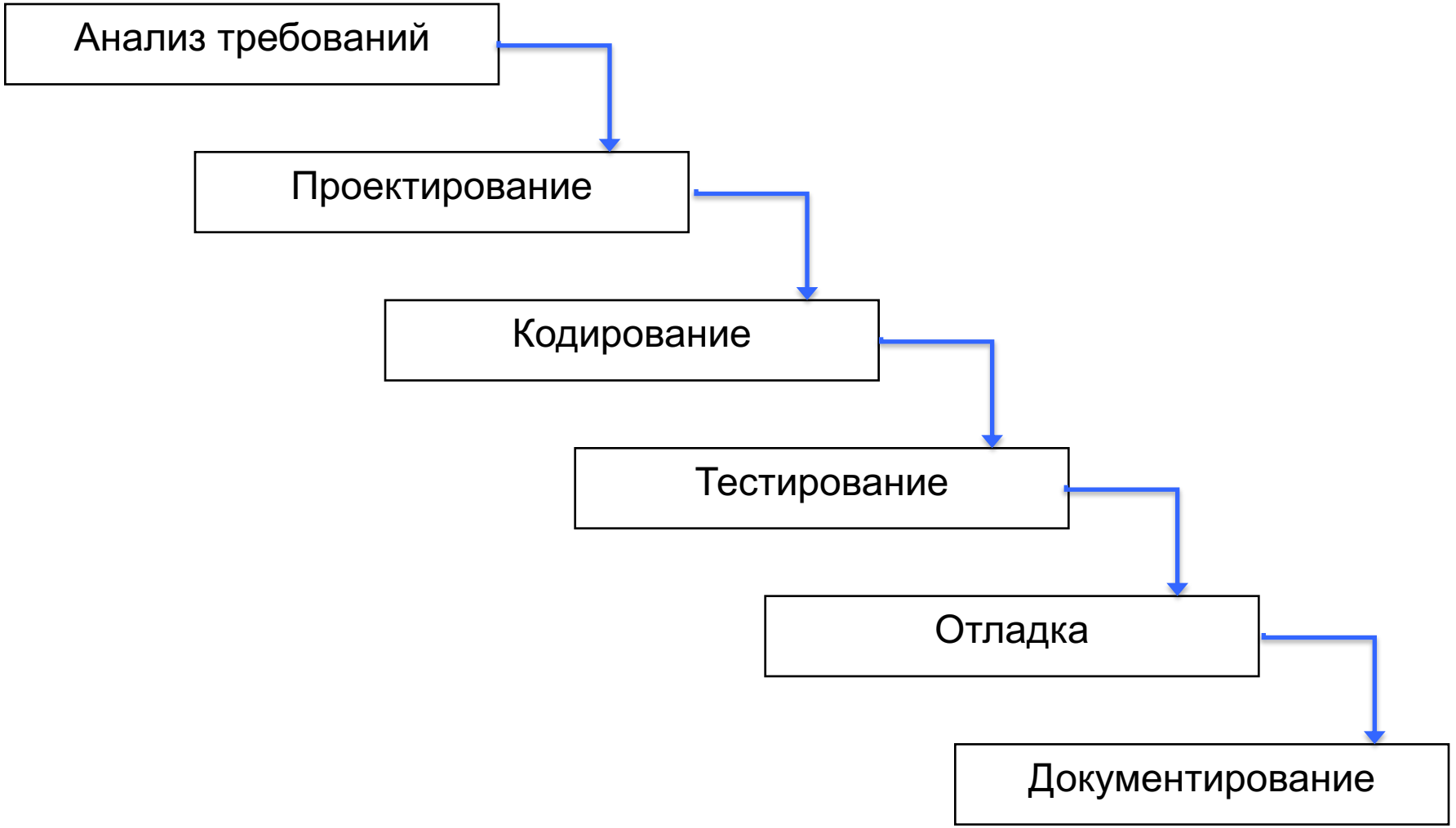
Формальное доказательство корректности работы

6. Документирование

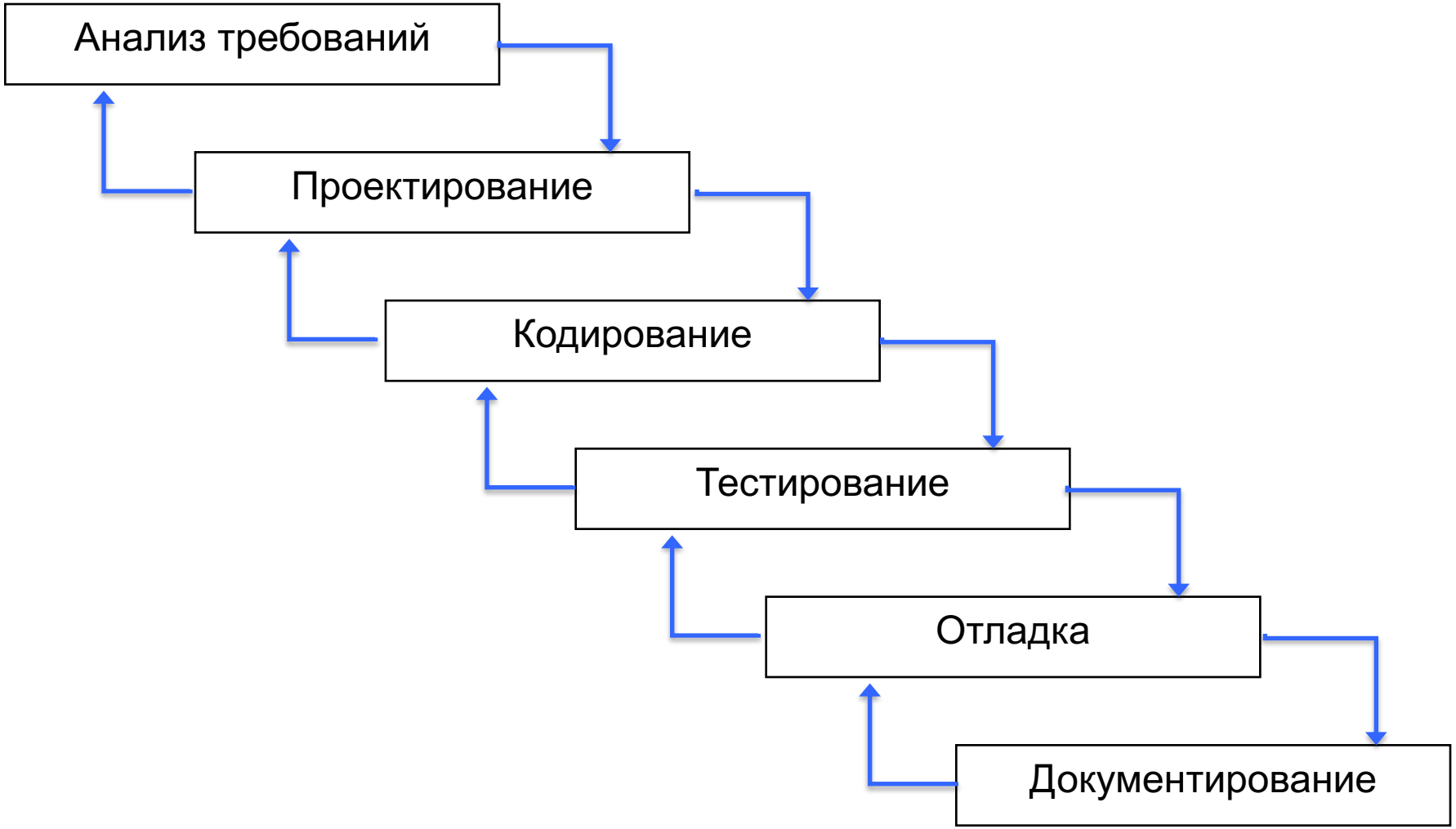
7. Внедрение

8. Сопровождение

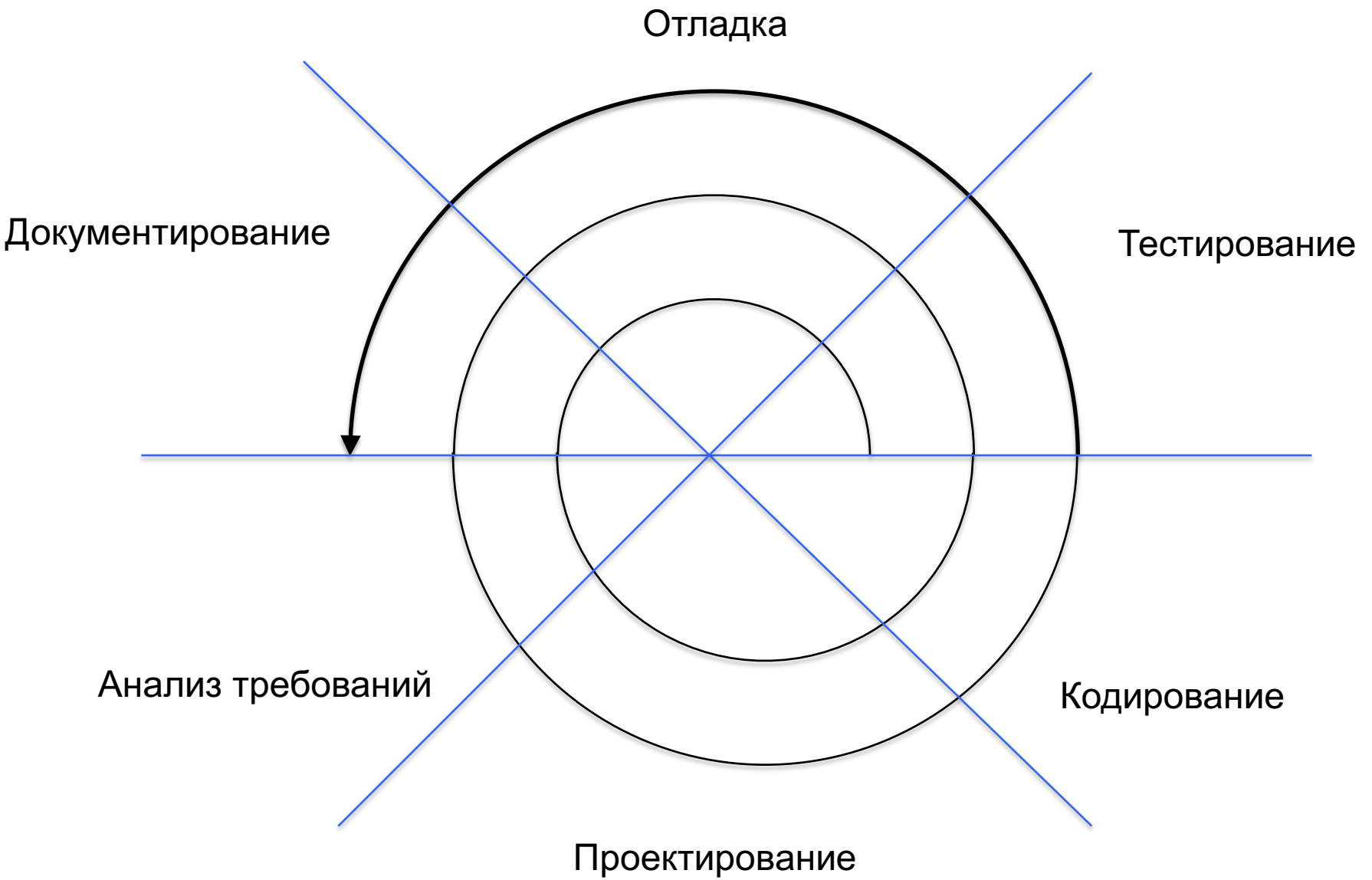
Каскадная модель



Каскадно-возвратная модель



Итерационная модель



Основные компоненты системы программирования.

1. **Транслятор** (переводит программы с языка программирования на машинный язык, что и позволяет выполнить их на ЭВМ).

2. **Макрогенератор** или макропроцессор (работает непосредственно перед транслятором, используется для получения макрорасширения исходной программы).

3. **Редактор текстов** (используется для составления программ на языке программирования).

4. **Редактор связей** или компоновщик (предназначен для связывания между собой (по внешним данным) объектных файлов, порождаемых компилятором, а также файлов библиотек, входящих в состав СП).

5. **Отладчик** (используется для проверочных запусков программ и исправления ошибок).

6. **Библиотеки стандартных программ** (облегчают работу программиста, используются на этапе трансляции и исполнения).

Дополнительные компоненты систем программирования

a) **Система контроля версий** для версионирования исходного текста ПП.

b) **Средства конфигурирования**

- помогают создавать **различные конфигурации** ПП в зависимости от конкретных параметров системного окружения, в котором ПП будет функционировать и от возможных различий отдельных версий ПП;
- поддерживают информацию обо всех предполагаемых и выполненных **изменениях ПП;**
- обеспечивают координированное **управление** развитием функциональности и улучшением характеристик системы.

c) **Средства тестирования** (помогают при составлении набора тестов).

d) **Профилировщик.** Профилирование — определение (в процентах) времени, затрачиваемого на выполнение отдельных фрагментов программы, как правило, для линейных участков кода (фрагментов программы, где нет передачи управления). Профилировщик часто используется для выявления мест для дальнейшей оптимизации программы.

e) **Справочная система** (содержит справочные материалы по языку программирования и компонентам СП).

Дополнительные компоненты систем программирования

f) Инструменты для статического анализа кода

- Производят анализ логики работы программы без её исполнения (работают с исходным текстом программы).
- Основное применение — поиск мест, где может содержаться логическая ошибка (lint и аналоги).
- Также используются для организации навигации по коду (генерация т. н. тэгов), полуавтоматического рефакторинга и проч.

g) Средства навигации по коду. В простейшем варианте (ctags) — анализ исходного текста, поиск в нем символов (определений функций, классов) и формирование указателя найденных символов для использования в текстовом редакторе. В более сложных случаях отыскиваются также отношения наследования, места использования символа в коде и проч.

h) Инструменты подготовки документации. Используются для автоматической генерации списков классов, функций и т. п. по исходному коду. При этом автоматически извлекаются комментарии к коду, и на выходе генерируется документация к коду, которая может компоноваться с концептуальной документацией на ПП и его подсистемы.

i) Управление разработкой. Планирование, отслеживание замечаний.

Виды систем программирования

(*По стратегии интеграции*)

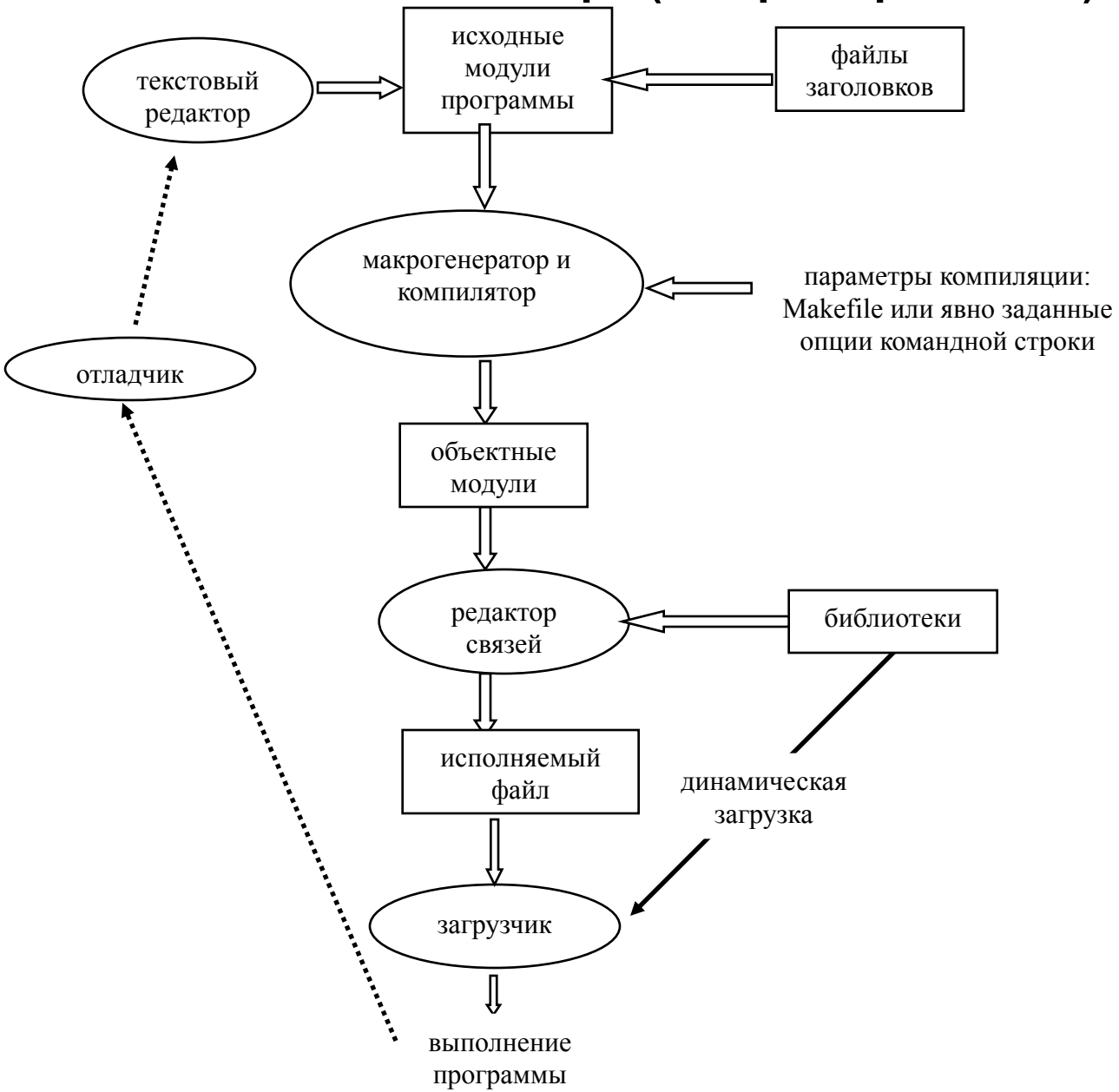
1. **Наборы независимых инструментов**

2. **Интегрированные системы программирования**

Стратегии трансляции

1. Компиляторы и ассемблеры
2. Интерпретаторы
3. Смешанная стратегия (байт-код, JIT-компиляция)

Общая схема функционирования основных компонентов СП на базе компилятора (на примере СП Си):



Общая схема функционирования основных компонентов СП на базе интерпретатора:

Интегрированная среда разработки

ИСР (IDE, integrated development environment) — комплекс программных средств, поддерживающих полный жизненный цикл ПП.

Простая ИСР содержит минимальный набор компонентов:

- текстовый редактор,
- компилятор,
- редактор связей,
- отладчик.

Состав продвинутой ИСР

- **модуль системы контроля версий** (все объекты, с которыми идет работа в рамках ИСР, хранятся в репозитории системы контроля версий);
- **графические средства** анализа и проектирования, обеспечивающие создание и редактирование иерархически связанных диаграмм, образующих модели ПП, а также графический пользовательский интерфейс, обеспечивающий взаимодействие с функциями API;
- **средства разработки приложений**, включая инструменты кодогенерации;
- **компилятор;**
- **текстовый редактор**;
- **средства статического анализа кода** с поддержкой навигации по коду и полуавтоматического рефакторинга;
- **средства документирования**;
- **средства тестирования и отладки**;
- **средства управления проектной деятельностью**, в т. ч. интеграция с системами отслеживания замечаний;
- **средства обратного конструирования** (позволяют восстановить логику и структуру программы по ее исходным текстам, с целью модификации или перенесения на другую платформу).

Текстовые редакторы

1. **Пакетные**
   + Макросредства

2. **Диалоговые**
   1) Строчные

   2) Экранные

Возможности текстового редактора (в ИСР)

1. Подготовка текста программы (обычные действия по созданию, редактированию, сохранению файла с текстом программы).

2. Многооконный интерфейс, управление окнами и вкладками.

3. Закладки, настраиваемые сочетания клавиш, шаблоны фрагментов текста, программное управление самим редактором

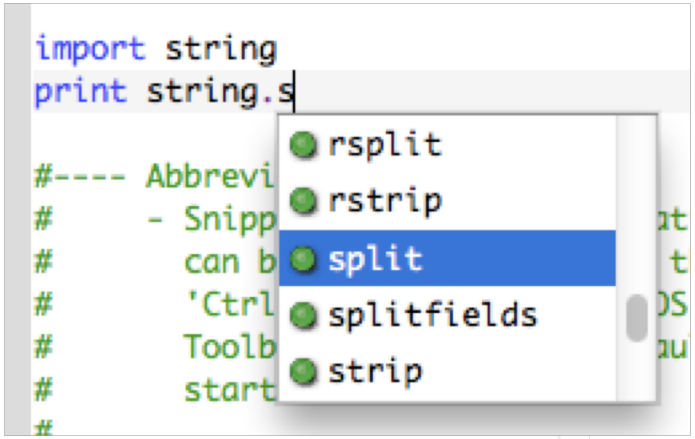
4. Интеграция с компилятором и средствами статического анализа кода.

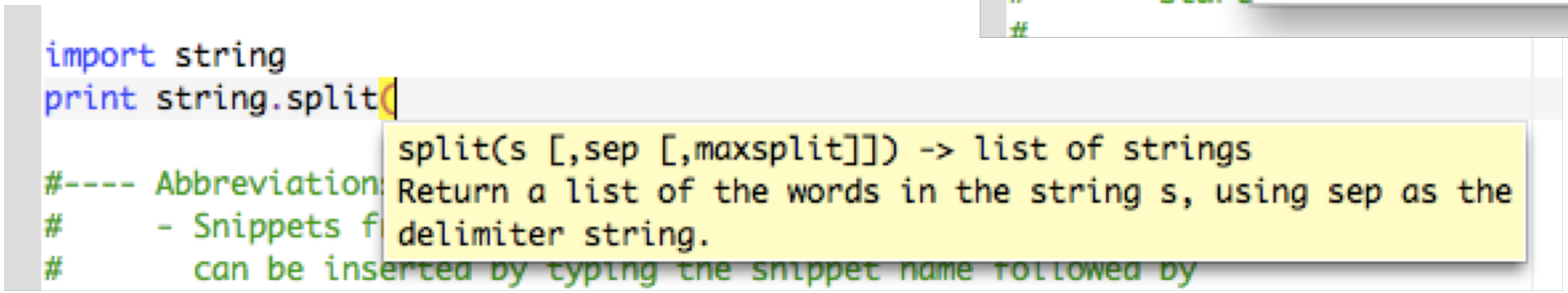
5. Интеграция с отладчиком.

Возможности текстового редактора (в ИСР)

Интеграция с компилятором и/или средствами статического анализа кода:

- визуализация текста с выделением лексем (синтаксическая подсветка элементов языка),
- дополнение кода, интерактивная подсказка,



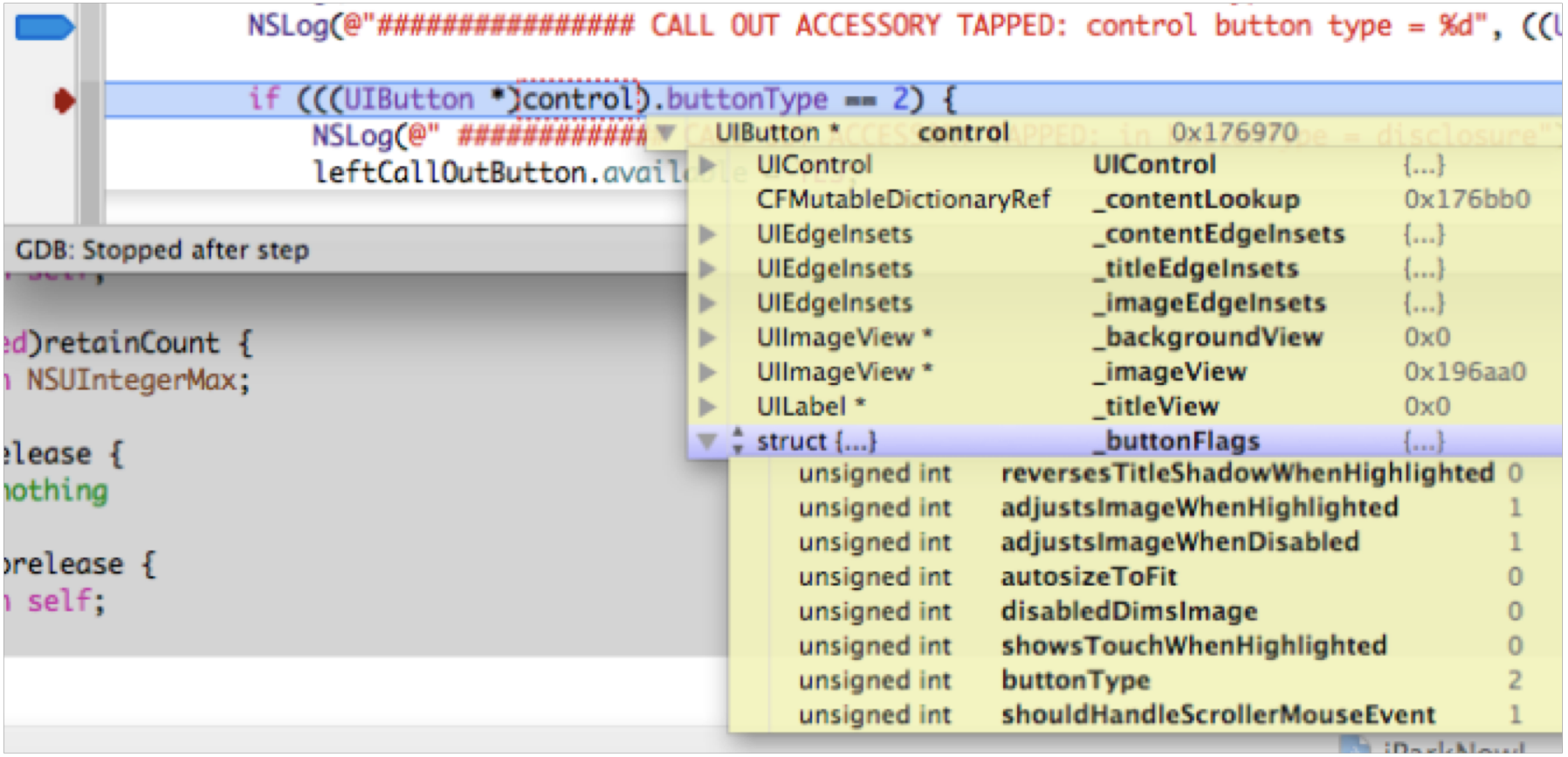


- всплывающие подсказки об атрибутах идентификаторов, если на них установить курсор, отображение ошибок, обнаруженных на этапе компиляции, в тексте программы,
- навигация по коду (переход к определению имени, поиск мест использования имени, поиск имени, навигация по иерархии наследования),
- рефакторинг кода (от простейших случаев: переименование идентификатора, генерация заглушки — до сложных: выделение базового класса, перенос метода вверх или вниз по иерархии).

Возможности текстового редактора (в ИСР)

4. Интеграция с отладчиком:

- отображение контрольных точек останова при отладке,
- отображение текущего значения объекта, при наведении курсора на идентификатор.



Задачи отладчика в рамках ИСР

1) пошаговое выполнение программы (шаг = строка; с трассировкой внутри вызываемой функции и без нее),

2) выполнение программы до строки, в которой в редакторе стоит курсор,

3) выделение выполняемой в данный момент строки,

4) приостановка выполнения программы, при этом:

- можно запросить значение переменной,
- можно заказать вычисление некоторого выражения,
- можно изменить значение переменной и продолжить выполнение программы (но не всякий отладчик позволяет изменять программный код, т.е. поддерживает частичную перекомпиляцию),

5) расстановка/снятие точек останова, которые визуализируются в текстовом редакторе,

6) выдача всей информации в терминах исходной программы.

Стратегии тестирования

Стратегия тестирования — это метод, используемый для отбора тестов, которые должны быть включены в тестовый комплект.

Стратегия является эффективной, если тесты, включенные в нее, с большой вероятностью обнаружат ошибки тестируемого объекта. Эффективность стратегии зависит от комбинации природы тестов и природы ошибок, на поиск которых эти тесты направлены.

•***Стратегия поведенческого теста*** основана на технических требованиях. Тестирование, выполняемое с помощью стратегии поведенческого теста, называется *поведенческим тестированием, функциональным тестированием* или *тестированием черного ящика*.

•***Стратегия структурного теста*** определяется структурой тестируемого объекта. Тестирование, выполненное с помощью стратегии структурного теста, называется также *тестированием белого ящика*. Стратегия структурного теста требует полного доступа к структуре объекта, то есть к исходной программе.

•***Стратегия гибридного теста*** является комбинацией поведенческой и структурной стратегий. Модули и низкоуровневые компоненты часто тестируются с помощью структурной стратегии. Большие компоненты и системы в основном тестируются с помощью поведенческой стратегии. Гибридная стратегия полезна на всех уровнях.

Способы тестирования

- Тестирование проводится не только на той стадии разработки программ, которая специально для этого предназначена, но и на предшествующих стадиях – при автономной отладке программ, еще до объединения их в единый программный комплекс. Такое тестирование называется ***модульным***. Его обычно проводят сами разработчики, которые проверяют точное соответствие программы выданной им спецификации.

- ***Интеграционное тестирование*** призвано проверить все аспекты работы программы от правильности взаимодействия внутренних программных компонентов до правильности взаимодействия программного комплекса с его пользователями.

- Во время ***пользовательского тестирования*** результаты работы программы проверяются с прикладной точки зрения.

- ***Нагрузочное тестирование*** дает возможность проверить безопасную и эффективную работу созданной программы в нормальном и пиковом режимах ее использования. Функциональность на этом этапе проверяется только в смысле ее влияния на важнейшие технические параметры программы, например, на время реакции системы на запрос пользователя.

- Важной в тестировании является возможность проведения ***регрессионного тестирования***. Регрессионные тесты, повторяемые после каждого исправления программы, позволяют убедиться, что функциональность программы, не связанная с внесенным исправлением, не затронута этим исправлением и не утрачена из-за него.

Редактор связей

Редактор связей (компоновщик) предназначен для связывания между собой (по внешним данным) объектных файлов, порождаемых компилятором, а также файлов библиотек, входящих в состав СП.

Редактор связей выполняет следующее:

- связывает между собой по внешним данным объектные модули, порождаемые компилятором и составляющие единую программу,
- связывает файлы статически подключаемых библиотек с целью получения единого исполняемого модуля,
- готовит таблицу трансляции относительных адресов для загрузчика,
- готовит таблицу точек вызова функций динамически подключаемых библиотек.

Типы библиотек

Библиотеки являются существенной частью систем программирования.

В настоящее время можно выделить 3 типа библиотек:

1. Библиотеки функций (или подпрограмм).
2. Библиотеки классов.
3. Библиотеки компонентов.

Библиотеки функций

Библиотеки функций во многом определяют возможности систем программирования в целом. Чем больше выбор библиотечных функций СП предоставляет пользователю, тем лучшие позиции она имеет на рынке средств разработки программного обеспечения.

Различают:

- *библиотеки для языков программирования* (например, функции ввода-вывода, работа со строками) и

- *библиотеки для решения задач в конкретной проблемной области* (например, функции, реализующие алгоритмы линейной алгебры).

Библиотеки функций представляют собой **откомпилированные объектные модули,** а необходимые фрагменты библиотеки функций включаются в исполняемый файл на этапе работы редактора связей**.**

Библиотеки классов

Библиотеки классов также являются важной частью современных систем программирования, базирующихся на ООЯП.

Недостаток библиотеки классов — все ее классы должны быть написаны на том же ЯП, на котором пишется программа, куда интегрируются библиотечные классы.

В библиотеке классов различают:

- *конкретные классы;*
- *абстрактные классы, иерархии классов;*
- *шаблоны классов,* иерархии шаблонов классов.

Библиотеки классов включаются в программу на этапе компиляции и компилируются со всей программой вместе.

Библиотеки компонентов

Библиотеки компонентов - это библиотеки готовых откомпилированных программных модулей, предназначенных для использования в качестве составных частей программ, и которыми можно манипулировать во время разработки программ.

Компоненты бывают **локальные** (находящиеся на той же ЭВМ, где создается ПП) и **распределенные** (расположенные на сервере и доступные по сети ЭВМ).

Примеры технологий, использующих библиотеки компонентов:

- Технология CORBA (Common Object Request Broker Architecture) от международной группы OMG позволяет использовать программные компоненты, размещённые как локально, так и дистанционно. Использование CORBA-компонент не зависит от языка, на котором они были написаны.

- Технология COM (Common Object Model) от компании Microsoft под ОС Windows позволяет использовать локально размещённые компоненты, независимо от языка их реализации. Её развитие привело к распределённой архитектуре DCOM (Distributed COM), а затем к ActiveX.

- Технология Java Beans от Sun Microsystems позволяет использовать компоненты, написанные на языке Java. Так как реализация Java-машины существует почти для всех ОС, отсутствует жёсткая привязка к конкретной ОС.

Динамически подключаемые библиотеки (ДБ)

ДБ в отличие от статических библиотек подключаются к программе не во время компиляции программы, а непосредственно в ходе её выполнения.

На этапе компоновки программы редактор связей, встречая вызовы функций ДБ, вместо процедуры связывания формирует таблицу точек вызова функций ДБ для последующей операции динамического связывания. Таким образом, процесс полной компоновки завершается уже на этапе выполнения целевой программы.

Преимущества ДБ:

• не требуется включать в программу объектный код часто используемых функций, что существенно сокращает объем кода;

• различные программы, выполняемые в некоторой ОС, могут пользоваться кодом одной и той же библиотеки, содержащейся в ОС;

• изменения и улучшения функций библиотек сводится к обновлению только содержимого ДБ, а уже существующие тексты программ не требуют перекомпиляции (этот же факт может оказаться недостатком, если при модификации функций меняется логика их работы, поэтому использование ДБ накладывает определенные обязательства как на разработчика программы, так и на создателя библиотеки).

Как правило, динамически подключаются системные функции ОС и общедоступные функции программного интерфейса (API).

Существует возможность создавать свои ДБ для использования при разработке прикладных программ.

Критерии проектирования стандартных библиотек. Требования по составу

Стандартная библиотека должна:

- обеспечивать поддержку свойств языка (например, управление памятью, предоставление информации об объектах во время выполнения программ);

- предоставлять информацию о зависящих от реализации аспектах языка, (например, о максимальных размерах целых значений);

- предоставлять функции, которые не могут быть написаны оптимально для всех вычислительных систем на данном языке программирования (например, *sqrt*() или *memmove*() — пересылка блоков памяти);

- предоставлять программисту нетривиальные средства, на которые он может рассчитывать, заботясь о переносимости программ (например, средства работы со списками, функции сортировки, потоки ввода/вывода);

- предоставлять основу для расширения собственных возможностей, в частности, соглашения и средства поддержки, позволяющие обеспечить операции для данных, имеющих определяемые пользователями типы, в том же стиле, в котором обеспечиваются операции для встроенных типов (например, ввод/вывод);

- служить основой и теоретическим базисом других библиотек.

Требования по свойствам компонентов стандартной библиотеки (1)

Компоненты стандартной библиотеки должны:

- иметь ***общезначимый*** характер (структуры данных и алгоритмы для работы с ними – стек, очередь, список, …, сортировка, поиск, копирование, …); быть важными и удобными для использования всеми программистами;

- быть настолько ***эффективными***, чтобы у пользователей библиотеки не возникало потребности заново программировать библиотечные средства;

- быть ***независимыми от*** конкретных ***алгоритмов*** или предоставлять возможность указывать алгоритм в качестве параметра;

- оставаться элементарными, чтобы не терять эффективности из-за излишних усложнений или попыток совместить различные функции в одной;

Требования по свойствам компонентов стандартной библиотеки (2)

- быть ***безопасными*** (устойчивыми к неправильному использованию, использование библиотеки не должно провоцировать ошибки, а наоборот, снижать их вероятность);

- обладать достаточной полнотой (***завершенностью***), чтобы ни у кого не возникало желания что-то заменить или доопределить;

- обладать удобной и безопасной системой умолчаний;

- поддерживать общепринятые стили программирования;

- обладать способностью к расширению, чтобы *работать с типами, определяемыми пользователем, было так же хорошо, как и со встроенными (базовыми) типами (**сочетаемость с базовыми типами данных и базовыми операциями**)*.

СП под UNIX. Координатор GNU Make.

Make существенно упрощает процесс сборки проектов.

Make отслеживает изменившиеся файлы и перекомпилирует при обращении к нему только их и файлы, связанные с ними по компиляции.

Информация о зависимостях по компиляции и необходимые команды по компиляции содержатся в файле Makefile (makefile или в файле с соответствующей структурой, имя которого задается при обращении к Make: Make -f <имя_файла>), который должен находиться в текущей директории.

Makefile состоит из последовательности записей вида:

```
цель: зависимости_по_компиляции
    команда ОС UNIX
    ...
    команда ОС UNIX
...
```

Цель - имя целевого файла или название действия.

Если обращение к Make происходит без параметра, то выполняются действия по достижению первой цели, если же параметр есть, то Make достигает цель, имя которой совпадает с именем параметра.

Если цель - имя файла, Make автоматически по дате модификации файлов, указанных среди файлов-зависимостей по компиляции, определяет, какие из них должны быть перекомпилированы и выполняет соответствующие команды.

В Makefile должны быть указаны зависимости и команды для получения как промежуточных объектных файлов, так и исполняемых файлов.

Пример 1. Makefile (для модельного SQL-интерпретатора):

```
client: client.o
        cc -o client client.o

server: server.o  parse.o  getlex.o  table.o
        cc -o server server.o  parse.o  getlex.o  table.o

table.o: table.c  table.h
        cc –c  table.c

parse.o: parse.c  parse.h  getlex.h  table.h
        cc –c  parse.c

getlex.o: getlex.c  parse.h  getlex.h
        cc –c  getlex.c

server.o: server.c  parse.h  getlex.h
        cc –c server.c

client.o: client.c
        cc –c  client.c

clean:
        rm *.o

all:  client   server
```

Некоторые дополнительные возможности создания Make-файлов

В начале файла можно вводить макросы для обозначения каких-либо часто повторяющихся фрагментов текста файла в виде:

имя = текст ,

а затем там, где нужно, использовать введенные имена таким образом:

$(имя) .

Некоторые предопределенные макроопределения:

$@ - полное имя текущей цели,

$* - имя текущей цели без типа файла (суффикса),

$? - список зависимостей, которые обновились с момента предыдущего обновления цели,

$< - полное имя исходного файла, к которому применяется правило трансформации.

В язык для Makefile введены некоторые предопределенные правила суффиксов по умолчанию для стандартного получения результирующих файлов по исходным файлам.

Например, правило трансформации ***.c.o*** означает, что для того, чтобы получить файл с расширением *.o* (если есть файл с расширением *.c*), надо выполнить указанную команду.

Строка, начинающаяся символом '#' , является комментарием.

Пустые строки игнорируются.

Любая строка, оканчивающаяся символом '\', продолжается на следующую строку.

gcc -MM позволяет сгенерировать фрагмент make-файла с зависимостями модулей.

Пример 2. Makefile (для модельного SQL-интерпретатора):

```
cc = gcc
serv_o = server.o parse.o getlex.o table.o

client:  client.o
	$(cc) -o client client.o

server: $(serv_o)
	$(cc) -o server $(serv_o)

.c.o:
	$(cc) -c $*.c

table.c:       table.h
parse.c:       parse.h getlex.h table.h
getlex.c:      parse.h getlex.h
server.c:      parse.h getlex.h

clean:
	rm *.o

all:    client server
```

Системы контроля версий

Система контроля версий в самом общем понимании осуществляет отслеживание версий (ревизий) некоего набора объектов (например, файлового дерева).

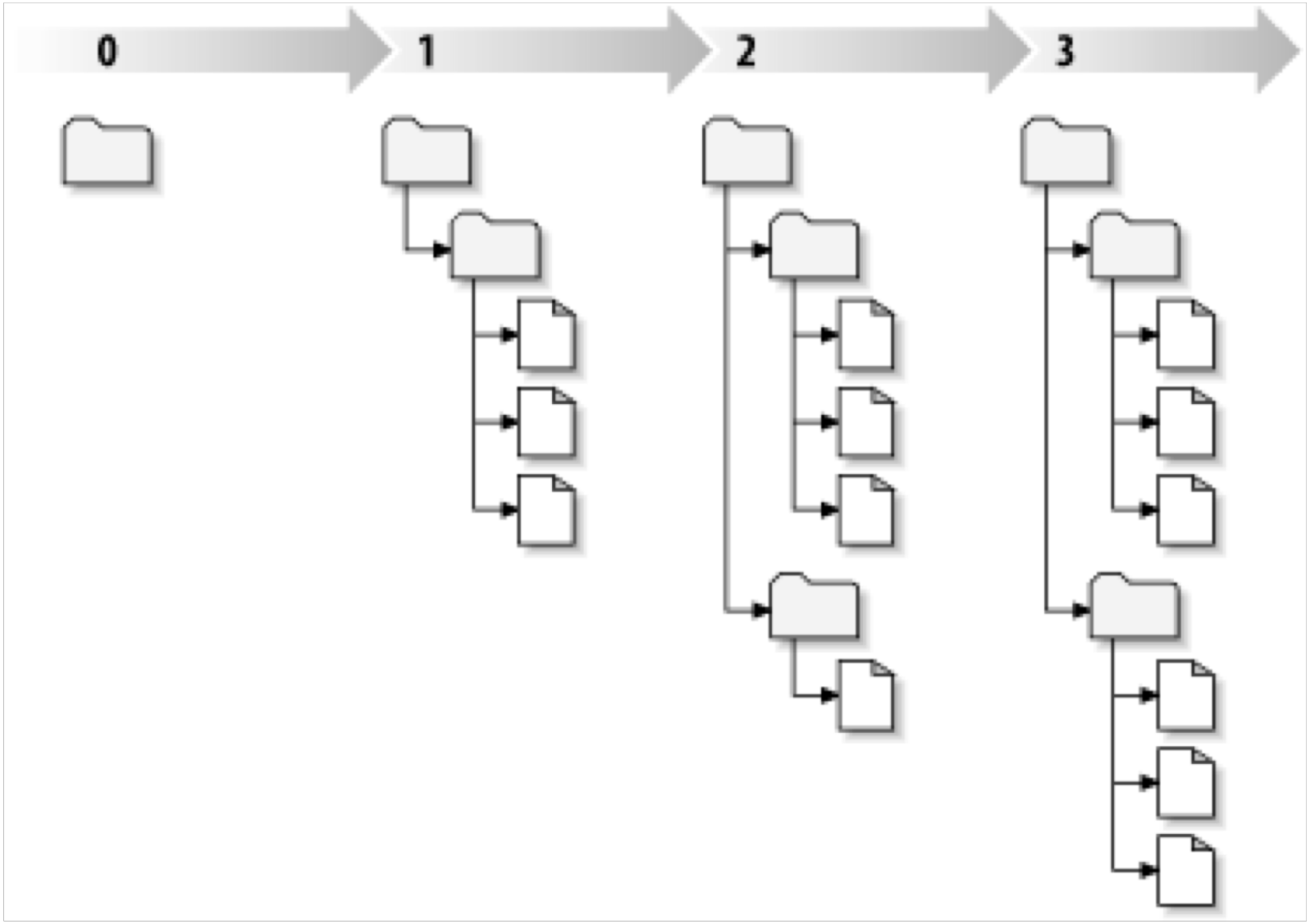
Применительно к процессу разработки ПП говорят

• об **управлении исходным кодом** (source control),
при этом отслеживаются ревизии исходного кода ПП и других ресурсов, необходимых для сборки ПП,

• или об **управлении конфигурациями** (configuration management),
при этом отслеживаются ревизии всего окружения, в том числе сопутствующих материалов, таких как файлы данных и документация.

Система контроля версий позволяет фиксировать ревизии исходного кода ПП, возвращаться к любой из них, отслеживать авторство изменений, производить анализ истории изменений и т. д.

Системы контроля версий



Развитие систем контроля версий

- Старейшие системы:
  SCCS (1972), RCS (1982)
  - отслеживается один файл на одной машине
  - рядом с файлом хранится история всех его ревизий
- Проектные клиент-серверные системы:
  CVS (1990), SVN (2000)
  - поддерживается отслеживание файлового дерева
  - фиксируется ревизия дерева в целом
  - история ревизий хранится в репозитории (на сервере)
  - работа с кодом ведется в рабочих копиях
- Распределенные системы:
  BitKeeper (1998), Darcs (2002), Git (2005), Mercurial (2005)
  - каждая рабочая копия может иметь собственный репозиторий
  - работа с репозиторием не требует доступа к серверу
  - возможна децентрализованная разработка

Контроль версий: основные понятия

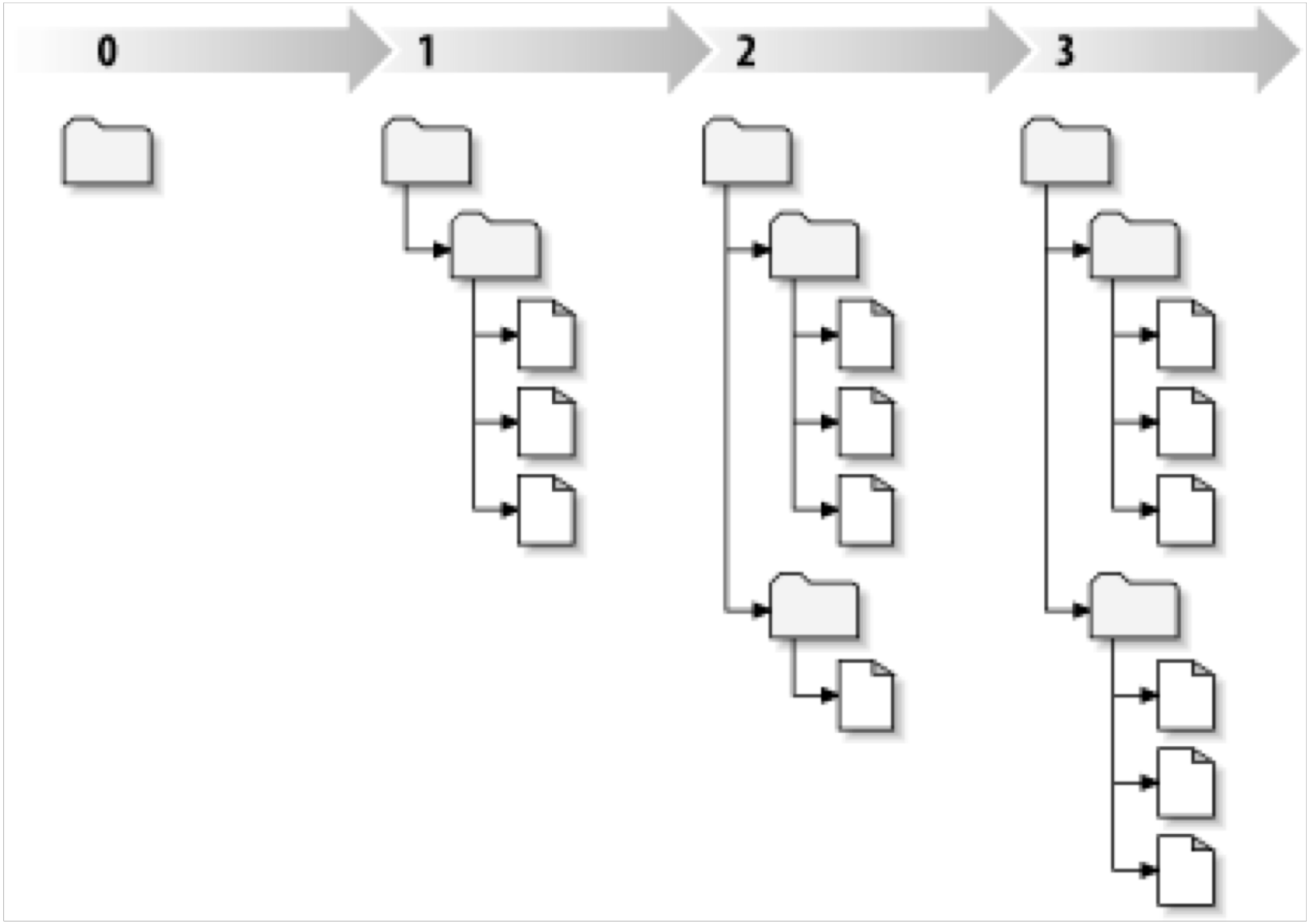
Сущности

- **Дерево**
- **Ревизия**
- **Набор изменений** (changeset)
- **Ветка**
- **Репозиторий**
- **Рабочая копия**

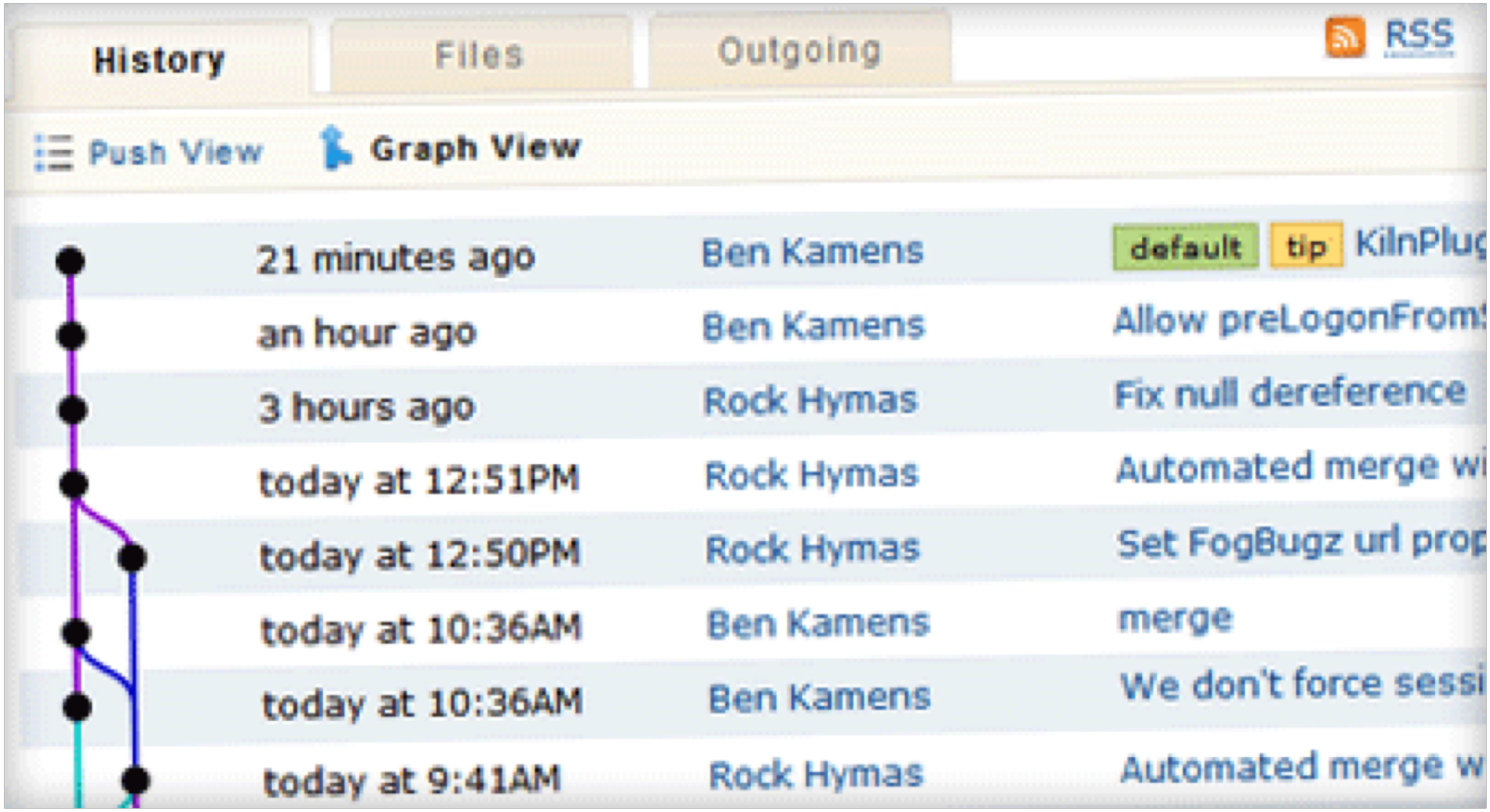
Операции

- **Фиксация** (commit)
- **Обновление** на ревизию
- **Ветвление**
- **Слияние** (merge)
- **Передача изменений** (pull/push)

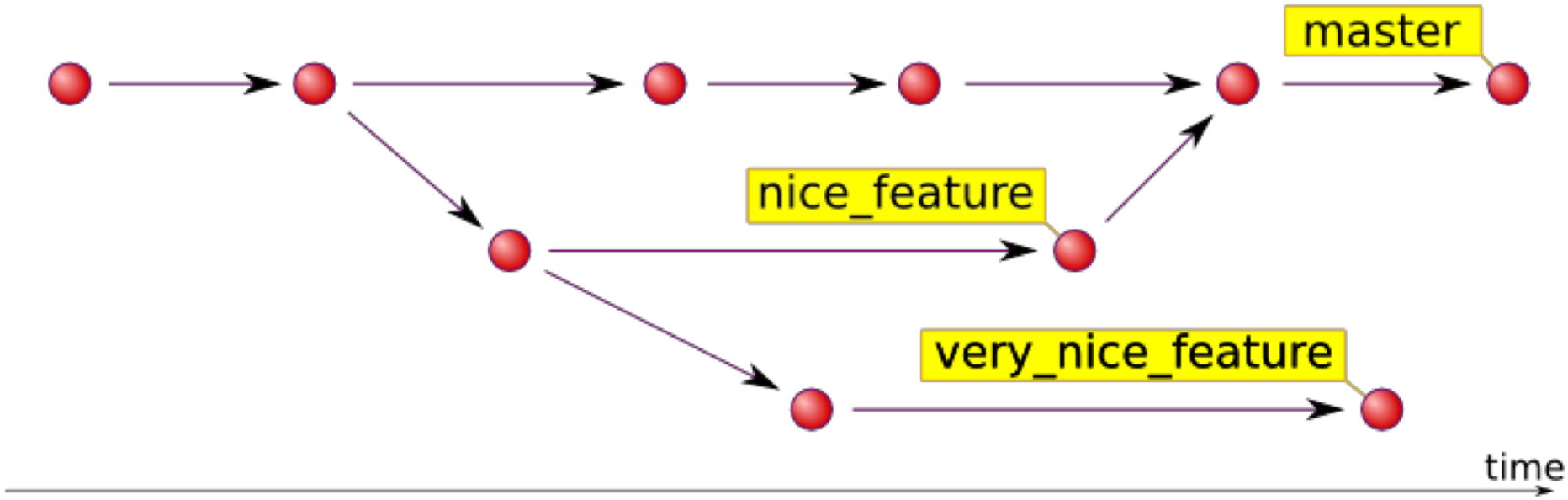
История изменений дерева



История изменений дерева



Ветки



Контроль версий: классификация

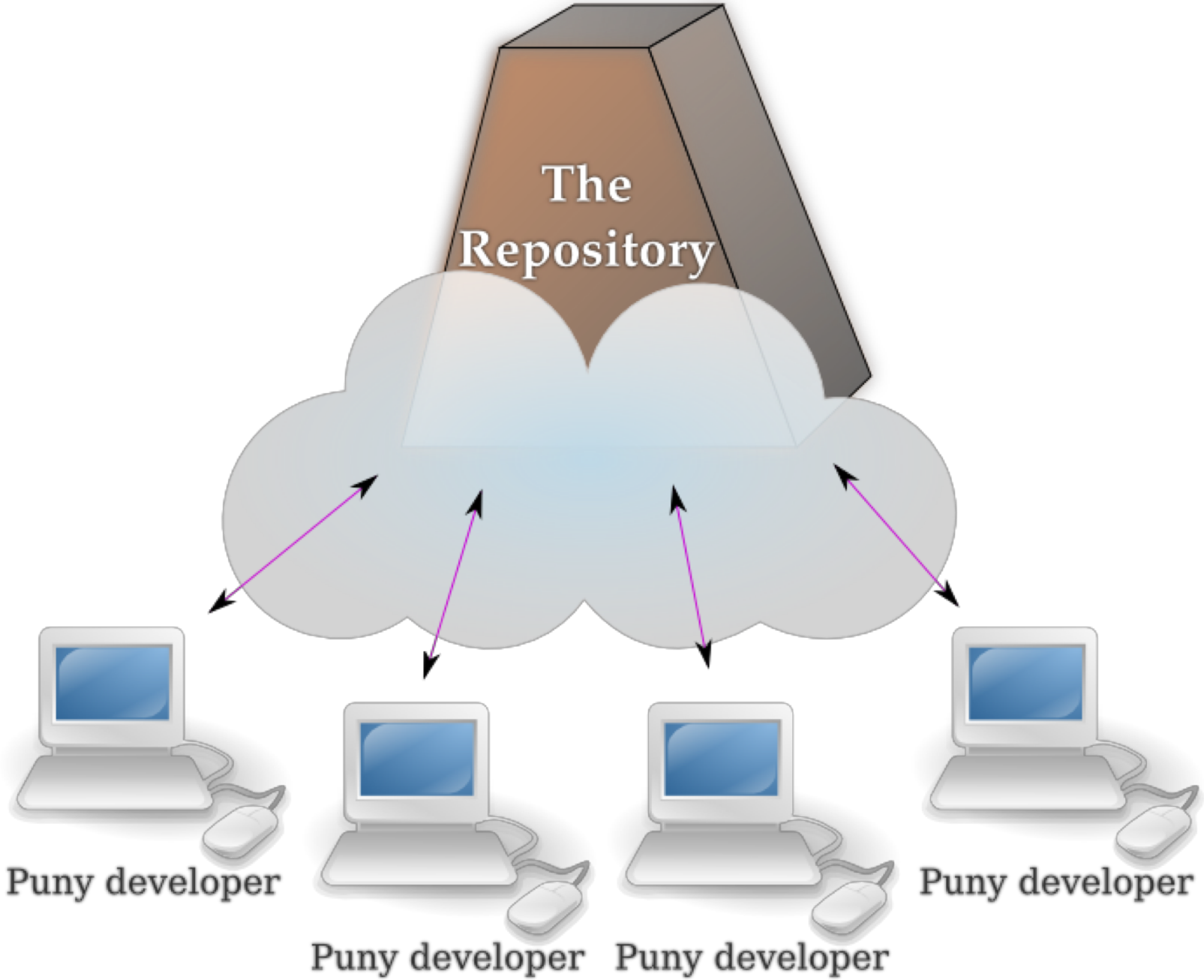
По расположению репозитория:

- централизованные (CVS, SVN),
- распределенные (Git, Mercurial, Darcs),
- комбинированные (Bazaar).

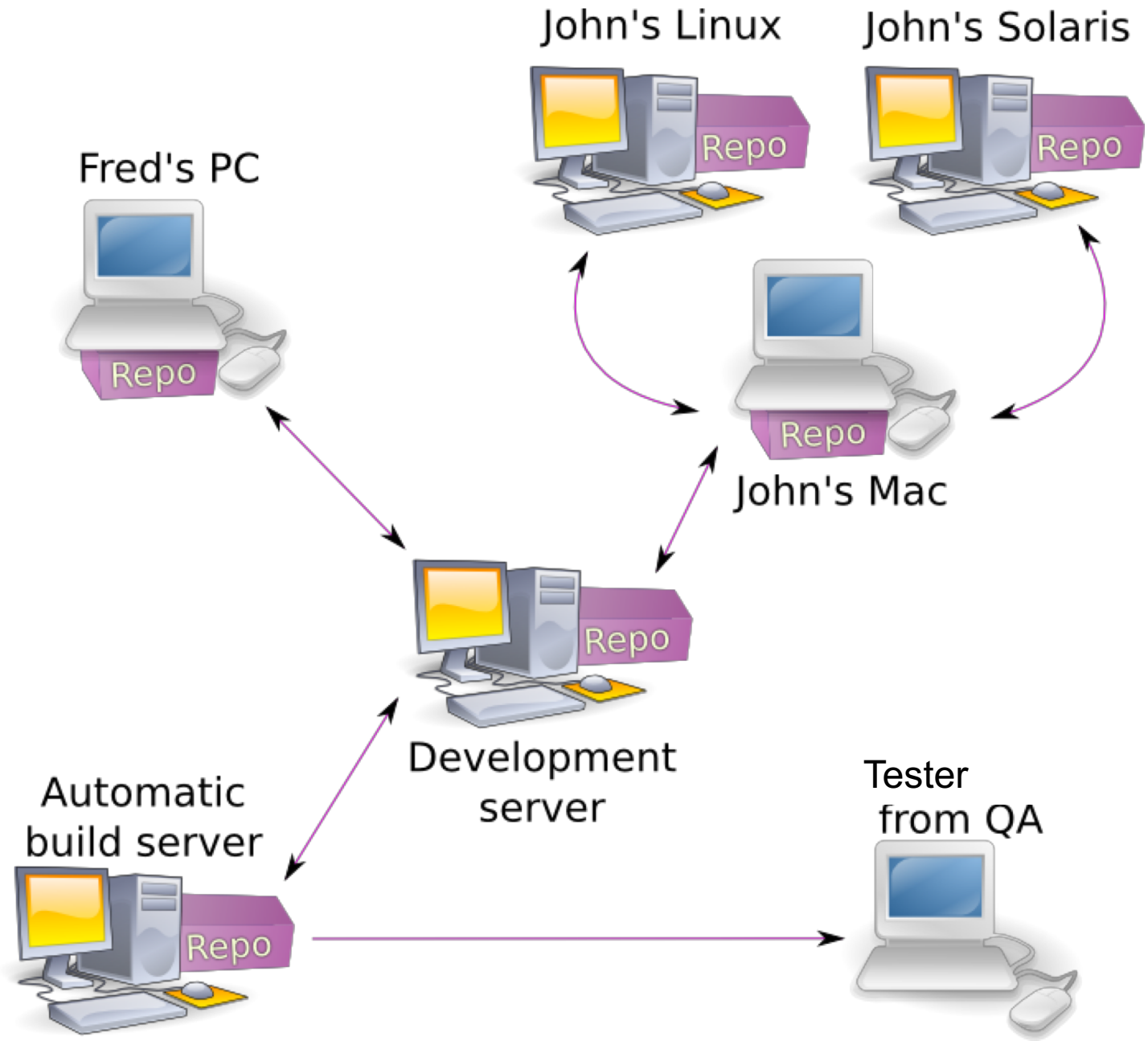
По объекту отслеживания:

- отслеживающие ревизии (CVS, SVN),
- отслеживающие наборы изменений:
  - наборы изменений организованы в ациклический орграф (Git, Mercurial)
  - наборы изменений организованы как набор патчей (Darcs)

Централизованные системы



Распределенные системы



Приемы работы с системами контроля версий

1. **Линейная работа**. Последовательная фиксация прогресса работы над ПП.
2. **Совместная линейная работа**. Совместная работа с фиксацией по принципу «кто успел» и последующим слиянием изменений.
3. **Ветки для разработки**, в которых новая функциональность дорабатывается до готовности к выпуску версии ПП.
4. **Ветки для тестирования**.
5. **Ветки для сопровождения старых версий.**
6. **Метки** (тэги) для фиксации значимых ревизий (например, версий ПП).
7. **Анализ истории** (в том числе аннотирование кода).

Режим аннотирования кода

```
[1586]  110 class ReportModule(Component):
[1]     111
[1860]  112     implements(INavigationContributor, IPermissionRequestor, IRequestHandler,
        113                IWikiSyntaxProvider)
[1586]  114
[6901]  115     items_per_page = IntOption('report', 'items_per_page', 100,
        116         """Number of tickets displayed per page in ticket reports,
        117         by default (''since 0.11'')""")
        118
        119     items_per_page_rss = IntOption('report', 'items_per_page_rss', 0,
        120         """Number of tickets displayed in the rss feeds for reports
        121         (''since 0.11'')""")
[11096] 122
[1586]  123     # INavigationContributor methods
        124
        125     def get_active_navigation_item(self, req):
        126         return 'tickets'
        127
        128     def get_navigation_items(self, req):
[4143]  129         if 'REPORT_VIEW' in req.perm:
[5776]  130             yield ('mainnav', 'tickets', tag.a(_('View Tickets'),
[4787]  131                                                href=req.href.report()))
[1586]  132
[11096] 133     # IPermissionRequestor methods
[1860]  134
[11096] 135     def get_permission_actions(self):
```

Популярные современные системы

1. Git
   - Высокая скорость
   - Github
2. Mercurial
   - Простота в использовании
   - Кроссплатформенность
3. Subversion (SVN)
   - Централизованная система

CASE-средства

CASE-средства (Computer Aided Software Engineering) – программные средства, поддерживающие полуавтоматическую разработку комплексного ПП на всех стадиях его жизненного цикла.

Основные характерные особенности CASE-средств:

- наличие мощных **графических средств** для описания и документирования ПП, обеспечивающие удобный интерфейс с разработчиком и развивающие его творческие возможности;
- **интеграция** отдельных компонент CASE-средств, обеспечивающая управляемость процессом разработки программной системы;
- наличие средств автоматического или автоматизированного кодирования и документирования ПП

Современные CASE-средства

Примером наиболее известного CASE-средства является объектно-ориентированное CASE-средство

Rational Rose (компании Rational Software Corporation).

В основе работы Rational Rose лежит построение диаграмм и спецификаций унифицированного языка моделирования

UML (Unified Modeling Language),

определяющих архитектуру проекта, его статические и динамические аспекты.

UML — язык для определения, представления, проектирования и документирования программных систем различной природы.

Некоторые дополнительные возможности современных систем программирования

Существует множество других программных средств, помогающих в проектировании, модификации и кодировании программ. Например,

- системы, преобразующие программы на процедурном (императивном) ЯП в программы на ООЯП (поскольку ОО программы считаются более простыми в сопровождении, это бывает полезно),
- системы, анализирующие исходный код, с целью получения высокоуровневых абстракций (например, Java $\Rightarrow$ диаграммы UML),
- средства, предоставляющие среду разработки программ по диаграммам UML,
- средства сборочного программирования (из готовых программных модулей, официально распространены достаточно слабо в связи с различными правовыми проблемами копирования модулей, низким качеством модулей и их плохой документацией).

Статистика отмечает, что около 80% программного обеспечения создается по уже имеющемуся.

Следовательно, большой интерес представляют собой репозитории, поддерживающие архивы, документацию и интеллектуальный поиск нужных прототипов и фрагментов проектов программ для реализации эффективного сборочного программирования.

Элементы теории трансляции

Транслятор

позволяет преобразовать программу, написанную на ЯП, отличном от машинного языка, к виду, допускающему выполнение на ЭВМ.

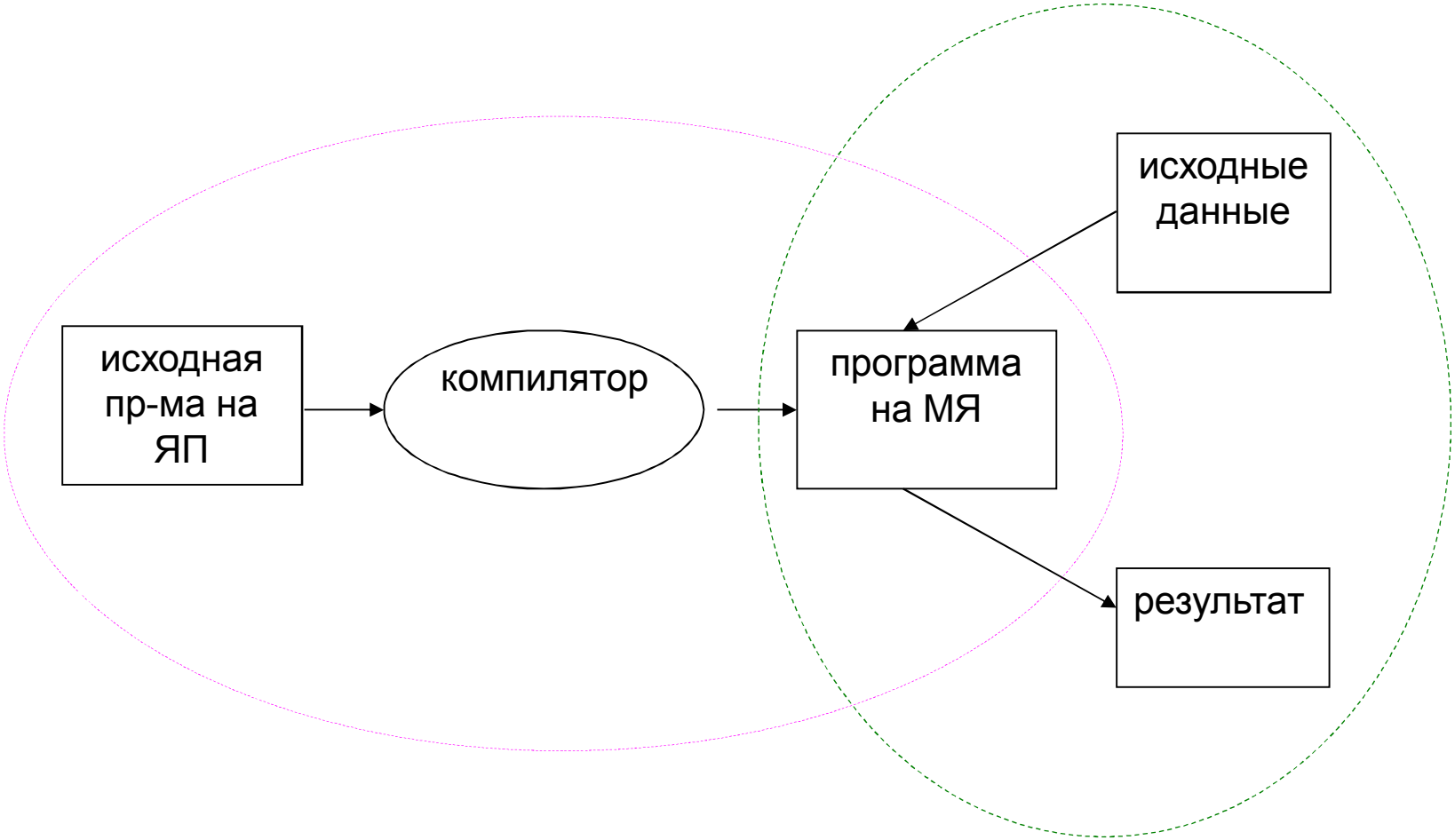
Компилятор

на вход получает программу на некотором ЯП (немашинном), а на выходе выдает объектный модуль (программу на машинном языке).

Интерпретатор

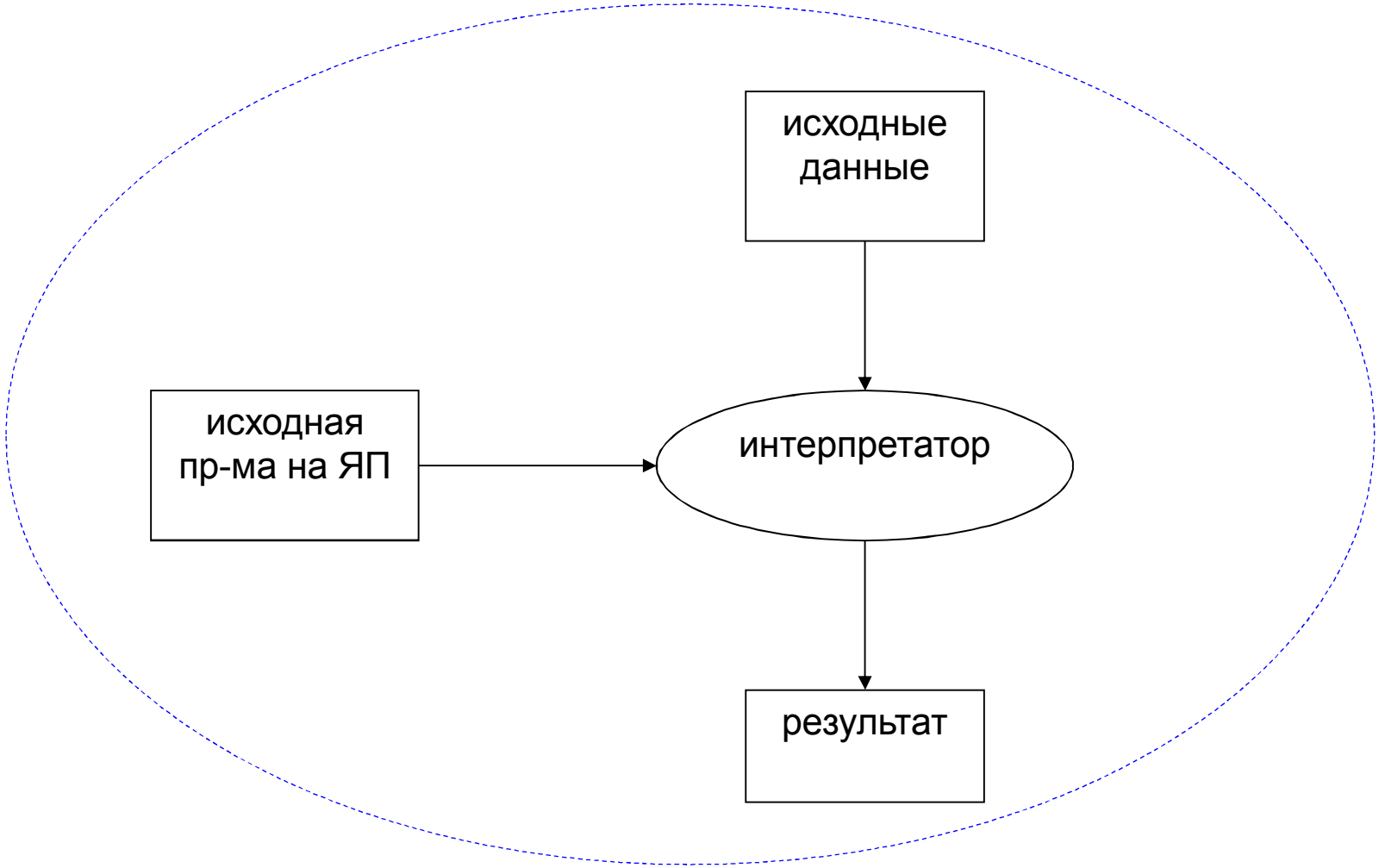
на вход получает программу на некотором ЯП (немашинном) и, считывая предложение за предложением исходной программы, анализирует их и тут же выполняет действия, указанные в этих предложениях.

Система программирования компилирующего типа



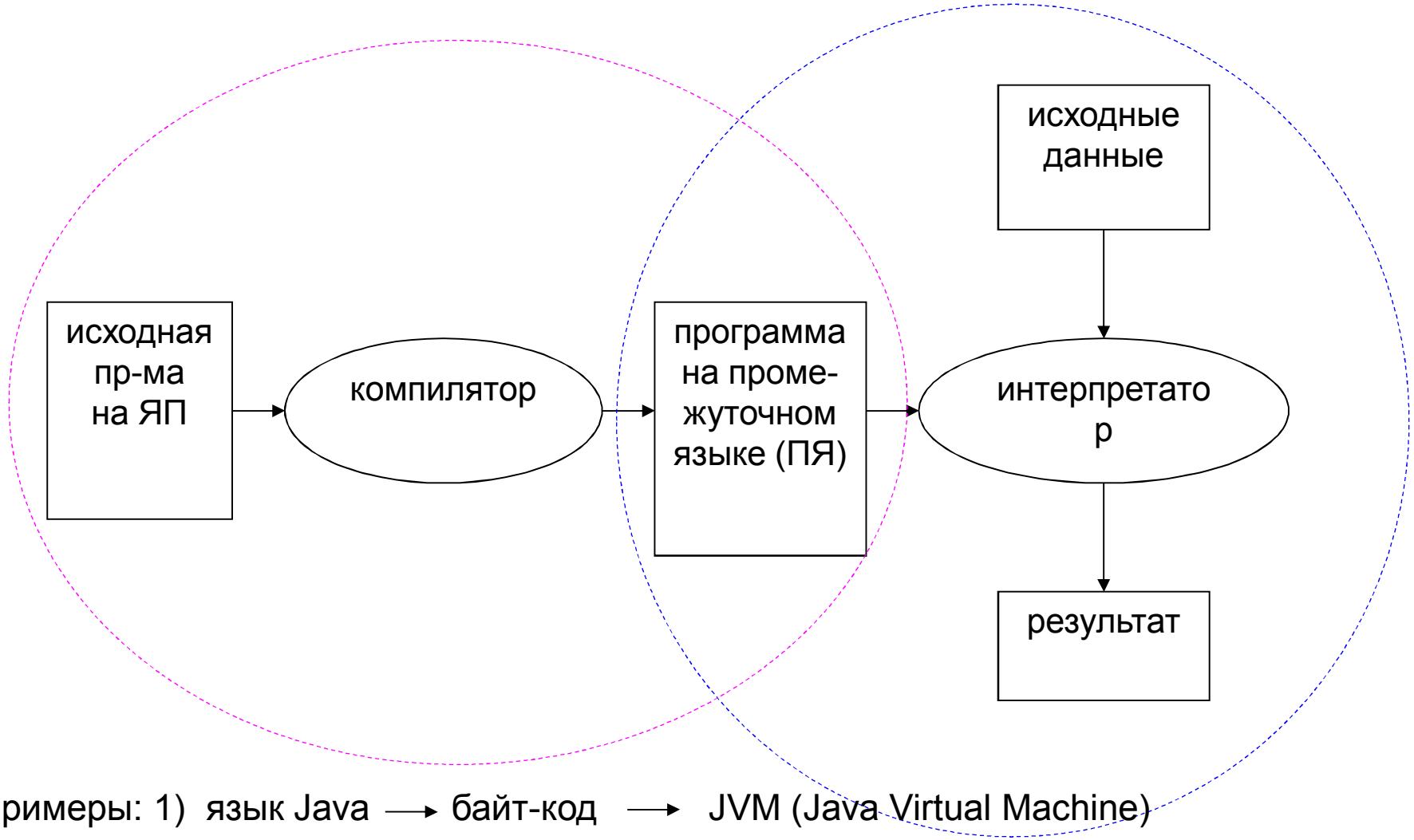
Примеры: MASM, Турбо Паскаль, gcc в UNIX

Система программирования интерпретирующего типа



Примеры: QBasic, командные интерпретаторы в UNIX -- bash, csh и др.

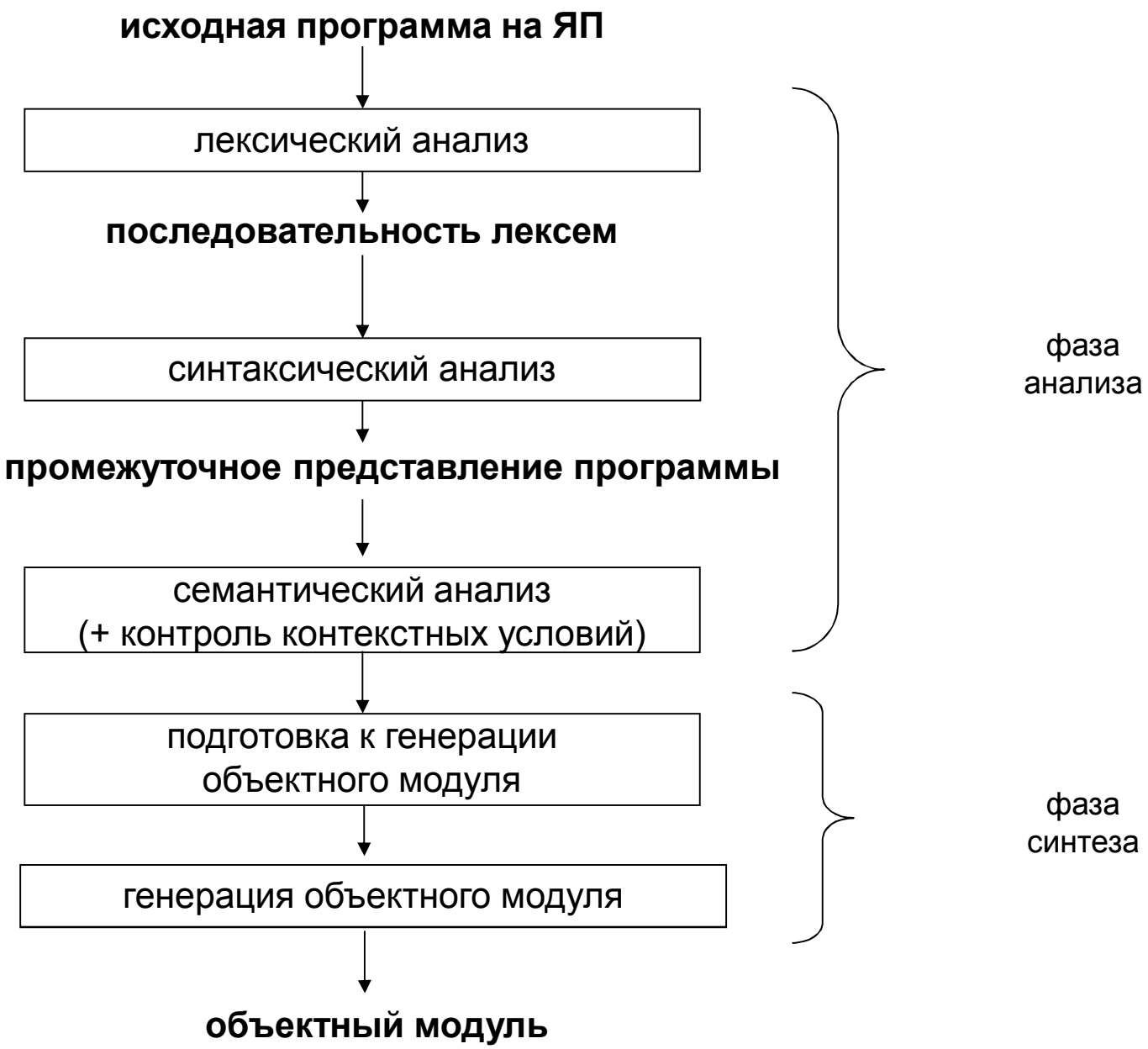
Смешанная стратегия



Примеры: 1) язык Java ⟶ байт-код ⟶ JVM (Java Virtual Machine)

2) Языки на технологии .NET (Basic,C#,C++) ⟶ промежуточный язык: CIL – Common Intermediate Language ⟶ JIT-компилятор (just-in-time)

Схема функционирования компилятора



Основные понятия теории формальных языков

Алфавит:

это конечное множество символов.

Пример: V = {a,b,c}

Цепочка символов в алфавите V :

любая конечная последовательность символов этого алфавита.

Пример: V = {a,b}. Цепочка: abbba

Пустая цепочка :

цепочка, которая не содержит ни одного символа.

Обозначение: ε

(иногда для этой цели используется символ Λ)

Конкатенация (сцепление) цепочек α и β:

цепочка, полученная приписыванием последовательности β справа к α

Обозначение: $\alpha\cdot\beta$ (или $\alpha\beta$)

Пример: если α = ab и β = cd, то

$\alpha\beta$ = abcd.

- Для любой цепочки α верно $\alpha\varepsilon = \varepsilon\alpha = \alpha$.

  (Аналог для чисел: $\forall$ x верно $1\cdot x = x\cdot 1 = x$)

- Операция конкатенации ассоциативна: $(\alpha\cdot\beta)\cdot\gamma = \alpha\cdot(\beta\cdot\gamma)$ для любых α, β, γ.

- Операция конкатенации некоммутативна:

  например, для α = ab и β = cd $\quad \alpha\cdot\beta \neq \beta\cdot\alpha$

Обращение *(*или *реверс)* цепочки α:
цепочка, символы которой записаны в обратном порядке.

Обозначение: α^R

Пример: если α = abcdef, то α^R = fedcba.

Для пустой цепочки: $\varepsilon = \varepsilon^R$.

Рекурсивное определение:

$$\alpha^R = \begin{cases} \varepsilon, & \text{если } \alpha = \varepsilon; \\ \beta^R s, & \text{если } \alpha = s\beta \text{, где s — символ алфавита} \end{cases}$$

n-я степень цепочки α (обозначается α^n) :

конкатенация n цепочек α

$$\underbrace{\alpha\ \alpha\ \alpha\ \ldots\ \alpha\ \alpha}_{n\ \text{раз}} = \alpha^n$$

Рекурсивное определение:

$$\alpha^n = \begin{cases} \varepsilon, & \text{если } n=0; \\ \alpha^{n-1}\alpha, & \text{если } n>0 \end{cases}$$

Длина *цепочки* :
это число составляющих ее символов

Пример: если α = abcdefg, то длина α равна 7.

Обозначение: $|\alpha|$

$|\varepsilon| = 0$

Рекурсивное определение:

$$|\alpha| = \begin{cases} 0, & \text{если } \alpha=\varepsilon; \\ |\beta| + 1, & \text{если } \alpha = \beta s, \text{ где } s \text{ — символ алфавита} \end{cases}$$

Обозначение $|\alpha|_s$ используется для числа вхождений символа s в цепочку α.

Язык в алфавите V :

подмножество цепочек конечной длины в этом алфавите.

V^* :

множество, содержащее все цепочки конечной длины в алфавите V, включая пустую цепочку ε.

Пример: $V = \{0,1\}$,

$V^* = \{\varepsilon, 0, 1, 00, 11, 01, 10, 000, 001, 011, ...\}$.

V^+ :

множество, содержащее все цепочки конечной длины в алфавите V, исключая пустую цепочку ε.

- $V^* = V^+ \cup \{\varepsilon\}$
- *Для любого языка* $L \subseteq V^*$

способы описания языков

Явное перечисление:

L = {ab, abb,ba, baa}
Язык L — конечный

Формулы*:*

$L = \{a^n b^n \mid n \geq 0\}$
Цепочки языка L: ε, ab, aabb, aaabbb, …

Порождающие грамматики Хомского

Распознаватели (МТ, ЛОА, конечные автоматы, МП-автоматы)

Декартово произведение множеств A и B :

множество $\{ (a,b) \mid a \in A, b \in B\}$ Обозначение: $A \times B$

Порождающая грамматика G :

это четверка $\langle T, N, P, S \rangle$, где

- T – алфавит *терминальных символов (терминалов)*,
- N – алфавит *нетерминальных символов (нетерминалов)*, не пересекающийся с T,
- P – конечное подмножество множества $(T \cup N)^+ \times (T \cup N)^*$;

элемент (α, β) множества P называется *правилом вывода* и записывается в виде $\alpha \to \beta$,

α содержит хотя бы один нетерминал;

S – *начальный символ (цель)* грамматики, $S \in N$.

правила

$\alpha \to \beta_1 \quad \alpha \to \beta_2 \ \ldots \ \alpha \to \beta_n$

записываются сокращенно

$\alpha \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

β_i для $i = 1, 2, \ldots, n$, - ***альтернатива*** правила вывода из цепочки α.

Порождающая грамматика G :

это четверка $\langle T, N, P, S \rangle$, где

- T – алфавит *терминальных символов (терминалов)*,
- N – алфавит *нетерминальных символов (нетерминалов)*, $T \cap N = \varnothing$,
- P – конечное подмножество множества $(T \cup N)^+ \times (T \cup N)^*$; элемент (α, β) множества P называется *правилом вывода* и записывается в виде $\alpha \to \beta$,
- S – *начальный символ (цель)* грамматики, $S \in N$.

Пример грамматики: $G_1 = \langle \{0,1\}, \{A, S\}, P, S \rangle$

P:

$S \to 0A1$

$0A \to 00A1$

$A \to \varepsilon$

Пусть $\beta \in (T \cup N)^*$ $\alpha \in (T \cup N)^+$

β ***непосредственно выводима*** *из* α
в грамматике $G = \langle T, N, P, S \rangle$,
если $\alpha = \xi_1\gamma\xi_2$,
$\beta = \xi_1\delta\xi_2$,
где $\xi_1, \xi_2, \delta \in (T \cup N)^*$, $\gamma \in (T \cup N)^+$ и $\gamma \to \delta \in P$.

Обозначение: $\alpha \to \beta$

Пример: $G_1 = \langle \{0,1\}, \{A, S\}, P, S \rangle$
P:
$S \to 0A1$
$0A \to 00A1$
$A \to \varepsilon$

Цепочка 00A11 непосредственно выводима из 0A1 в грамматике G_1 : $\underline{0A}1 \to \underline{00A11}$ (по правилу $0A \to 00A1$)

Цепочка $\beta \in (T \cup N)^*$ ***выводима*** из цепочки $\alpha \in (T \cup N)^+$
в грамматике $G = \langle T, N, P, S \rangle$,
если существуют цепочки $\gamma_0, \gamma_1, \ldots, \gamma_n$ $(n \geq 0)$, такие, что

$$\alpha = \gamma_0 \to \gamma_1 \to \ldots \to \gamma_n = \beta.$$

Обозначение: $\alpha \Rightarrow \beta$

Вывод длины n :
последовательность $\gamma_0, \gamma_1, \ldots, \gamma_n$

Любая цепочка выводится сама из себя за 0 шагов: $\alpha = \gamma_0 = \alpha$

$G_1 = \langle \{0,1\}, \{A, S\}, P, S \rangle$
P:
$S \to 0A1$
$0A \to 00A1$
$A \to \varepsilon$

$S \Rightarrow 000A111$ в G_1, т. к. $\exists$ вывод $S \to 0A1 \to 00A11 \to 000A111$.

Длина вывода равна 3.

Сентенциальная форма в грамматике $G = \langle T, N, P, S \rangle$:
$\alpha \in (T \cup N)^*$, для которой $S \Rightarrow \alpha$

Язык, порождаемый грамматикой $G = \langle T, N, P, S \rangle$:
множество $L(G) = \{\alpha \in T^* \mid S \Rightarrow \alpha\}$

Грамматика — это не алгоритм, а система правил подстановки, позволяющих строить выводы.

Язык, порождаемый грамматикой $G = \langle T, N, P, S \rangle$:

множество $L(G) = \{ \alpha \in T^* \mid S \Rightarrow \alpha \}$

Пример: $G_1 = \langle \{0,1\}, \{A,S\}, P, S \rangle$,

P:

(1) $S \to 0A1$

(2) $0A \to 00A1$

(3) $A \to \varepsilon$

$L(G) = ?$

$S \to_{(1)} 0A1 \to_{(3)} 01$

$S \to_{(1)} 0A1 \to_{(2)} 00A11 \to_{(3)} 0011$

$S \to_{(1)} 0A1 \to_{(2)} 00A11 \to_{(2)} 000A111 \to_{(3)} 000111$

…

Предположительно: $L_1 = \{0^n 1^n \mid n>0\}$

Язык, порождаемый грамматикой $G = \langle T, N, P, S \rangle$:

множество $L(G) = \{ \alpha \in T^* \mid S \Rightarrow \alpha \}$

Пример: $G_1 = \langle \{0,1\}, \{A,S\}, P, S \rangle$,

P:

(1) $S \to 0A1$

(2) $0A \to 00A1$

(3) $A \to \varepsilon$

Предположительно: $L_1 = \{0^n1^n \mid n>0\}$

Требуется доказать: $L(G_1) = L_1$:

(1) $L_1 \subseteq L(G_1)$, т.е. $\forall x \in L_1 \Rightarrow x \in L(G_1)$ (индукция по n)

(2) $L(G_1) \subseteq L_1$, т.е. $\forall x \in L(G_1) \Rightarrow x \in L_1$ (индукция по длине вывода)

Эквивалентность грамматик G_1 и G_2:

$L(G_1) = L(G_2)$

Пример:

$G_1 = \langle \{0,1\}, \{A,S\}, P_1, S \rangle$ и $G_2 = \langle \{0,1\}, \{S\}, P2, S \rangle$

P_1: $S \to 0A1$

$0A \to 00A1$

$A \to \varepsilon$

P_2: $S \to 0S1 \mid 01$

$$L(G_1) = L(G_2) = \{0^n1^n \mid n>0\}$$

Грамматики G_1 и G_2 ***почти эквивалентны***,

если $L(G_1) \cup \{\varepsilon\} = L(G_2) \cup \{\varepsilon\}$.

Пример:

$G_1 = \langle \{0,1\}, \{A,S\}, P_1, S \rangle$ и $G_2' = \langle \{0,1\}, \{S\}, P_2', S \rangle$

P_1: $S \to 0A1$

$0A \to 00A1$

$A \to \varepsilon$

P_2' : $S \to 0S1 \mid \varepsilon$

$L(G_1)=\{0^n1^n \mid n>0\}$, а $L(G_2') = \{0^n1^n \mid n\geq 0\}$

т.е. $L(G_2')=L(G_1) \cup \{\varepsilon\}$

$G_1 = (\{0,1\}, \{A,S\}, P_1, S)$

P_1: $S \to 0A1$

$0A \to 00A1$

$A \to \varepsilon$

Рассмотрим

$G_3 = (\{0,1\}, \{S\}, P, S)$, где P:

(1) $S \to 0S$

(2) $S \to 1S$

(3) $S \to \varepsilon$

Любая цепочка вида 0^n1^n порождается следующим способом:

-- n раз применить правило (1), затем

-- n раз применить правило (2)

-- и на последнем шаге применить правило (3).

Однако $L(G_3) \neq L(G_1)$,

т.к. $S \to 1S \to 10S \to 10 \in L(G_3)$,

$10 \notin L(G_1)$

Классификация грамматик и языков по Хомскому

грамматики классифицируются по виду их правил вывода

Четыре типа грамматик:

тип 0, тип 1, тип 2, тип 3

Язык, порождаемый грамматикой типа k $(k=0,1,2,3)$, является языком *типа* k.

$$G = \langle T, N, P, S \rangle$$

Тип 0

Любая порождающая грамматика является грамматикой *типа 0*.

На вид правил грамматик этого типа не накладывается никаких дополнительных ограничений.

Класс языков типа 0 совпадает с классом рекурсивно перечислимых языков (распознаваемых МТ).

Грамматики с ограничениями на вид правил вывода

Тип 1

Грамматика $G = \langle T, N, P, S \rangle$ называется *неукорачивающей*, если правая часть каждого правила из P не короче левой части (т. е. для любого правила $\alpha \to \beta \in P$ выполняется неравенство $|\alpha| \le |\beta|$).

В виде исключения в неукорачивающей грамматике допускается наличие правила $S \to \varepsilon$, при условии, что S (начальный символ) не встречается в правых частях правил.

Грамматикой *типа 1* будем называть неукорачивающую грамматику.

Тип 1 в некоторых источниках определяют с помощью так называемых контекстно-зависимых грамматик.

Грамматика $G = \langle T, N, P, S \rangle$ называется *контекстно-зависимой* (*КЗ*), если каждое правило из P имеет вид $\alpha \to \beta$, где $\alpha = \xi_1 A \xi_2$, $\beta = \xi_1 \gamma \xi_2$, $A \in N$, $\gamma \in (T \cup N)^+$, $\xi_1, \xi_2 \in (T \cup N)^*$.

В виде исключения в КЗ-грамматике допускается наличие правила $S \to \varepsilon$, при условии, что S (начальный символ) не встречается в правых частях правил.

КЗ-грамматика удовлетворяет определению неукорачивающей.

Неукорачивающие и КЗ-грамматики определяют один и тот же класс языков.

Тип 2

Грамматика $G = \langle T, N, P, S \rangle$ называется *контекстно-свободной* (*КС*), если каждое правило из P имеет вид $A \to \beta$, где $A \in N, \beta \in (T \cup N)^*$.

В КС-грамматиках допускаются правила с пустыми правыми частями.

Язык, порождаемый контекстно-свободной грамматикой, называется *контекстно-свободным* языком.

Грамматикой *типа 2* будем называть контекстно-свободную грамматику.

Любую КС-грамматику можно преобразовать в эквивалентную неукорачивающую КС-грамматику. (т.е. КС, удовлетворяющую также и определению неукорачивающей)

Тип 3

Грамматика $G = \langle T, N, P, S \rangle$ называется *праволинейной*, если каждое правило из P имеет вид $A \to wB$ либо $A \to w$, где $A \in N, B \in N, w \in T^*$.

Грамматика $G = \langle T, N, P, S \rangle$ называется *леволинейной*, если каждое правило из P имеет вид $A \to Bw$ либо $A \to w$, где $A \in N$, $B \in N, w \in T^*$.

Праволинейные и леволинейные грамматики определяют один и тот же класс языков. Такие языки называются *регулярными.* Право- и леволинейные грамматики тоже называют регулярными.

Регулярная грамматика является грамматикой *типа 3*.

Автоматной грамматикой называется праволинейная (леволинейная) грамматика, такая, что каждое правило с непустой правой частью имеет вид: $A \to a$ либо $A \to aB$ (для леволинейной, соответственно, $A \to a$ либо $A \to Ba$), где $A \in N$, $B \in N$, $a \in T$.

Для любой регулярной (автоматной) грамматики G существует неукорачивающая регулярная (автоматная) грамматика G', такая что $L(G) = L(G')$.

Иерархия Хомского

Справедливы следующие соотношения:

1) любая регулярная грамматика является КС-грамматикой;

2) любая неукорачивающая КС-грамматика является КЗ-грамматикой;

3) любая неукорачивающая грамматика является грамматикой типа 0.

Неукорачивающие Регулярные $\subset$ *Неукорачивающие КС* $\subset$ *КЗ* $\subset$ *Тип 0*

(Запись $A \subset B$ означает, что A является собственным подклассом класса B)

Справедливы следующие соотношения для языков:

- каждый регулярный язык является КС-языком, но существуют КС-языки, которые не являются регулярными, например:

$$L = \{a^n b^n \mid n > 0\};$$

- каждый КС-язык является КЗ-языком, но существуют КЗ-языки, которые не являются КС-языками, например:

$$L = \{a^n b^n c^n \mid n > 0\};$$

- каждый КЗ-язык является языком типа 0 (т. е. рекурсивно перечислимым языком), но существуют языки типа 0, которые не являются КЗ-языками, например: язык, состоящий из записей самоприменимых алгоритмов Маркова в некотором алфавите.

Иерархия классов языков

Тип 3 (*Регулярные*) $\subset$ *Тип 2* (*КС*) $\subset$ *Тип 1* (*КЗ*) $\subset$ *Тип 0*

Проблема «Можно ли язык, описанный грамматикой типа k ($k = 0, 1, 2$), описать грамматикой типа $k + 1$?» является алгоритмически неразрешимой.

Язык $L_{a,b} = \{a, b\}$. Какого он типа? Обычно требуется указать максимально возможный тип.

Ответ: типа 3

$S \to a \mid b$ — грамматика типа 3, порождающая данный язык.

($L_{a,b}$ является также языком типа 2, 1, 0 в силу иерархии Хомского)

(1) ***Примеры грамматик и языков***

$$S \rightarrow ABCS \mid ABc$$
$$BA \rightarrow AB$$
$$CA \rightarrow AC$$
$$CB \rightarrow BC$$
$$Cc \rightarrow cc$$
$$Bc \rightarrow bc$$
$$Bb \rightarrow bb$$
$$Ab \rightarrow ab$$
$$Aa \rightarrow aa$$

Тип 1. Неукорачивающая, но не КЗ

Язык: $\{a^n b^n c^n \mid n > 0\}$

Примеры грамматик и языков

(2)

$$S \to aSb \mid ab$$

Язык: $\{a^n b^n \mid n > 0\}$

(3)

$$S \to aS \mid a$$

Язык: $\{a^n \mid n > 0\}$

Иерархия классов Хомского

Задача распознавания

Даны грамматика G и цепочка x

$x \in L(G)$?

Для грамматик типа 1 (а также типов 2 и 3) по классификации Хомского задача распознавания разрешима, т.е. существует общий алгоритм, отвечающий на вопрос: $x \in L(G)$?

Контекстно-свободные грамматики и языки

КС-грамматики позволяют выразить такие свойства языков программирования, как скобочные структуры, последовательность описаний и операторов и др. Но не могут задавать контекстно-зависимые свойства, например, соответствие числа формальных и фактических параметров при вызове функции. Для КС-грамматик существуют эффективные алгоритмы анализа, поэтому они применяются в трансляции, контекстные условия проверяются на

этапе семантического анализа

Левый (левосторонний) вывод цепочки $\beta \in T^*$ из $S \in N$ в КС-грамматике $G = (T, N, P, S)$:

в этом выводе каждая очередная сентенциальная форма получается из предыдущей заменой самого левого нетерминала.

Правый (правосторонний) вывод цепочки $\beta \in T^*$ из $S \in N$ в КС-грамматике $G = (T, N, P, S)$:

в этом выводе каждая очередная сентенциальная форма получается из предыдущей заменой самого правого нетерминала.

Рассмотрим пример грамматики:

$G = (\{a,b,+\}, \{S,T\}, \{S \to T \mid T+S;\ T \to a \mid b\}, S)$

можно построить выводы для цепочки a+b+a :

(1) $S \to T+S \to T+T+S \to T+T+T \to a+T+T \to a+b+T \to a+b+a$

(2) $S \to T+S \to a+S \to a+T+S \to a+b+S \to a+b+T \to a+b+a$

(3) $S \to T+S \to T+T+S \to T+T+T \to T+T+a \to T+b+a \to a+b+a$

Здесь (2) - левосторонний вывод, (3) - правосторонний, а (1) не является ни левосторонним, ни правосторонним

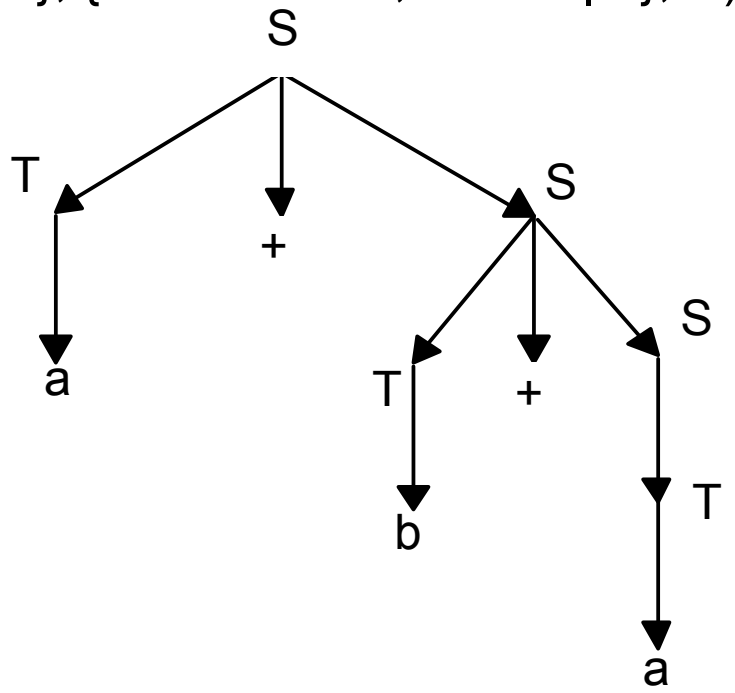
***Определение*:** упорядоченное ориентированное дерево называется **деревом вывода** (или **деревом разбора**) в КС-грамматике G = (T, N, P, S), если выполнены следующие условия:

(1) каждая вершина дерева помечена символом из множества $N \cup T \cup \{\varepsilon\}$, при этом корень дерева помечен символом S; листья - символами из $T \cup \{\varepsilon\}$;

(2) если вершина дерева помечена символом A , а ее непосредственные потомки - символами $a_1, a_2, \ldots, a_n$, где каждое $a_i \in T \cup N$, то $A \to a_1 a_2 \ldots a_n$ - правило вывода в этой грамматике;

(3) если вершина дерева помечена символом A , а ее единственный непосредственный потомок помечен символом ε, то $A \to \varepsilon$ — правило вывода в этой грамматике.

Пример дерева вывода для цепочки a+b+a в грамматике G = ({a,b,+}, {S,T}, {S → T | T+S; T → a | b}, S) :



(1) S→T+S → T+T+S → T+T+T → a+T+T → a+b+T → a+b+a

(2) S → T+S → a+S → a+T+S → a+b+S → a+b+T → a+b+a

(3) S → T+S → T+T+S → T+T+T → T+T+a → T+b+a→ a+b+a

КС-грамматика G называется **неоднозначной**,

если существует хотя бы одна цепочка $\alpha \in L(G)$, для которой может быть построено два или более различных деревьев вывода.

Это утверждение эквивалентно тому, что цепочка α имеет два или более разных левосторонних (или правосторонних) выводов.

В противном случае грамматика называется **однозначной.**

Утв. Проблема определения, является ли заданная КС-грамматика однозначной, является **алгоритмически неразрешимой**.

Язык, порождаемый грамматикой, называется **неоднозначным**, если он не может быть порожден никакой однозначной грамматикой.

Утв. Проблема определения, порождает ли данная КС-грамматика однозначный язык (т.е. существует ли эквивалентная ей однозначная грамматика), является **алгоритмически неразрешимой.**

- Пример неоднозначного языка:

$L = \{a^n b^n c^m \mid n>0, m>0\} \cup \{a^n b^m c^m \mid n>0, m>0\}$

Приведенные КС-грамматики

Символ $x \in (V_T \cup V_N)$ называется *недостижимым* в грамматике $G=(V_T, V_N, P, S)$, если он не появляется ни в одной сентенциальной форме этой грамматики.

Символ $A \in V_N$ называется *бесплодным* в грамматике $G=(V_T, V_N, P, S)$, если множество выводимых из этого символа терминальных цепочек пусто.

КС-грамматика называется приведенной, если в ней нет недостижимых и бесплодных символов.

Приведенные КС-грамматики

Алгоритм приведения грамматики:

1. Найти и удалить все бесплодные символы и правила, их содержащие.
2. Найти и удалить все недостижимые символы и правила, их содержащие.

Примечание. Если начальный символ грамматики окажется бесплодным, то следует удалить содержащие его правила, а сам символ оставить в алфавите нетерминалов V_N, так как по определению грамматики V_N обязан содержать начальный символ.

Для нахождения бесплодных и недостижимых символов полезен граф КС-грамматики:

- каждому символу из $V_T \cup V_N$ соответствует единственная вершина, помеченная этим символом; если в P есть правило с пустой правой частью ε, то граф имеет вершину, помеченную ε ;
- вершина X соединяется с вершиной Y стрелкой (дугой), если в грамматике есть правило $X \to \alpha Y \beta$, $\alpha, \beta \in (V_T \cup V_N)^*$;
- X соединяется с вершиной ε, если в грамматике есть правило $X \to \varepsilon$.

Алгоритм удаления бесплодных символов:

1. Отметить терминальные вершины (вершины, помеченные терминальными символами), а также вершину ε, если таковая имеется.

2. Если в P есть правило $A \to \alpha$, где α состоит из уже отмеченных в графе символов, а вершина A не отмечена, то отметить эту вершину. Повторять шаг 2 пока возможно.

3. Из грамматики удалить неотмеченные символы и правила, их содержащие.

Алгоритм удаления бесплодных символов:

1. Отметить терминальные вершины (вершины, помеченные терминальными символами), а также вершину ε, если такая имеется.
2. Если в P есть правило $A \to \alpha$, где α состоит из уже отмеченных в графе символов, а вершина A не отмечена, то отметить эту вершину. Повторять шаг 2 пока возможно.
3. Из грамматики удалить неотмеченные символы и правила, их содержащие.

Алгоритм удаления недостижимых символов:

1. Отметить вершины, в которые есть путь из вершины S.
2. Удалить из грамматики неотмеченные символы и правила, их содержащие.

Пример. Дана грамматика

G=({a, b, c, d, e}, {S, A, B, C, D}, P, S)

P: $S \to aAB \mid C$

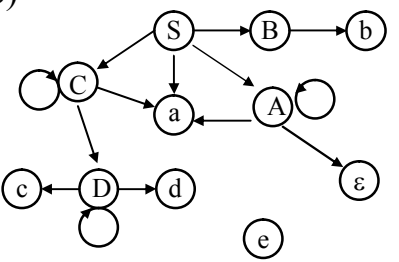
$D \to cDc \mid d$

$C \to aCD$

$A \to aA \mid a \mid \varepsilon$

$B \to b$

Граф грамматики G:



Пример. Дана грамматика

G=({a, b, c, d, e}, {S, A, B, C, D}, P, S)

P: $S \to aAB \mid C$

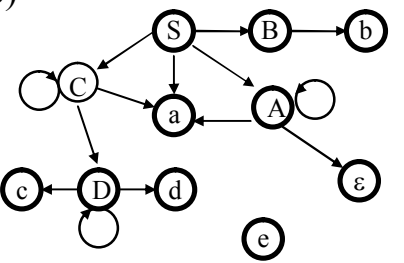
$D \to cDc \mid d$

$C \to aCD$

$A \to aA \mid a \mid \varepsilon$

$B \to b$

Граф грамматики G:



Не отмеченные жирным кружком символы бесплодны.

Пример. Дана грамматика

G=({a, b, c, d, e}, {S, A, B, ~~C~~, D}, P, S)

P: $S \to aAB \mid \sout{C}$

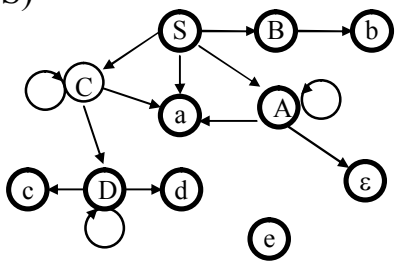
$D \to cDc \mid d$

$\sout{C \to aCD}$

$A \to aA \mid a \mid \varepsilon$

$B \to b$

Граф грамматики G:



Не отмеченные жирным кружком символы бесплодны.

Удалив из G бесплодные символы, получим эквивалентную грамматику

G_1 =({a, b, c, d, e}, {S, A, B, D}, P_1 , S)

P_1 : $S \to aAB$

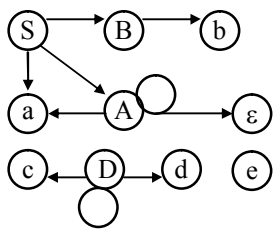
$D \to cDc \mid d$

$A \to aA \mid a \mid \varepsilon$

$B \to b$

G_1 не содержит бесплодных символов.

Граф грамматики G_1 :

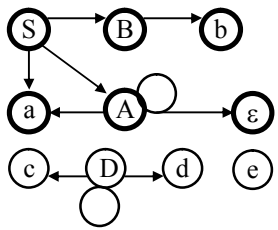


Находим недостижимые символы

$G_1 = (\{a, b, c, d, e\}, \{S, A, B, D\}, P_1, S)$

P_1 :
$S \to aAB$
$D \to cDc \mid d$
$A \to aA \mid a \mid \varepsilon$
$B \to b$

Граф грамматики G_1 :



Здесь неотмеченные символы являются недостижимыми.

$G_1 = (\{a, b, \sout{c}, \sout{d}, \sout{e}\}, \{S, A, B, \sout{D}\}, P_1, S)$

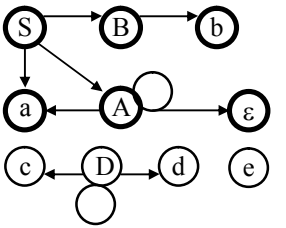
P_1 : $S \to aAB$

~~$D \to cDc \mid d$~~

$A \to aA \mid a \mid \varepsilon$

$B \to b$

Граф грамматики G_1 :



Здесь неотмеченные символы являются недостижимыми.

Удалив из G_1 недостижимые символы, получим эквивалентную грамматику:

$G_2 = (\{a, b\}, \{A, B\}, P_2, S)$

P_2 : $S \to aAB$

$A \to aA \mid a \mid \varepsilon$

$B \to b$

G_2 – приведенная грамматика

$L(G)=L(G_1)=L(G_2)=\{ a^n b \mid n \geq 1 \}$

Задача. Убедиться, что если в рассмотренном выше примере поменять местами шаги (1) и (2) алгоритма приведения грамматики, то результатом будет неприведенная грамматика.

Устранение правил с пустой правой частью из КС-грамматики

1. Построить множество $X=\{A \in N \mid A \Rightarrow \varepsilon\}$.
2. Удалить правила с пустой правой частью.
3. Если $S \in X$, то S' – новый начальный символ, $S' \to S \mid \varepsilon \in P$.
4. $\forall\ A \in X$ правило вида $B \to \alpha_1 A_1 \alpha_2 A_2 \ldots \alpha_n A_n \alpha_{n+1}$,

   где $\alpha_i \in ((N - X) \cup T)^*$

   заменить 2^n правилами, соответствующими всем возможным комбинациям вхождений A между α_i:

   $B \to \alpha_1 \alpha_2 \ldots \alpha_n \alpha_{n+1}$

   $B \to \alpha_1 \alpha_2 \ldots \alpha_n A_n \alpha_{n+1}$

   ...

   $B \to \alpha_1 \alpha_2 A_2 \ldots \alpha_n A_n \alpha_{n+1}$

   $B \to \alpha_1 A_1 \alpha_2 A_2 \ldots \alpha_n A_n \alpha_{n+1}$

   Замечание: если все $\alpha_i = \varepsilon \quad \forall\ i = 1,\ldots,n+1$, то правило $B \to \varepsilon$ не включать в новую грамматику.
5. Удалить бесполезные символы и правила, их содержащие.

Пример.	$S \to BC \mid Ab$		$S \to C \mid b \mid Ab$
исходная	$B \to \varepsilon$	эквивалентная	$C \to c$
грамматика	$C \to c$	грамматика	$A \to Aa \mid a$
	$A \to Aa \mid \varepsilon$		

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.

- 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.

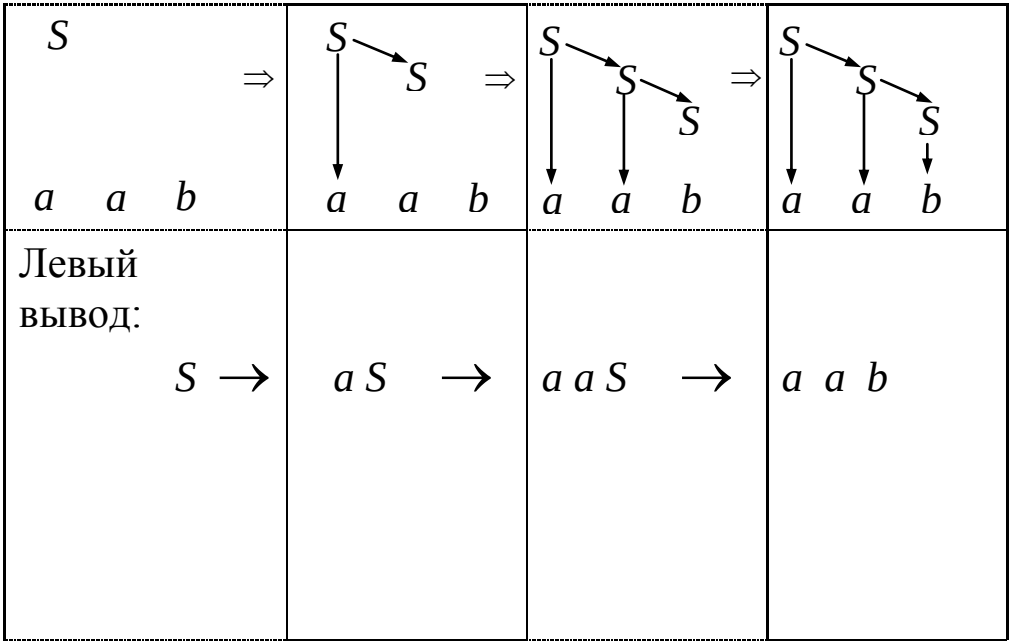
Для КС-языков и регулярных языков существуют эффективные алгоритмы разбора.

Регулярные и КС-языки используются при описании синтаксиса языков программирования

Построение дерева вывода

G: $S \to a\,S \mid b$ Цепочка: aab

Сверху вниз:



Левый вывод:

$S \to$ $a\,S \to$ $a\,a\,S \to$ $a\,a\,b$

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 $\leftarrow$.

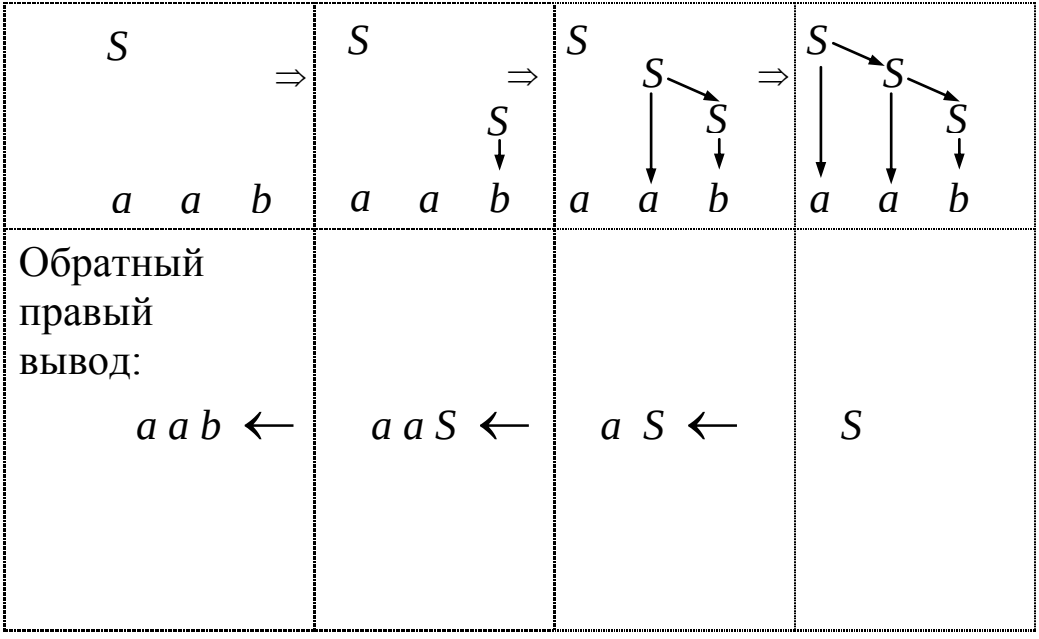
С помощью сверток можно построить вывод «задом наперед» (обращение вывода): от цепочки к цели грамматики S. Например, сентенциальную форму aaS можно свернуть к aS, а затем к S :

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

Построение дерева вывода

G: $S \to a\,S \mid b$ Цепочка: aab

Снизу вверх:



РЕГУЛЯРНЫЕ ЯЗЫКИ

способы описания:

-- регулярные грамматики

(леволинейные либо праволинейные)

-- конечные автоматы

(недетерминированные или детерминированные)

-- регулярные выражения

(см. материал “О регулярных языках” на cmcmsu.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.

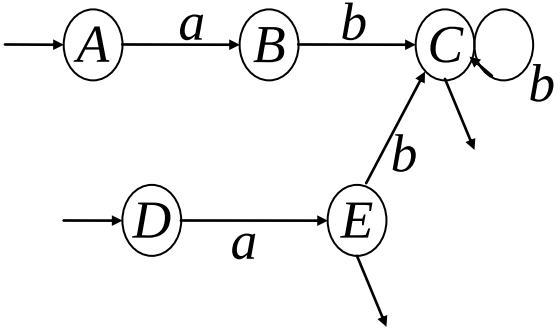
Путь в ориентированном графе может быть представлен последовательностью дуг. *Пустой* путь можно представить одной вершиной, которая считается одновременно началом и концом пути.

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.

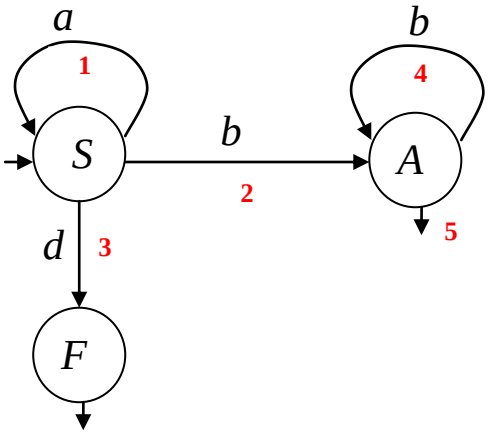
$$L(\mathcal{A}_1) = \{ab^n \mid n \geq 0\}$$

Алгоритм построения НКА по праволинейной автоматной грамматике

1. Множество вершин НКА состоит из нетерминалов грамматики и, возможно, еще одной новой вершины F, которая объявляется заключительной.
2. Каждому правилу вида $A \to aB$ в автомате соответствует дуга из вершины A в вершину B, помеченная символом a: $A \xrightarrow{a} B$. Каждому правилу вида $A \to a$ соответствует дуга $A \xrightarrow{a} F$. Других дуг нет.
3. Начальной вершиной автомата является вершина, соответствующая начальному символу грамматики. Заключительными являются новая вершина F, если она использовалась на шаге 2, и каждая вершина A, такая что для нетерминала A в грамматике есть правило $A \to \varepsilon$.

Пример (НКА по праволинейной гр-ке)

1. $S \to aS$
2. $S \to bA$
3. $S \to d$
4. $A \to bA$
5. $A \to \varepsilon$



$S \to_1 aS \to_2 abA \to_4 abbA \to_5 abb$ – вывод в грамматике для abb

$\to S \xrightarrow{a}_1 S \xrightarrow{b}_2 A \xrightarrow{b}_4 A \to_5$ – соответствующий путь в автомате для abb

Алгоритм построения праволинейной автоматной грамматики по НКА с единственной начальной вершиной

1. Нетерминалами грамматики будут вершины автомата, терминалами — пометки дуг.
2. Для каждой дуги $A \xrightarrow{a} B$ в грамматику добавляется правило $A \to aB$. Для каждой заключительной вершины B в грамматику добавляется правило $B \to \varepsilon$.
3. Начальным символом будет нетерминал, соответствующий начальной вершине.

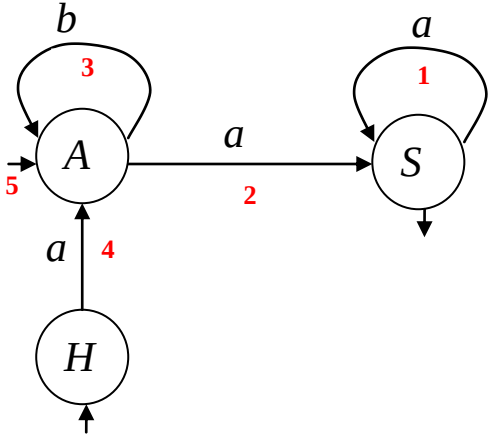
(4.) К построенной по пунктам 1–3 грамматике можно применить алгоритм устранения ε-правил.

Алгоритм построения НКА по леволинейной автоматной грамматике

1. Множество вершин НКА состоит из нетерминалов грамматики и, возможно, 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. Начальными являются вершина H, если она использовалась на шаге 2, и каждая вершина A, такая что для нетерминала A в грамматике есть правило $A \to \varepsilon$.

Пример (НКА по леволинейной гр-ке)

1. $S \to Sa$
2. $S \to Aa$
3. $A \to Ab$
4. $A \to a$
5. $A \to \varepsilon$



$aba \leftarrow_4 Aba \leftarrow_3 Aa \leftarrow_2 S$ – обращение вывода в грамматике для aba

(свертки)

$\to H \xrightarrow{a}_4 A \xrightarrow{b}_3 A \xrightarrow{a}_2 S \to$ – соответствующий путь в автомате для aba

(моделирует свертки)

Алгоритм построения леволинейной автоматной 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 можно применить алгоритм устранения ε-правил.

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

P: $S \to C\perp$

$C \to Ab \mid Ba$

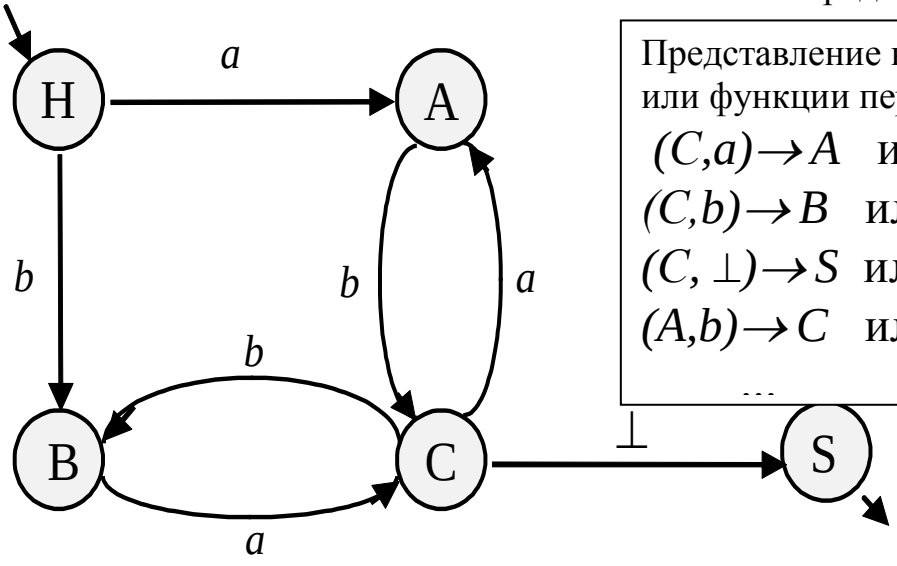
$A \to a \mid Ca$

$B \to b \mid Cb$

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

	a	b	$\perp$
H	A	B	-
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 $\delta(A,B) = \{C\}$

... ...

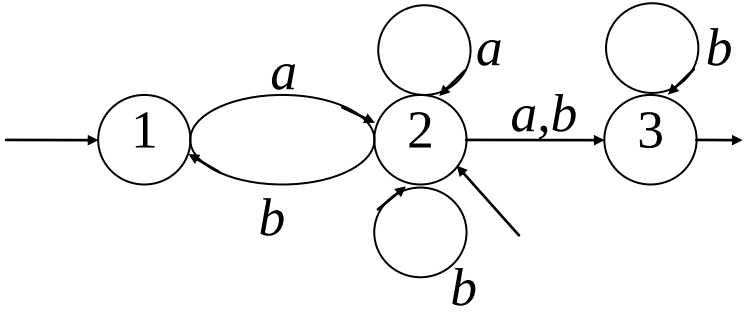
Конечный автомат называется **детерминированным** конечным автоматом (ДКА), если он имеет единственное начальное состояние, и любые две дуги, исходящие из одной и той же вершины имеют различные пометки.

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.

Пример. $\mathcal{A}'$:



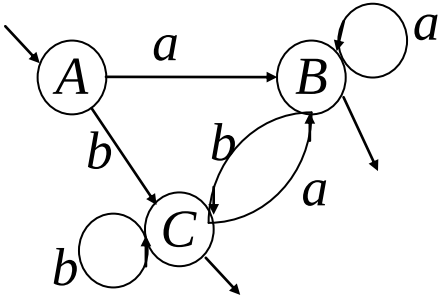
Процесс построения ДКА по заданному НКА удобно изобразить в виде таблицы, начав с состояния {1, 2}. Затем заполняем строки для вновь появляющихся состояний.

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

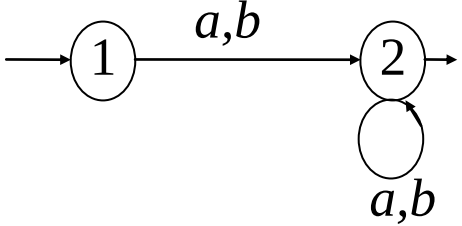
С учетом переобозначений построим по таблице ДКА $\mathcal{A}$:



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

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

$\mathcal{A}''$:



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

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.

(Получаемый ДКА будет всюду определенным, он умеет дочитать до конца любую цепочку. Тупиковое состояние ∅ (состояние ошибки) и связанные с ним дуги при необходимости можно удалить).

Для более удобной работы с диаграммами состояний пользуемся следующими соглашениями:

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) для любого автомата сущесвтует состояние ошибки (ERR); переход в это состояние означает, 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.

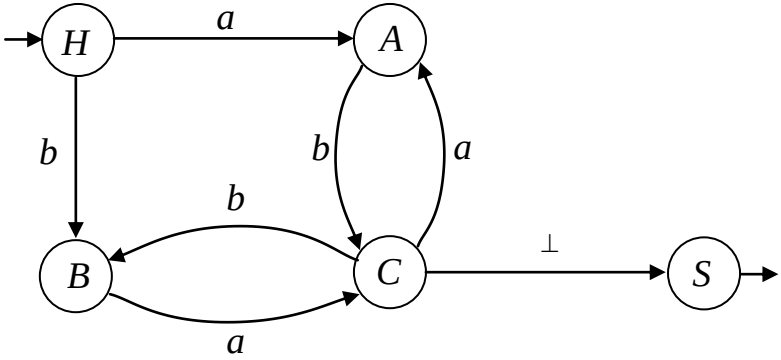
Выход: «Да», если $x \in L(\mathcal{A})$, иначе — «Нет».

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

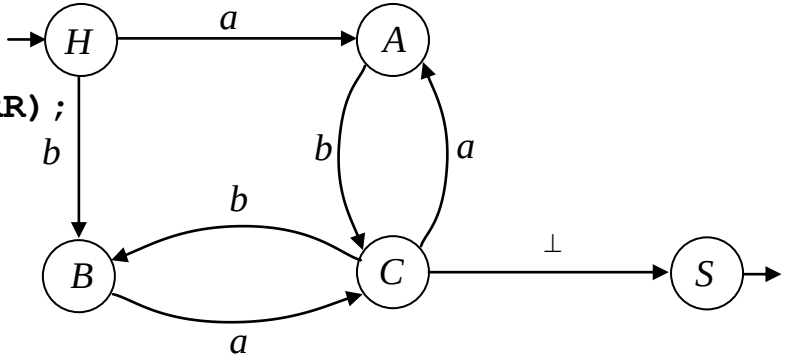
P: $S \to C\perp$
$C \to Ab \mid Ba$
$A \to a \mid Ca$
$B \to b \mid Cb$



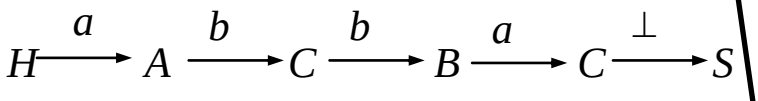
Программа-анализатор на C++ :

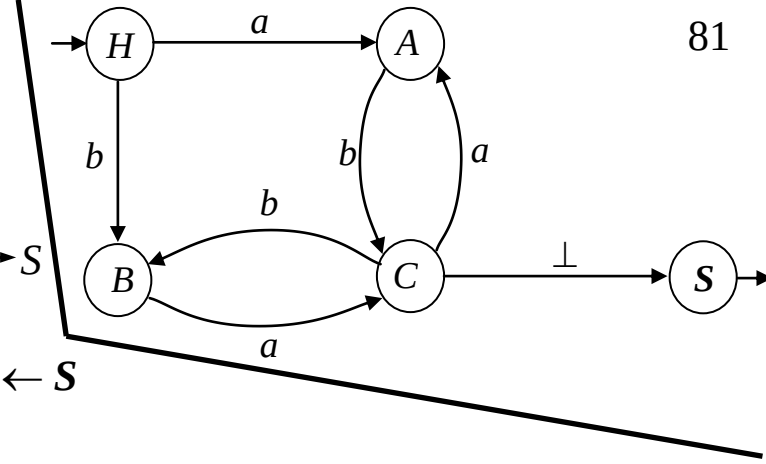
```
#include <iostream>
char c; //текущий символ

void gc () {std::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;  // начинаем с начального состояния H

gc(); // считываем 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: if (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;

}
```

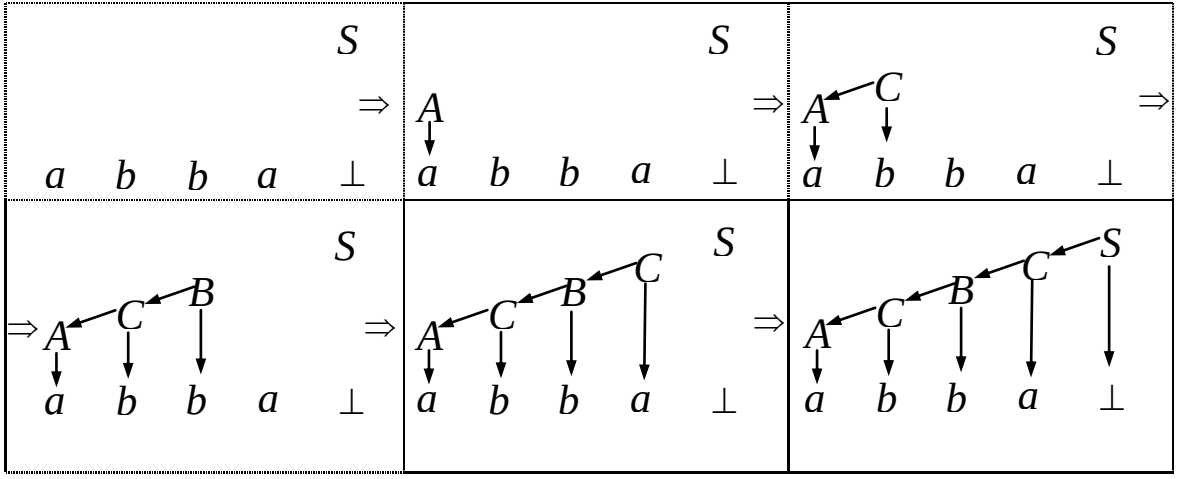


Пример разбора цепочки $abba\perp$





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



Недетерминированный разбор

Если конечный автомат, построенный по грамматике, не является детерминированным, 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.info/download/formal.grammars.and.languages.2009.pdf)

Конечные автоматы (диаграммы состояний) с действиями

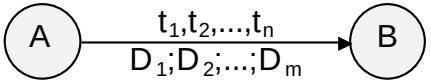
идентификатор (I):

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

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:

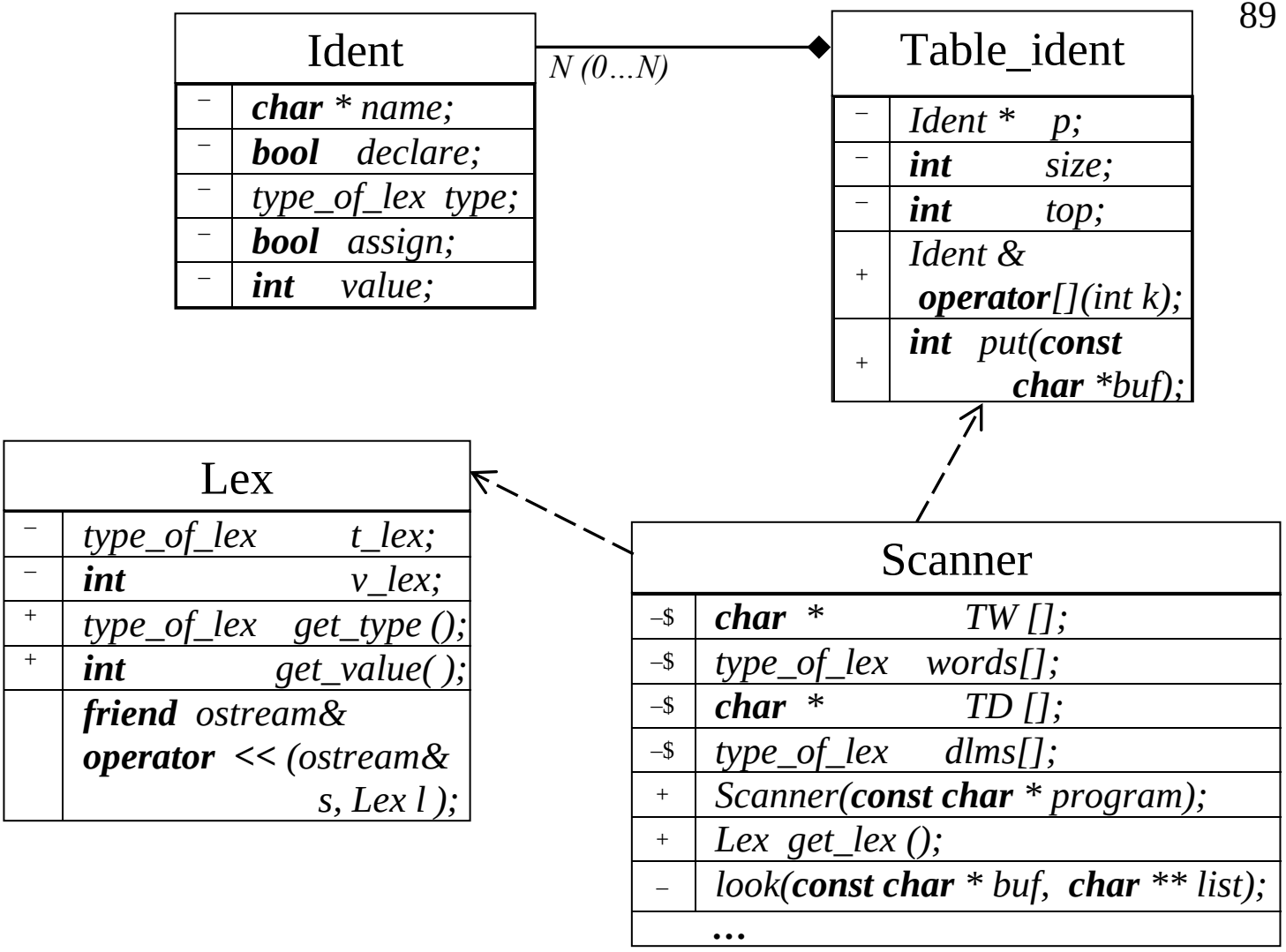
```
enum  type_of_lex {LEX_NULL, /*0*/
                   LEX_AND, LEX_BEGIN, … LEX_WRITE, /*18*/
                   LEX_FIN, /*19*/
                   LEX_SEMICOLON, LEX_COMMA, … LEX_GEQ, /*35*/
                   LEX_NUM, /*36*/
                   LEX_ID, /*37*/
                   POLIZ_LABEL, /*38*/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

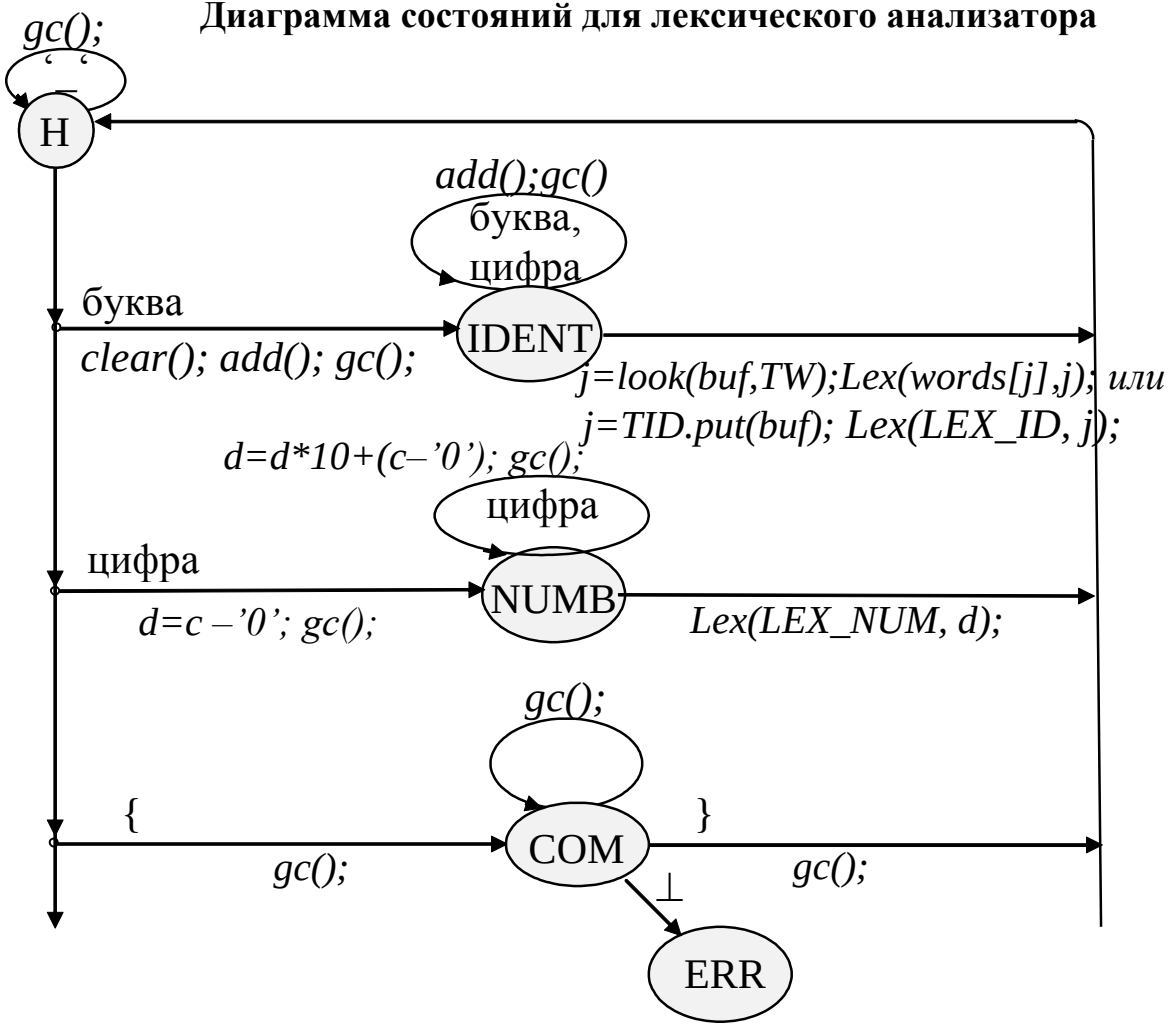
```
    void  add () { buf [ buf_top ++ ] = c; }
    int   look (const char *buf, char **list) {
            int i = 0;
            while (list[i]) {
              if (!strcmp(buf, list[i])) return i;
              i++;
            }
            return 0;
    }
    void  gc () { c = fgetc (fp); }

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

Таблицы лексем М-языка:

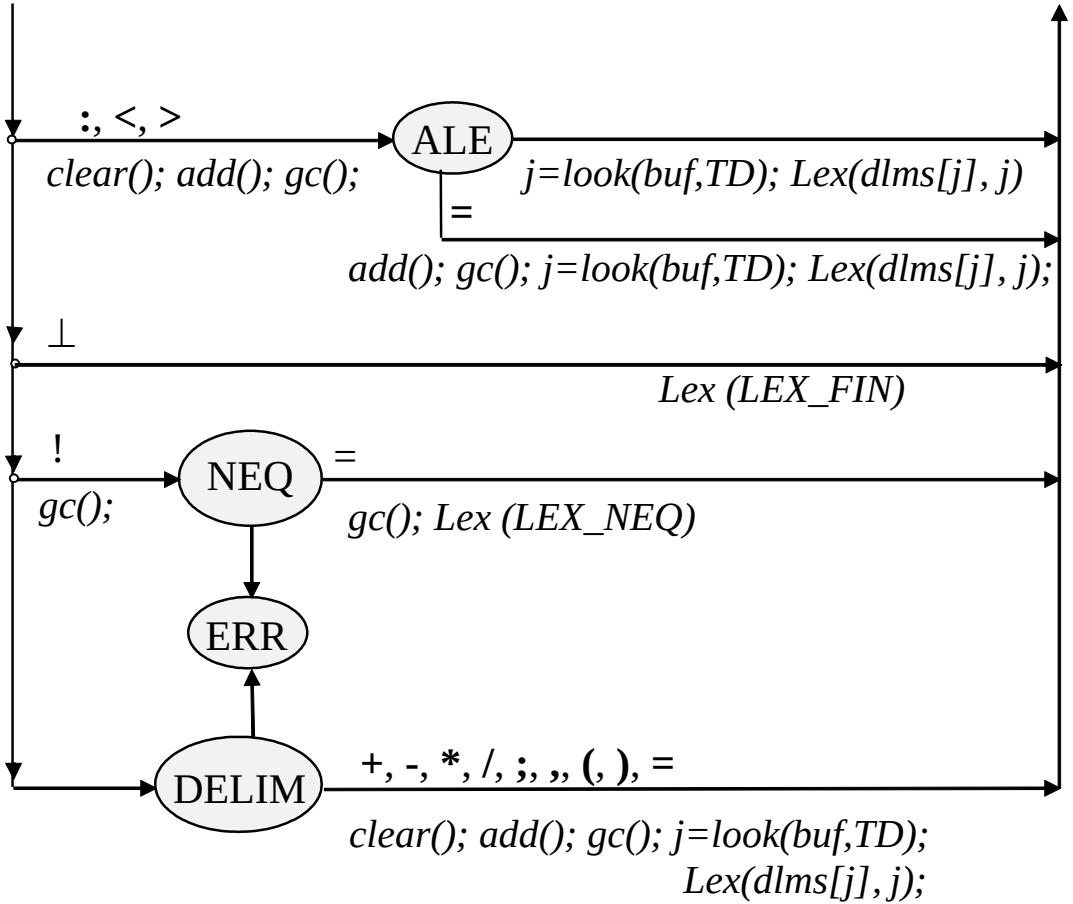
```
char * Scanner:: TW [] = {

    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()
```

Задача разбора (синтаксический анализ)

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).

Существуют различные методы синтаксического анализа для КС-грамматик;

для некоторых подклассов есть эффективные методы, затрачивающие линейное время Cn на анализ цепочки длины n.

Каждый метод синтаксического анализа предполагает свой способ построения по грамматике программы-анализатора, которая будет осуществлять разбор цепочек.

В основе анализатора может быть автомат с магазинной памятью. Мы рассмотрим другой способ – метод рекурсивного спуска (система рекурсивных процедур).

Анализатор некорректен, если:

-- не распознает хотя бы одну цепочку, принадлежащую языку;

-- распознает хотя бы одну цепочку, языку не принадлежащую;

-- зацикливается на какой-либо цепочке.

Метод анализа *примен**и**м* к данной грамматике, если анализатор, построенный в соответствии с этим методом, корректен.

Метод рекурсивного спуска (РС-метод)

Пример: *пусть дана грамматика* $G = (\{a,b,c,d\}, \{S,A,B\}, P, S)$, *где*

$$P: \quad S \to ABd$$
$$A \to a \mid cA$$
$$B \to bA$$

и надо определить, принадлежит ли цепочка cabad языку L(G).

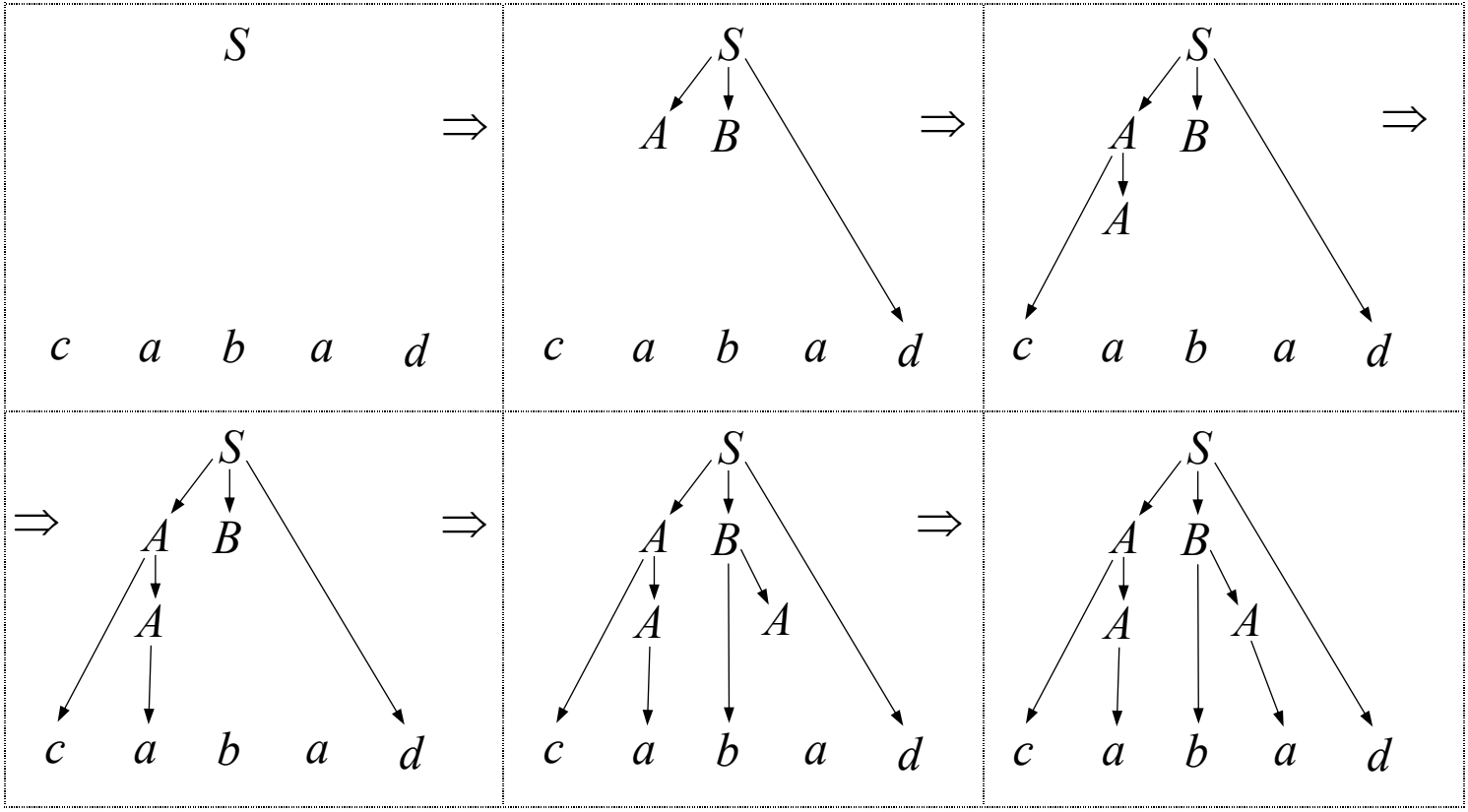
Построим левый вывод этой цепочки:

$S \to ABd \to cABd \to caBd \to cabAd \to cabad$

Следовательно, цепочка принадлежит языку *L(G)*.

$$S \to ABd \to cABd \to caBd \to cabAd \to cabad$$

Построение левого вывода эквивалентно построению дерева вывода методом «сверху вниз» (нисходящим методом) :



Метод рекурсивного спуска (РС-метод):

Для **каждого нетерминала** грамматики создается своя процедура с **именем** этого нетерминала; ее задача — начиная с указанного места исходной цепочки найти подцепочку, которая выводится из этого нетерминала.

Если подцепочку удалось найти, то работа процедуры считается нормально завершенной и осуществляется возврат в точку вызова, иначе — разбор прекращается и сообщается об ошибке, цепочка не принадлежит языку.

Тело каждой такой процедуры пишется непосредственно по правилам вывода соответствующего нетерминала: терминалы из правой части распознаются самой процедурой, а нетерминалы соответствуют вызовам процедур, носящих их имена.

Программа – анализатор для G_1

G_1:

$$S \rightarrow ABd$$
$$A \rightarrow a \mid cA$$
$$B \rightarrow bA$$

```
#include <iostream>
using namespace std;
int  c;     // текущий символ
void A ();
void B ();
void gc ()
{
   cin >> c; // считать символ (лексему) из входного потока
}

void S ()
{
   cout << "S-->ABd, ";      // применяемое правило вывода
   A();
   B();
   if ( c == 'd' ) gc();
   else  throw c;
}
```

```
void A ()
{
   if ( c =='a' )
   {
      cout << "A-->a, ";
      gc ();
   }
   else if ( c =='c' )
   {
      cout << "A-->cA, ";
      gc ();
      A ();
   }
   else
      throw c;
}
```

G_1:

$$S \rightarrow ABd$$
$$A \rightarrow a \mid cA$$
$$B \rightarrow bA$$

```
void B ()
{
   if ( c =='b' )
   {
      cout << "B-->bA, ";
      gc ();
      A ();
   }
   else
      throw c;
}
```

G_1:

$$S \rightarrow ABd$$
$$A \rightarrow a \mid cA$$
$$B \rightarrow bA$$

```
int main ()
{
   try
   {
      gc ();
      S ();
      if ( c != '⊥' ) // проверяем, что достигнут конец
                      //  цепочки
         throw c;
      cout << "SUCCESS !!!" << endl;
      return 0;
   }
   catch ( int c )
   {
      cout << "ERROR on lexeme" << c << endl;
      return 1;
   }
}
```

G_1:

$$S \rightarrow ABd$$
$$A \rightarrow a \mid cA$$
$$B \rightarrow bA$$

Достаточное условие применимости метода рекурсивного спуска

Для применимости метода рекурсивного спуска достаточно, чтобы каждое правило в грамматике имело вид:

(а) либо $X \to \alpha$,

где $\alpha \in (T \cup N)^*$ и это единственное правило вывода для этого нетерминала;

(б) либо $X \to a_1\alpha_1 \mid a_2\alpha_2 \mid \ldots \mid a_n\alpha_n$,

где $a_i \in T$ для всех $i = 1, 2, \ldots, n$; $a_i \neq a_j$ для $i \neq j$; $\alpha_i \in (T \cup N)^*$, т. е. если для нетерминала X правил вывода несколько, то они должны начинаться с терминалов, причем все эти терминалы должны быть различными;

Это условие не является необходимым.

Найдем критерий применимости

РС-метод примени́м, если и только если левый вывод (или дерево нисходящим способом) можно построить, начиная с начального символа S, так, что на каждом шаге вывода решение о том, какое правило (альтернативу) применять для замены левого нетерминала, безошибочно принимается по первому символу из непрочитанной части входной цепочки (т. е. по «текущему» символу).

Рассмотрим примеры

G_2:

$S \rightarrow aA \mid B \mid d$

$A \rightarrow d \mid aA$

$B \rightarrow aA \mid a$

G_2 неоднозначна, РС-метод неприменим.

нельзя дать однозначный прогноз, что делать на первом шаге при анализе цепочки, начинающейся с символа a (т. е. по текущему символу a невозможно сделать однозначный выбор: $S \rightarrow aA$ или $S \rightarrow B$)

G_3 однозначна, но РС-метод неприменим

G_3:

$S \to A \mid B$

$A \to aA \mid d$

$B \to aB \mid b$

Определение: *множество first* (α) — это множество терминальных символов, которыми начинаются цепочки, выводимые из цепочки α в грамматике $G = \langle T, N, P, S \rangle$, т. е.

$$first(\alpha) = \{ a \in T \mid \alpha \Rightarrow a\alpha', \text{ где } \alpha \in (T \cup N)^*,\ \alpha' \in (T \cup N)^* \}.$$

Например: $first(A) = \{ a, d \}$, $first(B) = \{ a, b \}$. Пересечение этих множеств непусто: $first(A) \cap first(B) = \{ a \} \neq \varnothing$, и поэтому метод рекурсивного спуска к G_3 неприменим.

Итак, **наличие в грамматике правил вида** $X \to \alpha \mid \beta$**, таких что** $first(\alpha) \cap first(\beta) \neq \varnothing$**, делает метод рекурсивного спуска неприменимым**.

Рассмотрим еще несколько примеров.

G_4:

$S \to aA \mid BDc$

$A \to BAa \mid aB \mid b$

$B \to \varepsilon$

$D \to B \mid b$

$first(aA) = \{ a \}$, $first(BDc) = \{ b, c \}$;

$first(BAa) = \{ a, b \}$, $first(aB) = \{ a \}$,

$first(b) = \{ b \}$;

$first(\varepsilon) = \varnothing$;

$first(B) = \varnothing$, $first(b) = \{ b \}$.

Метод рекурсивного спуска неприменим к грамматике G_4, так как $first(BAa) \cap first(aB) = \{ a \} \neq \varnothing$.

G_5:

$S \rightarrow aA$

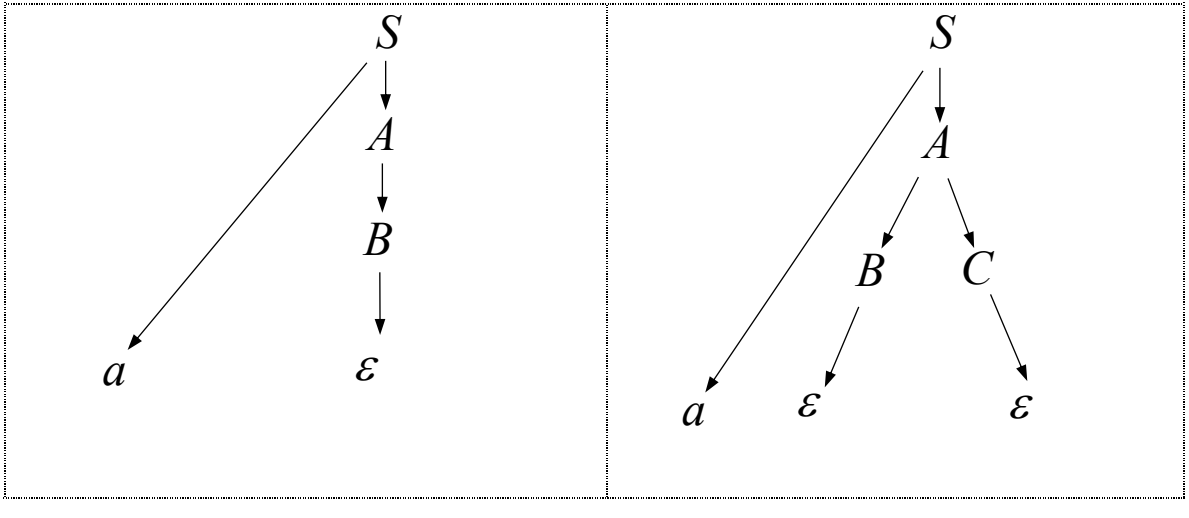
$A \rightarrow BC \mid B$

$C \rightarrow b \mid \varepsilon$

$B \rightarrow \varepsilon$

Пересечение множеств *first* пусто, но РС-метод неприменим.

Действительно, $BC \Rightarrow \varepsilon$ и $B \Rightarrow \varepsilon$. Цепочка a имеет два различных дерева вывода



Таким образом, **если в грамматике для двух различных правил $X \to \alpha \mid \beta$ выполняются соотношения $\alpha \Rightarrow \varepsilon$ и $\beta \Rightarrow \varepsilon$, то метод рекурсивного спуска неприменим**.

Рассмотрим примеры с единственной альтернативой, из которой выводится ε.

G_6:

$S \to cAd \mid d$

$A \to aA \mid \varepsilon$

Метод применим: если текущий символ a, то выбираем альтернативу $A \to aA$ иначе $A \to \varepsilon$ —

G_7:

$S \to Bd$

$B \to cAa \mid a$

$A \to aA \mid \varepsilon$

Неприменим, т.к. для A невозможно правильно выбрать альтернативу без «заглядывания» на символ вперед.

Определение: *множество follow(A)* — это множество терминальных символов, которые могут появляться в сентенциальных формах грамматики непосредственно справа от A, т. е.

$$follow(A) = \{\, a \in T \mid S \Rightarrow \alpha A \beta,\ \beta \Rightarrow a\,\gamma,\ A \in N,\ \alpha, \beta, \gamma \in (T \cup N)^*\}$$

Тогда, **если в грамматике есть пара правил $X \to \alpha \mid \beta$, таких что $\beta \Rightarrow \varepsilon$, $first(X) \cap follow(X) \neq \varnothing$, то метод рекурсивного спуска неприменим к данной грамматике.**

Утверждение. Пусть G — КС-грамматика. Метод рекурсивного спуска применим к G, если и только если для любой пары альтернатив вида $X \to \alpha \mid \beta$ выполняются следующие условия:

(1) $first(\alpha) \cap first(\beta) = \varnothing$;

(2) справедливо не более чем одно из двух соотношений:

$\alpha \Rightarrow \varepsilon$, $\beta \Rightarrow \varepsilon$;

(3) если $\beta \Rightarrow \varepsilon$, то $first(X) \cap follow(X) = \varnothing$.

Канонический вид для РС-метода

(1) либо $X \to \alpha$,
где $\alpha \in (T \cup N)^*$ и это единственное правило вывода для этого нетерминала;

(2) либо $X \to a_1\alpha_1 \mid a_2\alpha_2 \mid \ldots \mid a_n\alpha_n$,
где $a_i \in T$ для всех $i = 1, 2, \ldots, n$; $a_i \neq a_j$ для $i \neq j$;
$\alpha_i \in (T \cup N)^*$, т. е. если для нетерминала X правил вывода несколько, то они должны начинаться с терминалов, причем все эти терминалы должны быть попарно различными;

(3) либо $X \to a_1\alpha_1 \mid a_2\alpha_2 \mid \ldots \mid a_n\alpha_n \mid \varepsilon$,
где $a_i \in T$ для всех $i = 1, 2, \ldots, n$; $a_i \neq a_j$ для $i \neq j$;
$\alpha_i \in (T \cup N)^*$, и $first(X) \cap follow(X) = \varnothing$.

Этот вид удобен для построения рекурсивных процедур, но он дает только достаточное условие применимости.

Вопрос*: если грамматика не удовлетворяет критерию применимости РС-метода, то существует ли эквивалентная КС-грамматика, для которой метод рекурсивного спуска применим?*

К сожалению, нет алгоритма, отвечающего на этот вопрос для произвольной КС-грамматики, т.е. это ***алгоритмически неразрешимая проблема.***

Модификация метода для грамматик с итерациями:
Для правил вида $L \to a\ \{,a\}$
(*эквивалент* $L \to a \mid a, L$)

рекурсию заменяем итерацией:

```
void L()
{ if (c != 'a') throw c;
  gc();
 while (c == ',')
  {gc(); if (c != 'a') throw c; else
                                  gc();}
}
```

Важно, чтобы в любой сентенциальной форме после *L* не было запятой, иначе *L* прочитает «не свою» запятую. (Вместо запятой в данном примере может быть любой другой символ.)

Пример, когда анализатор по вышеприведенной схеме не дает корректный ответ:

G:

$S \to LB\perp$

$L \to a\ \{, a\}$

$B \to ,b$

Если для этой грамматики написать анализатор, действующий РС-методом, то цепочка *a,a,a,b* будет признана им ошибочной, хотя $a,a,a,b \in L(G)$.

В языках программирования после повторяющихся конструкций обычно идет какой-нибудь новый символ, так что подобных проблем не возникает:

var *a,b,c,d : integer;* или ***int*** *a,b,c,d;*

Если грамматику переписать без итерации { }

$S \rightarrow LB\perp$

$L \rightarrow a\ M$

$M \rightarrow , a\ M \mid \varepsilon$

$B \rightarrow , b$

то нетрудно видеть, что $first\ (,\ a) \cap follow\ (M) = \{\ ,\ \} \neq \varnothing$ и поэтому метод рекурсивного спуска неприменим.

Эквивалентные преобразования для КС-грамматик, которые могут помочь перестроить исходную грамматику так, чтобы РС-метод был примени́м.

1) Если в грамматике есть нетерминалы, правила вывода которых **леворекурсивны**, т.е. имеют вид

$$A \to A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m,$$

где $\alpha_i \in (V_T \cup V_N)^+$, $\beta_j \in (V_T \cup V_N)^*$, $i = 1, 2, \ldots, n$; $j = 1, 2, \ldots, m$,

то левую рекурсию всегда можно заменить правой:

$$A \to \beta_1 A' \mid \ldots \mid \beta_m A'$$
$$A' \to \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$$

Будет получена грамматика, эквивалентная исходной.

2) Если в грамматике есть нетерминал, у которого несколько правил вывода начинаются **одинаковыми терминальными символами**, т.е. имеют вид

$$A \to a\alpha_1 \mid a\alpha_2 \mid \ldots \mid a\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m,$$
$$\text{где } a \in V_T; \quad \alpha_i, \beta_j \in (V_T \cup V_N)^*,$$

то можно преобразовать правила вывода данного нетерминала, объединив правила с общими началами в одно правило:

$$A \to aA' \mid \beta_1 \mid \ldots \mid \beta_m$$
$$A' \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

Будет получена грамматика, эквивалентная исходной.

3) Если в грамматике есть нетерминал, у которого **несколько** правил вывода, и среди них есть правила, **начинающиеся нетерминальными символами**, т.е. имеют вид

$$A \to B_1\alpha_1 \mid \ldots \mid B_n\alpha_n \mid a_1\beta_1 \mid \ldots \mid a_m\beta_m$$
$$B_1 \to \gamma_{11} \mid \ldots \mid \gamma_{1k}$$
$$\ldots$$
$$B_n \to \gamma_{n1} \mid \ldots \mid \gamma_{np}, \quad \text{где} \quad B_i \in N; \quad a_j \in T; \quad \alpha_i, \beta_j, \gamma_{ij} \in (T \cup N)^*,$$

то можно заменить вхождения нетерминалов B_i их правыми частями из правил вывода:

$$A \to \gamma_{11}\alpha_1 \mid \ldots \mid \gamma_{1k}\alpha_1 \mid \ldots \mid \gamma_{n1}\alpha_n \mid \ldots \mid \gamma_{np}\alpha_n \mid a_1\beta_1 \mid \ldots \mid a_m\beta_m$$

Будет получена грамматика, эквивалентная исходной.

4) Если в грамматике есть правила

$A \to \alpha_1 A \mid \ldots \mid \alpha_n A \mid \beta_1 \mid \ldots \mid \beta_m \mid \varepsilon$

$B \to \alpha A \beta$

и $FIRST(A) \cap FOLLOW(A) \neq \varnothing$ (из-за вхождения A в правило вывода для B), то можно преобразовать их в такие:

$B \to \alpha A'$

$A' \to \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \beta_1 \beta \mid \ldots \mid \beta_m \beta \mid \beta$

Полученная грамматика будет эквивалентна исходной.

Задача разбора (синтаксический анализ) для неоднозначных грамматик

две постановки задачи:

(1) *Даны КС-грамматика G и цепочка x. Требуется проверить: $x \in L(G)$? Если да, то построить все деревья вывода для x (или все левые выводы для x, или все правые выводы для x)*

Для решения этой задачи можно обобщить метод рекурсивного спуска, чтобы он работал с возвратами, пробуя различные подходящие альтернативы.

(2) *Даны КС-грамматика G и цепочка x. Требуется проверить: $x \in L(G)$? Если да, то построить одно дерево вывода для x (возможно, «наиболее подходящее» в некотором смысле).*

Рассмотрим пример. Грамматика неоднозначна. РС-метод неприменим.

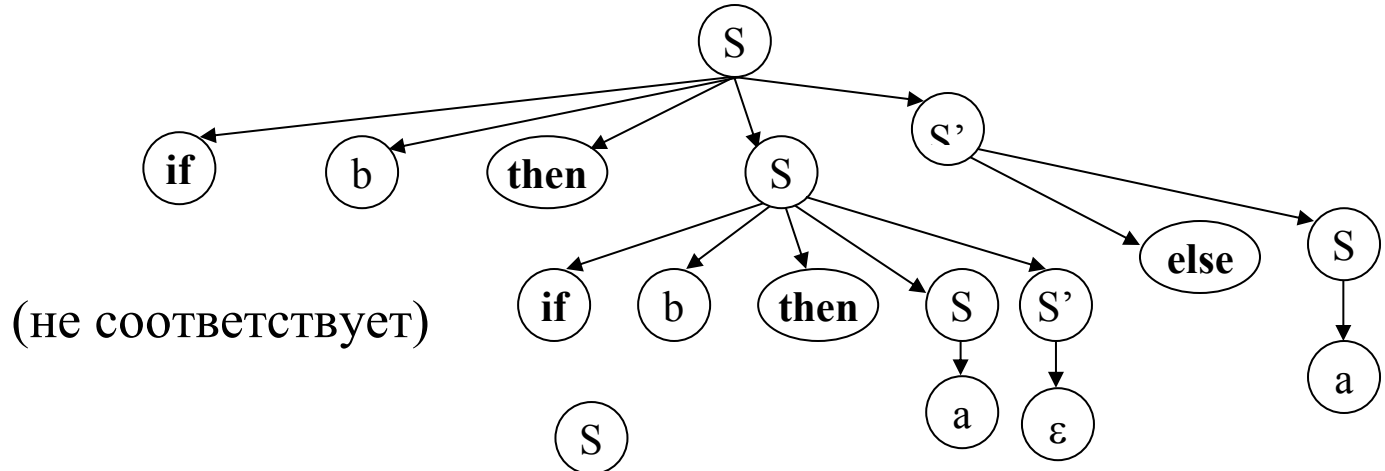
$G = (\{\mathbf{if}, \mathbf{then}, \mathbf{else}, a, b\}, \{S\}, P, S, S')$,

где $P = \{S \to \mathbf{if}\ b\ \mathbf{then}\ S\ S' \mid a;$

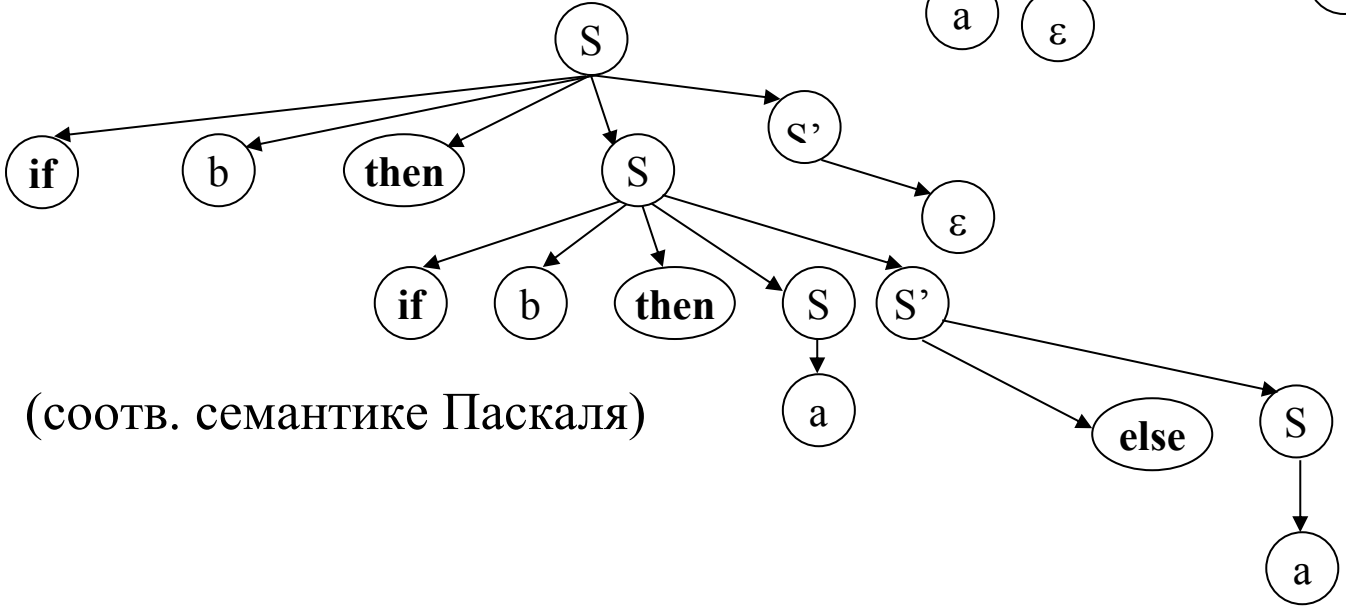
$S' \to \mathbf{else}\ S \mid \varepsilon\ \}.$

В этой грамматике для цепочки **if** *b* **then if** *b* **then** *a* **else** *a* можно построить два различных дерева вывода:

Одно из них соответствует семантике Паскаля: **else** относится к ближайшему i**f**. Такое дерево можно получить, написав РС-процедуры, где S’ по возможности отдает предпочтение непустой альтернативе.



(не соответствует)



(соотв. семантике Паскаля)

Синтаксический анализатор для М-языка

Будем считать, что синтаксический и лексический анализаторы взаимодействуют следующим образом: анализ исходной программы идет под управлением синтаксического анализатора; если для продолжения анализа ему нужна очередная лексема, то он запрашивает ее у лексического анализатора; тот выдает одну лексему и "замирает" до тех пор, пока синтаксический анализатор не запросит следующую лексему.

Соглашения:

- лексический анализатор — это функция-член класса Scanner — get_lex (), которая в качестве результата выдает лексемы типа (class) Lex;
- в переменной Lex curr_lex будем хранить текущую лексему, выданную лексическим анализатором, (а в переменной c_val – ее значение, в c_type – ее тип, это пригодится на этапе семантического анализа.)

Грамматика 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

```
class Parser {
        Lex curr_lex;
        type_of_lex c_type;
        int c_val;
        Scanner scan;
        // Stack < int, 100 > st_int;
        // Stack < type_of_lex, 100 >  st_lex;
        void P(); void D1(); void D(); void B();void S();
        void E();  void E1();  void T();  void F();

         void gl (){
            curr_lex = scan.get_lex();
            c_type = curr_lex.get_type();
          // c_val = curr_lex.get_value();
        }
  public:
    // Poliz  prog;
Parser (const char *program):scan(program)//,prog (1000)
    {}
    void analyze();
};
```

```
void Parser::analyze () {
        gl();
        P();
    // prog.print();
        cout << endl << "OK" << endl;
}


void Parser::P () {
    if (c_type == LEX_PROGRAM) gl();
    else throw curr_lex;
    D1();
    if (c_type == LEX_SEMICOLON) gl();
    else throw curr_lex;
    B();
    if (c_type != LEX_FIN) throw curr_lex;
}
```

```
void Parser::D1 () {
    if (c_type == LEX_VAR){
        gl();
        D();
        while (c_type == LEX_COMMA){ gl(); D();}
    }
    else throw curr_lex;
}
...
```

Семантический анализ

Контекстно-свободные грамматики, с помощью которых описывают синтаксис языков программирования, не позволяют задавать контекстные условия, имеющиеся в любом языке.

Примеры наиболее часто встречающихся контекстных условий:

(а) каждый используемый в программе идентификатор должен быть описан, но не более одного раза в одной зоне описания;

(б) при вызове функции число фактических параметров и их типы должны соответствовать числу и типам формальных параметров;

(в) обычно в языке накладываются ограничения на типы операндов любой операции, определенной в этом языке, на типы левой и правой частей в операторе присваивания, на тип параметра цикла, на тип условия в операторах цикла и условном операторе и т.п.

Семантический анализатор для М-языка

Контекстные условия, выполнение которых надо контролировать в программах на М-языке:

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 правилам (как в Паскале).

Проверку контекстных условий совместим с синтаксическим анализом. Для этого в синтаксические правила вставим вызовы процедур, осуществляющих необходимый контроль, а затем перенесем их в процедуры рекурсивного спуска.

Обработка описаний

Лексический анализатор запомнил в таблице идентификаторов *TID* все идентификаторы-лексемы, которые были обнаружены в тексте исходной программы. Информация о типе переменных и о наличии их описания заносится в ту же таблицу.

i-ая строка таблицы *TID* соответствует идентификатору-лексеме вида (*LEX_ID*, *i*).

Лексический анализатор заполнил поле name; значения полей declare и type заполняются на этапе семантического анализа.

Раздел описаний имеет вид:

$D \rightarrow I \{,I\}: [int \mid bool]$,

т.е. имени типа (*int* или *bool*) предшествует список идентификаторов. Эти идентификаторы (вернее, номера соответствующих им строк таблицы *TID*) надо запоминать (мы будем их запоминать в стеке целых чисел ***Stack<int,100> st_int***), а когда будет проанализировано имя типа, надо заполнить поля declare и type в этих строках.

Функция ***void Parser::dec (type_of_lex type)*** считывает из стека номера строк таблицы *TID*, заносит в них информацию о типе соответствующих переменных, о наличии их описаний и контролирует повторное описание переменных.


```
void Parser::dec ( type_of_lex type ) {
    int i;
    while ( !st_int.is_empty()) {
        i = st_int.pop();
        if ( TID[i].get_declare() ) throw "twice";
        else {
            TID[i].put_declare();
            TID[i].put_type(type);
        }
    }
}
```

С учетом имеющихся функций правило вывода с действиями для обработки описаний будет таким:

$D \rightarrow \langle st\_int.reset() \rangle\ I < st\_int.push\ (c\_val) >$
$\{, I < st\_int.push\ (c\_val) >\}:$
$[\ int < dec\ (LEX\_INT) > \mid bool < dec\ (LEX\_BOOL) >\ ]$

Контроль контекстных условий в выражении

Типы операндов и обозначение операций будем хранить в стеке ***Stack<type_of_lex, 100> st_lex***.

Если в выражении встречается лексема-целое_число или логические константы ***true*** или ***false***, то соответствующий тип сразу заносится в стек.

Если операнд — лексема-переменная, то необходимо проверить, описана ли она; если описана, то ее тип надо занести в стек. Эти действия можно выполнить с помощью функции check_id:

```
void parser::check_id(){
    if(TID[c_val].get_declare())
        st_lex.push(TID[c_val].get_type());
    else throw "not declared";
}
```

Для контроля контекстных условий каждой тройки — "операнд-операция-операнд" (для проверки соответствия типов операндов данной двуместной операции) будем использовать функцию *check_op*:


```
void Parser::check_op () {
    type_of_lex t1, t2, op, t = LEX_INT, r = LEX_BOOL;
    t2 = st_lex.pop();
    op = st_lex.pop();
    t1 = st_lex.pop();
    if(op==LEX_PLUS ||op==LEX_MINUS ||op==LEX_TIMES
||op==LEX_SLASH)
        r = LEX_INT;
    if (op == LEX_OR || op == LEX_AND)
        t = LEX_BOOL;
    if (t1 == t2  &&  t1 == t) st_lex.push(r);
    else throw "wrong types are in operation";
 // prog.put_lex (Lex (op) );
}
```

Для контроля за типом операнда одноместной операции *not* будем использовать функцию *check_not*:

```
void Parser::check_not () {
    if (st_lex.pop() != LEX_BOOL)
        throw "wrong type is in not";
    else {
        st_lex.push (LEX_BOOL);
    // prog.put_lex (Lex (LEX_NOT));
    }
}
```

В грамматике модельного языка задано старшинство операций: наивысший приоритет имеет операция отрицания, затем в порядке убывания приоритета — группа операций умножения (*, /, and), группа операций сложения (+, –, or), операции отношения.

$E \rightarrow E1 \mid E1\ [= \mid < \mid > \mid >= \mid <= \mid !=]\ E1$ 143
$E1 \rightarrow T\ \{[+ \mid - \mid or]\ T\}$
$T \rightarrow F\ \{[* \mid / \mid and]\ F\}$
$F \rightarrow I \mid N \mid [true \mid false] \mid not\ F \mid (E)$

Именно это свойство грамматики позволит провести синтаксически-управляемый контроль контекстных условий.

Правила вывода выражений модельного языка с действиями для контроля контекстных условий:

$E \rightarrow E1 \mid E1\ [= \mid < \mid > \mid >= \mid <= \mid !=] <st\_lex.push(c\_type)> E1 <check\_op()>$
$E1 \rightarrow T\ \{ [+ \mid - \mid or] <st\_lex.push\ (c\_type)> T <check\_op()>\}$
$T \rightarrow F\ \{ [* \mid / \mid and] <st\_lex.push\ (c\_type)> F <check\_op()>\}$
$F \rightarrow I <check\_id()> \mid N <st\_lex.push\ (LEX\_INT)> \mid$
$[true \mid false] <st\_lex.push\ (LEX\_BOOL)> \mid not\ F <check\_not()> \mid (E)$

Контроль контекстных условий в операторах

$S \rightarrow I := E \mid \textit{if } E \textit{ then } S \textit{ else } S \mid \textit{while } E \textit{ do } S \mid B \mid \textit{read (I)} \mid \textit{write (E)}$

1) **Оператор присваивания** $I := E$

Контекстное условие: в операторе присваивания типы переменной I и выражения E должны совпадать.

В результате контроля контекстных условий выражения E в стеке останется тип этого выражения (как тип результата последней операции); если при анализе идентификатора I проверить, описан ли он, и занести его тип в тот же стек (для этого можно использовать функцию check_id()), то достаточно будет в нужный момент считать из стека два элемента и сравнить их:

```
void Parser::eq_type () {
    if (st_lex.pop() != st_lex.pop())
        throw "wrong types are in :=";
}
```

Правило для оператора присваивания:

I <check_id()> := E <eq_type()>

2) Условный оператор и оператор цикла

if E then S else S | while E do S

Контекстные условия: в условном операторе и в операторе цикла в качестве условия возможны только логические выражения.

В результате контроля контекстных условий выражения E в стеке останется тип этого выражения (как тип результата последней операции); следовательно, достаточно извлечь его из стека и проверить:

```
void Parser::eq_bool () {
    if ( st_lex.pop() != LEX_BOOL )
        throw "expression is not boolean";
}
```

Правила для условного оператора и оператора цикла будут такими:

if E <eq_bool()> then S else S | while E <eq_bool()> do S

3) Для поверки операнда **оператора ввода** read (I) можно использовать следующую функцию:

```
void Parser::check_id_in_read () {
    if (! TID[c_val].get_declare()) throw "not declared";
}
```

Правило для оператора ввода будет таким:

read (I < check_id_in_read()>).

В итоге получаем процедуры для синтаксического анализа методом рекурсивного спуска с синтаксически управляемым контролем контекстных условий, которые легко написать по правилам грамматики с действиями.

В качестве примера приведем функцию для нетерминала *D* (раздел описаний):

```
void Parser::D () {
    st_int.reset();
    if (c_type != LEX_ID) throw curr_lex;
    else {
        st_int.push ( c_val );
        gl();
        while (c_type == LEX_COMMA){
            gl();
            if (c_type != LEX_ID) throw curr_lex;
            else {
                st_int.push ( c_val ); gl();
            }
        }
        if (c_type != LEX_COLON) throw curr_lex;
        else {gl();
          if (c_type == LEX_INT) {dec ( LEX_INT ); gl();}
            else
              if (c_type == LEX_BOOL){dec ( LEX_BOOL );
                                                  gl();}
                else throw curr_lex;
}}}
```

Внутреннее представление программы

Основные свойства языка внутреннего представления программ:

1) внутреннее представление фиксирует синтаксическую структуру исходной программы;

2) генерация внутреннего представления происходит в процессе синтаксического анализа;

3) конструкции языка внутреннего представления должны относительно просто транслироваться в объектный код либо достаточно эффективно интерпретироваться.

Некоторые способы внутреннего представления программ:

(а) постфиксная запись

(б) префиксная запись

(в) многоадресный код с явно именуемыми результатами

(г) многоадресный код с неявно именуемыми результатами

(д) связные списочные структуры, представляющие синтаксическое дерево.

В основе каждого из этих способов лежит некоторый метод представления синтаксического дерева.

ПОЛИЗ – польская инверсная запись **(постфиксная запись)**

Пример. Обычной (инфиксной) записи выражения

a*(b+c)-(d-e)/f

соответствует такая постфиксная запись:

abc+*de-f/-

- порядок операндов остался таким же, как и в инфиксной записи,
- учтено старшинство операций,
- нет скобок.

Простым будем называть выражение, состоящее из одной константы или имени переменной.

Приоритет и ассоциативность операций в инфиксных выражениях позволяют четко установить границы операндов:

a — простое выражение ;

$a + b * c \sim a + (b * c)$;

$a - b + c - d \sim ((a - b) + c) - d$.

ПОЛИЗ выражений

(1) если E является простым выражением, то ПОЛИЗ выражения E — это само выражение E;

(2) ПОЛИЗом выражения $\boldsymbol{E_1\ \theta\ E_2}$,
где θ — знак бинарной операции, E_1 и E_2 операнды для θ,
является запись $\boldsymbol{E_1'\ E_2'\ \theta}$,
где E_1' и E_2' — ПОЛИЗ выражений E_1 и E_2 соответственно;

(3) ПОЛИЗом выражения $\boldsymbol{\theta\ E}$, где θ — знак унарной операции, а E — операнд θ,
является запись $\boldsymbol{E'\ \theta}$,
где E' — ПОЛИЗ выражения E;

(4) ПОЛИЗом выражения (E) является ПОЛИЗ выражения E.

Алгоритм интерпретации с помощью стека

ПОЛИЗ просматривается поэлементно слева направо. В стеке хранятся значения промежуточных вычислений и результат.

(1) если очередной элемент — операнд, то его значение заносится в стек;

(2) если очередной элемент — операция, то на "вершине" стека сейчас находятся ее операнды (это следует из определения ПОЛИЗа и предшествующих действий алгоритма); они извлекаются из стека, над ними выполняется операция, результат снова заносится в стек;

(3) когда выражение, записанное в ПОЛИЗе, прочитано, в стеке останется один элемент — это значение всего выражения.

Замечание: для интерпретации, кроме ПОЛИЗа выражения, необходима дополнительная информация об операндах, хранящаяся в таблицах.

Алгоритм Дейкстры перевода в ПОЛИЗ выражений

Будем считать, что ПОЛИЗ выражения будет формироваться в массиве, содержащем лексемы — элементы ПОЛИЗа, и при переводе в ПОЛИЗ будет использоваться вспомогательный стек, также содержащий элементы ПОЛИЗа — операции, имена функций и круглые скобки.

1. Выражение просматривается один раз слева направо.

2. Пока есть непрочитанные лексемы входного выражения, выполняем действия:

а) Читаем очередную лексему.

б) Если лексема является числом или переменной, добавляем ее в ПОЛИЗ-массив.

в) Если лексема является символом функции, помещаем ее в стек.

г) Если лексема является разделителем аргументов функции (например, запятая):

до тех пор, пока верхним элементом стека не станет открывающаяся скобка, выталкиваем элементы из стека в ПОЛИЗ-массив. Если открывающаяся скобка не встретилась, это означает, что в выражении либо неверно поставлен разделитель, либо несогласованы скобки.

д) Если лексема является операцией θ, тогда:

1) пока приоритет θ меньше либо равен приоритету операции, находящейся на вершине стека (для лево-ассоциативных операций), или приоритет θ строго меньше приоритета операции, находящейся на вершине стека (для право-ассоциативных операций) выталкиваем верхние элементы стека в ПОЛИЗ-массив;
2) помещаем операцию θ в стек.

е) Если лексема является открывающей скобкой, помещаем ее в стек.

ж) Если лексема является закрывающей скобкой, выталкиваем элементы из стека в ПОЛИЗ-массив до тех пор, пока на вершине стека не окажется открывающая скобка. При этом открывающая скобка удаляется из стека, но в ПОЛИЗ-массив не добавляется. Если после этого шага на вершине стека оказывается символ функции, выталкиваем его в ПОЛИЗ-массив. Если в процессе выталкивания открывающей скобки не нашлось и стек пуст, это означает, что в выражении не согласованы скобки.

3. Когда просмотр входного выражения завершен, выталкиваем все оставшиеся в стеке символы в ПОЛИЗ-массив. (В стеке должны были оставаться только символы операций; если это не так, значит в выражении не согласованы скобки.)

Представление операторов

Оператор присваивания

I := E

в ПОЛИЗе будет записан как

I E’ :=

где ":=" – это двухместная операция, а I и E – ее операнды; подчеркнутое I означает, что операндом операции ":=" является адрес переменной I, а не ее значение. Вместо подчеркивания можно также использовать обозначение &I.

Расширение набора операций ПОЛИЗА

Операция перехода (обозначается «!») в терминах ПОЛИЗа означает, что процесс интерпретации надо продолжить с того элемента ПОЛИЗа, который указан как операнд этой операции.

Чтобы можно было ссылаться на элементы ПОЛИЗа, будем считать, что все они перенумерованы, начиная с 1 (например, занесены в последовательные элементы одномерного массива).

Пусть ПОЛИЗ оператора, помеченного меткой L, начинается с номера p, тогда оператор перехода

goto L в ПОЛИЗе записывается так:

p !

где ! – операция выбора элемента ПОЛИЗа, номер которого равен p.

Операция условный переход "по лжи" с семантикой

if (!B) goto L

Это двухместная операция с операндами B и L. Обозначим ее !F, тогда в ПОЛИЗе переход «по лжи» записывается так:

B’ p !F

где p — номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L, B’ — ПОЛИЗ логического выражения B.

Семантика *условного оператора*

if E then S_1 **else** S_2

с использованием введенной операции может быть описана так:

if (! E) goto L_2; S_1; goto L_3; L_2: S_2; L_3: ...

Тогда ПОЛИЗ условного оператора будет таким (порядок операндов — прежний):

E’ p_2 !F S_1’ p_3 ! S_2’ ... ,

где p_i – номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i, i = 2,3, E’ – ПОЛИЗ логического выражения E.

Семантика оператора *цикла* **while E do** S может быть описана так:

L_0: if (! E) goto L_1; S; goto L_0; L_1: ... .

Тогда ПОЛИЗ оператора цикла while будет таким (порядок операндов – прежний!):

E’ p_1 !F S’ p_0 ! ... ,

где p_i - номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i, i = 0,1, E’ – ПОЛИЗ логического выражения E.

Операторы ввода и вывода М-языка являются одноместными операциями.

Оператор ввода **read** (I) в ПОЛИЗе будет записан как **I read**

Оператор вывода **write** (E) в ПОЛИЗе будет записан как **E’ write**,

где E’ – ПОЛИЗ выражения E.

Синтаксически управляемый перевод

В основе синтаксически управляемого перевода лежит уже известная нам грамматика с действиями.

$G_{\texttt{expr}}$ – грамматика, описывающая простейшее арифметическое выражение:

$$E \to T \{+T\}$$
$$T \to F \{*F\}$$
$$F \to a \mid b \mid (E)$$

$G_{\texttt{expr\_polish}}$ – грамматика с действиями по переводу выражения в ПОЛИЗ :

$$E \to T \{+T \langle\, cout << '+';\rangle\}$$
$$T \to F \{*F \langle\, cout << '*';\rangle\}$$
$$F \to a \langle\, cout << 'a';\rangle \mid b \langle\, cout << 'b';\rangle \mid (E)$$

В процессе анализа методом рекурсивного спуска входной цепочки $a + b * c$ по грамматике $G_{\texttt{expr\_polish}}$ в выходной поток будет выведена цепочка $a\ \ b\ \ c * +$

Определение: Пусть T_1 и T_2 — алфавиты. *Формальный перевод* τ — это подмножество множества всевозможных пар цепочек в алфавитах T_1 и T_2: $\tau \subseteq (T_1^* \times T_2^*)$.

Назовем *входным* языком перевода τ язык $L_{вх}(\tau) = \{\alpha \mid \exists\, \beta : (\alpha, \beta) \in \tau \}$.

Назовем *целевым* (или *выходным*) языком перевода τ язык $L_{ц}(\tau) = \{\beta \mid \exists\, \alpha : (\alpha, \beta) \in \tau \}$.

Перевод τ *неоднозначен*, если для некоторых $\alpha \in T_1^*$, β, $\gamma \in T_2^*$, $\beta \neq \gamma$ справедливы соотношения: $(\alpha, \beta) \in \tau$ и $(\alpha, \gamma) \in \tau$.

Рассмотренная выше грамматика $G_{\texttt{expr\_polish}}$ задает однозначный перевод: каждому выражению ставится в соответствие единственная польская запись. Неоднозначные переводы могут быть интересны при изучении моделей естественных языков; для трансляции языков программирования используются однозначные переводы.

Генератор внутреннего представления программы на М-языке

Каждый элемент в ПОЛИЗе – это лексема, т.е. пара вида (тип_лексемы, значение_лексемы). При генерации ПОЛИЗа будем использовать дополнительные типы лексем:

POLIZ_GO – “!” ;
POLIZ_FGO – “!F” ;
POLIZ_LABEL– для ссылок на номера элементов ПОЛИЗа;
POLIZ_ADDRESS – для обозначения операндов-адресов (например, в ПОЛИЗе оператора присваивания).

Будем считать, что генерируемая программа размещается в объекте

```
Poliz  prog (1000);
```

класса Poliz:

```
    class Poliz{
      lex *p;
      int size;
      int free;
    public:
      Poliz(int max_size){p=new Lex [size =
  max_size]; free = 0;};
      ~Poliz(){delete []p;};
      void put_lex(Lex l){p[free]=l; free++;};
      void put_lex(Lex l, int place){p[place]=l;};
      void blank(){free++;};
      int get_free(){return free;};
      lex& operator[](int index){
          if (index > size) throw "POLIZ:out of
  array";else
    if(index > free) throw "POLIZ:indefinite element
                                      of array";
          else return p[index];
      };
      void print(){
      for (int i=0; i < free; i++) cout << p[i]; };
};
```

Добавим действия по генерации в некоторые функции семантического анализа: check_not( ) и check_op( ).

```
void Parser::check_not () {
    if (st_lex.pop() != LEX_BOOL)
        throw "wrong type is in not";
    else {
        st_lex.push (LEX_BOOL);
        prog.put_lex (Lex (LEX_NOT));
    }
}
```

```
    void Parser::check_op () {
        type_of_lex t1, t2, op, t = LEX_INT, r =
LEX_BOOL;
        t2 = st_lex.pop();
        op = st_lex.pop();
        t1 = st_lex.pop();
        if(op==LEX_PLUS ||op==LEX_MINUS
||op==LEX_TIMES ||op==LEX_SLASH)
            r = LEX_INT;
        if (op == LEX_OR || op == LEX_AND)
            t = LEX_BOOL;
        if (t1 == t2  &&  t1 == t) st_lex.push(r);
        else throw "wrong types are in operation";
        prog.put_lex (Lex (op) );
    }
```

Грамматика, содержащая действия по контролю контекстных условий и переводу выражений модельного языка в ПОЛИЗ

$E \rightarrow E1 \mid E1\ [\ = \mid < \mid > \mid <= \mid >=\ ]\ \langle\ st\_lex.push\ (c\_type)\ \rangle\ E1\ \langle\ check\_op\ (\ )\ \rangle$

$E1 \rightarrow T\ \{\ [\ + \mid - \mid or\ ]\ \langle\ st\_lex.push\ (c\_type\ )\ \rangle\ T\ \langle\ check\_op\ (\ )\ \rangle\ \}$

$T \rightarrow F\ \{\ [\ * \mid / \mid and\ ]\ \langle\ st\_lex.push\ (c\_type\ )\ \rangle\ F\ \langle\ check\_op\ (\ )\ \rangle\ \}$

$F \rightarrow I\ \langle\ check\_id\ (\ );\ prog.put\_lex\ (\ curr\_lex\ );\ \rangle \mid$

$N\ \langle\ st\_lex.push\ (LEX\_INT);\ prog.put\_lex\ (\ curr\_lex\ );\ \rangle \mid$

$[\ true \mid false\ ]\ \langle\ st\_lex.push\ (LEX\_BOOL);\ prog.put\_lex\ (\ curr\_lex\ );\ \rangle \mid$

$not\ F\ \langle\ check\_not(\ );\ \rangle \mid (E)$

Пример реализации процедуры анализа и перевода для нетерминала F :


```
void Parser::F ()
{
  if ( c_type == LEX_ID )
  {
    check_id();
    prog.put_lex (Lex (LEX_ID, c_val));
    gl();
  }
  else if ( c_type == LEX_NUM )
  {
    st_lex.push ( LEX_INT );
    prog.put_lex ( curr_lex );
    gl();
  }
  else if ( c_type == LEX_TRUE )
  {
    st_lex.push ( LEX_BOOL );
    prog.put_lex (Lex (LEX_TRUE, 1) );
    gl();
  }
```

```
  else if ( c_type == LEX_FALSE)
  {
    st_lex.push ( LEX_BOOL );
    prog.put_lex (Lex (LEX_FALSE, 0) );
    gl();
  }
  else if (c_type == LEX_NOT)
  {
    gl();
    F();
    check_not();
  }
  else if ( c_type == LEX_LPAREN )
  {
    gl();
    E();
    if ( c_type == LEX_RPAREN)
      gl();
    else
      throw curr_lex;
  }
  else
    throw curr_lex;
}
```

Действия для оператора присваивания

$S \to I \langle$ *check_id* ( ); *prog.put_lex* (*Lex* (*POLIZ_ADDRESS, c_val*)); $\rangle$:= $E \langle$*eqtype* ( ); *prog.put_lex* (*Lex* (*LEX_ASSIGN*)); $\rangle$

Для условного

if (!E) goto l2; S_1; goto l3; l2: S_2; l3:…

$S \to$ ***if*** $E \langle$ *eqbool*( ); *pl2 = prog.get_free*( ); *prog.blank*( );
prog.put_lex (*Lex* (*POLIZ_FGO*)); $\rangle$
then $S_1 \langle$ *pl3 = prog.get_free*( ); *prog.blank*( );
prog.put_lex (*Lex* (*POLIZ_GO*));
prog.put_lex (*Lex* (*POLIZ_LABEL, prog.get_free*( )), *pl2*); $\rangle$
else $S_2 \langle$ *prog.put_lex* (*Lex* (*POLIZ_LABEL, prog.get_free*( )), *pl3*); $\rangle$

Оператор цикла **while E do** S описывается так:

$$L_0\text{: if (!E) goto } l_1\text{; S; goto } l_0\text{; } l_1\text{: ... .}$$

а грамматика с действиями по контролю контекстных условий и переводу оператора цикла в ПОЛИЗ будет такой:

$S \rightarrow$ ***while*** ⟨ *pl0 = prog.get_free* ( )*;* ⟩ *E* ⟨ *eqbool* ( )*;*
pl1 = prog.get_free ( )*; prog.blank* ( )*;*
prog.put_lex (*Lex* (*POLIZ_FGO*))*;* ⟩
do *S* ⟨*prog.put_lex* (*Lex* (*POLIZ_LABEL, pl0*)*;*
prog.put_lex (*Lex* (*POLIZ_GO*))*;*
prog.put_lex (*Lex* (*POLIZ_LABEL, prog.get_free*( ))*, pl1*)*;* ⟩

Замечание: переменные pli (i=0,1,2,3) должны быть локализованы в процедуре S, иначе возникнет ошибка при обработке вложенных условных операторов.

Грамматика с действиями по контролю контекстных условий и переводу в ПОЛИЗ операторов ввода и вывода:

$S \rightarrow$ *read* (*I* ⟨ *check_id_in_read*( );
prog.put_lex (*Lex* (*POLIZ_ADDRESS, c_val*)); ⟩)
⟨ *prog.put_lex* (*Lex* (*LEX_READ*)); ⟩

$S \rightarrow$ *write* (*E*) ⟨ *prog.put_lex* (*Lex* (*LEX_WRITE*)); ⟩

Интерпретатор ПОЛИЗа для модельного языка

```
class Executer {
    Lex pc_el;
    public:
        void execute (Poliz& prog);
};

void Executer::execute ( Poliz& prog ) {
    Stack < int, 100 > args;
    int i, j, index = 0, size = prog.get_free();
    while ( index < size ) {
        pc_el = prog [ index ];
        switch ( pc_el.get_type () ) {
            case LEX_TRUE: case LEX_FALSE: case LEX_NUM:
            case POLIZ_ADDRESS: case POLIZ_LABEL:
                args.push ( pc_el.get_value () ); break;
            case LEX_ID:
                i = pc_el.get_value ();
                if ( TID[i].get_assign () ){
                     args.push ( TID[i].get_value () );
                                                  break;}
                else throw "POLIZ: indefinite identifier";
```

```
case LEX_NOT:
    args.push( !args.pop() ); break;
case LEX_OR:
    i = args.pop();
    args.push ( args.pop() || i ); break;
case LEX_AND:
    i = args.pop();
    args.push ( args.pop() && i ); break;
case POLIZ_GO:
    index = args.pop() - 1; break;
case POLIZ_FGO:
    i = args.pop();
    if ( !args.pop() ) index = i-1; break;
case LEX_WRITE:
    cout << args.pop () << endl; break;
```

```
case LEX_READ:
    {int k;
    i = args.pop ();
    if ( TID[i].get_type () == LEX_INT ){
        cout << "Input int value for";
        cout << TID[i].get_name () << endl;
        cin >> k;
    }
    else {
        char j[20];
        rep:
        cout << "Input boolean value;
        cout << (true or false) for";
        cout << TID[i].get_name() << endl;
        cin >> j;
        if (!strcmp(j, "true")) k = 1;
        else
        if (!strcmp(j, "false")) k = 0;
        else {
            cout<< "Error in input:true/false";
            cout << endl;
```

```
                        goto rep;}
                }
                TID[i].put_value (k);
                TID[i].put_assign ();
                break;}
            case LEX_PLUS:
                args.push ( args.pop() + args.pop() ); break;
            case LEX_TIMES:
                args.push ( args.pop() * args.pop() ); break;
            case LEX_MINUS:
                i = args.pop();
                args.push ( args.pop() - i ); break;
            case LEX_SLASH:
                i = args.pop();
                if (!i) { args.push(args.pop() / i); break;}
                else throw "POLIZ:divide by zero";
            case LEX_EQ:
                args.push ( args.pop() == args.pop() );
break;
            case LEX_LSS:
                i = args.pop();
                args.push ( args.pop() < i); break;
            case LEX_GTR:
```

```
                i = args.pop();
                args.push ( args.pop() > i ); break;
            case LEX_LEQ:
                i = args.pop();
                args.push ( args.pop() <= i ); break;
            case LEX_GEQ:
                i = args.pop();
                args.push ( args.pop() >= i ); break;
            case LEX_NEQ:
                i = args.pop();
                args.push ( args.pop() != i ); break;

            case LEX_ASSIGN:
                i = args.pop();
                j = args.pop();
                TID[j].put_value(i);
                TID[j].put_assign(); break;
            default: throw "POLIZ: unexpected elem";
        }//end of switch
        index++;
        };//end of while
    cout << "Finish of executing!!!" << endl;
}
```

```
class Interpretator {
        Parser pars;
        Executer E;
    public:
        Interpretator (char* program): pars (program){};
        void interpretation ();
};


void Interpretator::interpretation (){
        pars.analyze ();
        E.execute ( pars.prog );
}
```

```
int main () {
    try {
            Interpretator I ("program.txt");
            I.interpretation ();
            return 0;
    }
    catch (char c) {
        cout << "unexpected symbol " << c << endl; return 1;
    }
    catch (Lex l) {
        cout << "unexpected lexeme"; cout << l; return 1;
    }
    catch(const char *source) {
        cout << source << endl; return 1;
    }
}
```

Распределение памяти

Распределение памяти - это процесс, в результате которого отдельным элементам исходной программы ставятся в соответствие адрес, размер и атрибуты области памяти, необходимой для размещения лексических единиц.

Область памяти - это блок ячеек памяти, выделяемых для данных и каким-то образом объединенных логически.

Распределение памяти выполняется после фазы анализа текста исходной программы **на этапе подготовки к генерации объектного модуля** (перед генерацией кода объектного модуля).

Исходными данными для процесса распределения памяти служат сведения о семантике конструкций ЯП, таблица идентификаторов, построенная лексическим анализатором и информация, полученная синтаксическим анализатором при анализе декларативной чисти программы.

Современные компиляторы, в основном, работают с относительными, а не с абсолютными адресами ячеек памяти.

Распределение памяти

Семантика программ подразумевает, что при их выполнении области памяти будут необходимы для хранения:

- кодов пользовательских программ;
- данных, необходимых для работы этих программ;
- кодов системных программ, обеспечивающих поддержку пользовательских программ в период их выполнения;
- записей о текущем состоянии процесса выполнения программ (например, записей об активации процедур).

По **способу использования** области памяти делятся на ***глобальные*** и ***локальные***, а по **способу распределения** – на ***статические*** и ***динамические***.

Классы памяти

Выделяемую память можно разделить на локальную / глобальную и статическую / динамическую.

Классы памяти

Локальная память - это область памяти, которая выделяется в начале выполнения некоторого фрагмента результирующей программы (блока, функции, оператора…) и может быть освобождена по завершении выполнения данного фрагмента. Доступ к локальной области памяти всегда запрещен за пределами того фрагмента программы, в котором она выделяется.

Глобальная память - это область памяти, которая выделяется один раз при инициализации результирующей программы и действует все время выполнения программы. Как правило, глобальная область памяти доступна из любой части исходной программы.

Статическая память - это область памяти, размер которой известен на этапе компиляции. Для статической памяти компилятор порождает некоторый адрес (как правило, относительный), и дальнейшая работа с ней происходит относительно этого адреса.

Динамическая память - это область памяти, размер которой становится известным только на этапе выполнения результирующей программы. Для динамической памяти компилятор порождает фрагмент кода, который отвечает за распределение памяти (ее выделение и освобождение). Как правило, с динамическими областям памяти связаны многие операции с указателями и с экземплярами объектов (классов) в ООЯП.

Динамические области памяти можно разделить на динамические области памяти, **выделяемые пользователем** и **непосредственно компилятором**.

Общие принципы генерации объектного кода

При генерации объектного кода компилятор переводит текст программы во внутреннем представлении в текст программы на выходном языке (как правило, машинном).

Генерация объектного кода происходит на основе:

- определенной на фазе анализа компиляции синтаксической структуры программы,
- информации, хранящейся в таблице идентификаторов,
- результата распределения памяти.

Характер и сложность отображения промежуточного представления программы в последовательность команд на машинном языке зависит от языка внутреннего представления и архитектуры вычислительной системы, на которую ориентирована результирующая программа.

Часто для построения кода результирующей программы компиляторы используют синтаксически управляемый перевод.

Оптимизация программ

Оптимизация программы - это изменение компилируемой программы (в основном переупорядочивание и замена операций) с целью получения более эффективной объектной программы.

Используются два критерия эффективности результирующей программы :

- **скорость** выполнения программы и
- **объем памяти**, необходимый для выполнения программы.

В общем случае задача построения оптимального кода программы алгоритмически неразрешима. К тому же, компилятор обладает весьма ограниченными средствами анализа семантики входной программы в целом.

Основная оптимизация программы должна производится программистом.

Принципиально различаются два основных вида оптимизирующих преобразований:

- **машинно-независимые преобразования** исходной программы,
- **машинно-зависимые преобразования** результирующей объектной программы.

Оптимизация может привести к изменению смысла программы. Например, в случае исключения из программы вызова функции с "побочным эффектом".

У современных компиляторов существует возможность выбора критерия оптимизации и отдельных методов оптимизации.

Машинно-независимые оптимизирующие преобразования

Машинно-независимые преобразования исходной программы производятся в основном над ее внутренним представлением и основаны на известных математических и логических преобразованиях.

1. Удаление недостижимого кода.

(задача компилятора найти и убрать его).

Пример:

```
if (1)
    S1;          ⇒      S1;
else
    S2;
```

Машинно-независимые оптимизирующие преобразования

2. Оптимизация линейных участков программы.

В современных системах программирования профилировщик на основе результатов запуска программы выдаёт информацию о том, на какие её линейные участки приходится основное время выполнения.

а) Удаление бесполезных присваиваний.

$$a = b * c;\ d = b + c;\ a = d * c;\ \Rightarrow\ d = b + c;\ a = d * c;$$

Однако, в следующем примере эта операция уже не бесполезна:

$$p = \&\ a;\ a = b * c;\ d = *\ p + c;\ a = d * c;$$

б) Исключение избыточных вычислений.

$$d = d + b * c;\ a = d + b * c;\ c = d + b * c; \Rightarrow$$
$$t = b * c;\ d = d + t;\ a = d + t;\ c = a;$$

в) Свертка объектного кода (выполнение во время компиляции тех операций исходной программы, для которых значения операндов уже известны).

$$i = 2 + 1;\ j = 6 * i + i; \Rightarrow i = 3;\ j = 21;$$

Машинно-независимые оптимизирующие преобразования

г) Перестановка операций (для дальнейшей свертки или оптимизации вычислений).

a = 2 * b * 3 * c; $\Rightarrow$ ***a = (2 * 3) * (b * c);***

a = (b + c) + (d + e); $\Rightarrow$ ***a = (b + (c + (d + e)));***

д) Арифметические преобразования (на основе алгебраических и логических тождеств).

a = b * c + b * d; $\Rightarrow$ ***a = b * (c + d);***

a * 1 $\Rightarrow$ ***a,*** ***a * 0*** $\Rightarrow$ ***0,*** ***a + 0*** $\Rightarrow$ ***a .***

е) Оптимизация вычисления логических выражений.

a* || *b* || *c* || *d; $\Rightarrow$ ***a,*** если **a** есть **true**.

Но! ***a* || *f(b)* || *g(c)*** не всегда **a** (при **a = true**), может быть побочный эффект.

Машинно-независимые оптимизирующие преобразования

3. Подстановка кода функции вместо ее вызова в объектный код.

Этот метод, как правило, применим к простым функциям и процедурам, вызываемым непосредственно по адресу, без применения косвенной адресации через таблицы RTTI (Run Time Type Information).

Некоторые компиляторы допускают применять метод только к функциям, содержащим последовательные вычисления без циклов.

Язык C++ позволяет явно указать (inline), для каких функций желательно использовать inline-подстановку.

Машинно-независимые оптимизирующие преобразования

4. Оптимизация циклов.

а) Вынесение инвариантных вычислений из циклов.

```
for (i = 1; i <= 10; i++)
        a [i] = b * c * a [i];                    ⇒
d = b * c;  for (i = 1; i <= 10; i++)
                        a [i] = d * a [i];
```

б) Замена операций с индуктивными (образующими арифметическую прогрессию) переменными (как правило, умножения на сложение).

```
for (i = 1; i <= N; i++)
        a [i] = i * 10;                           ⇒
t = 10; i = 1; while (i <= N) {
                    a [i] = t;    t = t + 10;    i++;
               }

s = 10; for (i = 1; i <= N; i++) {
         r = r + f (s); s = s + 10; }             ⇒
s = 10; m = N * 10; while (s <= m) {
                        r = r + f (s); s = s + 10; }
```

(избавились от одной индуктивной переменной).

Машинно-независимые оптимизирующие преобразования

в) **Слияние циклов**.

```
for (i =1; i <= N; i++)
      for (j = 1; j <= M; j++)
            a [i] [j] = 0;        ⇒

k = N * M;
for (i = 1; i <= k; i++)
            a [i] = 0;
```
(только в объектном коде!)

г) **Развертывание циклов** (можно выполнить для циклов, кратность выполнения которых известна на этапе компиляции).

```
for (i = 1; i <= 3; i++)
            a [i] = i;        ⇒

a [1] = 1;
a [2] = 2;
a [3] = 3;
```

Машинно-зависимые оптимизирующие преобразования

Машинно-зависимые преобразования результирующей объектной программы зависят от архитектуры вычислительной системы, на которой будет выполняться результирующая программа. При этом может учитываться объем кэш-памяти, методы организации работы процессора ….

Эти преобразования, как правило, являются "ноу-хау", и именно они позволяют существенно повысить эффективность результирующего кода.

.

1. Распределение регистров процессора.

Использование регистров общего назначения и специальных регистров (аккумулятор, счетчик цикла, базовый указатель) для хранения значения операндов и результатов вычислений позволяет увеличить быстродействие программы.

Доступных регистров всегда ограниченное количество, поэтому перед компилятором встает вопрос их оптимального распределения и использования при выполнении вычислений.

Машинно-зависимые оптимизирующие преобразования

2. Оптимизация передачи параметров в процедуры и функции.

Обычно параметры процедур и функций передаются через стек. При этом всякий раз при вызове процедуры или функции компилятор создает объектный код для размещения ее фактических параметров в стеке, а при выходе из нее - код для освобождения соответствующей памяти.

Можно уменьшить код и время выполнения результирующей программы за счет оптимизации передачи параметров в процедуру или функцию, передавая их **через регистры** процессора.

Реализация данного оптимизирующего преобразования зависит от количества доступных регистров процессора в целевой вычислительной системе и от используемого компилятором алгоритма распределения регистров.

Недостатки метода:

- оптимизированные таким образом процедуры и функции не могут быть использованы в качестве **библиотечных**, т.к. методы передачи параметров через регистры не стандартизованы и зависят от реализации компилятора.
- этот метод не может быть использован, если где-либо в функции требуется выполнить **операции с адресами** параметров.

Языки Си и C++ позволяют явно указать (**register**), какие параметры и локальные переменные желательно разместить в регистрах.

Машинно-зависимые оптимизирующие преобразования

3. Оптимизация кода для процессоров, допускающих распараллеливание вычислений.

При возможности параллельного выполнения нескольких операций компилятор должен порождать объектный код таким образом, чтобы в нем было максимально возможное количество соседних операций, все операнды которых не зависят друг от друга.

Для этого надо найти оптимальный порядок выполнения операций для каждого оператора (переставить их).

$a + b + c + d + e + f; \Rightarrow$

для одного потока обработки данных: $((((a + b) + c) + d) + e) + f;$

для двух потоков обработки данных: $((a + b) + c) + ((d + e) + f);$

для трех потоков обработки данных: $(a + b) + (c + d) + (e + f);$

Описание языка программирования

- **Алфавит** задается перечислением конечного непустого множества символов, которые могут быть использованы для записи текстов на языке.

- **Синтаксис** определяется набором правил, устанавливающих, какие комбинации символов алфавита являются правильными текстами на определяемом языке и позволяющих связать с каждым правильным текстом на этом языке некоторую синтаксическую структуру.

- **Семантика** определяет смысл синтаксически правильных конструкций языка, то, что означает конструкция.

  Семантика обычно описывается словами. Четкое и точное описание семантики очень важно для транслятора, т.к. его цель - получить *эквивалентную* программу на МЯ (для компилятора), либо точно выполнить указанные действия (для интерпретатора).

- **Прагматика** языка объясняет как та или иная конструкция из состава языка используется и для решения каких задач она предназанчена