Ответы на вопросы экзамена по курсу «Языки программирования» 09.01.2017

В ответах курсивом выделены необязательные пояснения, которые можно опустить (особенно на экзамене)

Вариант 1

Задача 1-1

Есть ли ошибка в тексте приведенной ниже функции `r_vowels(s)` на языке JavaScript (функция подсчитывает число гласных букв русского алфавита в строке-аргументе s)? Если есть, то объясните, в чем она состоит и как изменить текст функции, чтобы она работала правильно во всех реализациях языка?

```
function r_vowels(s) {
  var i,cnt = 0; const vowels = 'АЕЁИОУЫЭЮЯаеёиоуыэюя';
  for (i=0; i<s.length; ++i) if (vowels.indexOf(s[i])>=0) ++cnt;
  return cnt;
}
```

Ответ

Ошибка состоит в том, что операцию индексирования в JavaScript некорректно применять к строкам (поэтому большинство современных руководств по языку прямо запрещает индексирование строк). Это связано с тем, что операция индексирования строки корректно выдает символ с соответствующим индексом только в случае использования кодировки с фиксированной длиной представления символа (однобайтовые кодировки типа Win1251, кодировка Юникода UTF16 и т. п.). В случае кодировок с переменной длиной, таких как кодировка Юникода UTF8 (а именно эта кодировка используется в большинстве реализаций JavaScript!), i-й байт строки может не иметь отношения к i-му символу.
Исправить ошибку можно (как минимум) двумя способами — либо использовать операцию выделения i-го символа из строки (`s.substr(i,1)` или `s.slice(i,i+1)`) вместо индексирования `s[i]`, либо разбить строку `s` на символы с помощью операции `split()` с пустой строкой-разделителем (перед циклом написать `s=s.split('')`), получив массив строк из одного символа, который вполне корректно индексировать.

Задача 1-2

Что будет выдано в стандартный канал вывода после выполнения следующей программы на C++? Считать, что все нужные директивы `include` и `using` добавлены в текст.

```
class TT {
  char * _p;
public:
  TT(const TT& t) { _p = strdup(t._p); cout << "TT& " << _p<< endl;}
  explicit TT(const char * p = "") {_p = strdup(p); cout << "Hello, "<< _p << endl; }
  TT& operator = (const TT& t) {
    if (this != &t) {
      cout << "goodbye, " << _p << endl; free(_p);  _p = strdup(t._p);
    }
    cout << "Hello, " << _p << endl; return *this;
  }
  ~TT() { cout << "goodbye, " << _p << endl; free(_p);}
};
int main() { TT t("world"); t = TT("bar"); return 0; }
```

Добавьте в класс TT РОВНО ОДИН метод так, чтобы результатом стали следующие

строки:
Hello, world
Hello, bar
Hello, bar
goodbye, world
goodbye, bar

Ответ

Будет выдано (нумерацию строк игнорировать):

```
1 Hello, world
2 Hello, bar
3 goodbye, world
4 Hello, bar
5 goodbye, bar
6 goodbye, bar
```

Объяснение:
строка 1; конструктор TT(const char) для объекта t с параметром «world»*
строка 2; конструктор TT(const char) для временного объекта с параметром «bar»*
строка 3,4; операция присваивания TT::operator =(const T&) для объекта t с параметром-временным объектом. Теперь t содержит строку "bar" так же, как и временный объект
строка 5; деструктор временного объекта
строка 6; деструктор объекта t

Для обеспечения требуемой выдачи можно добавить перегрузку операции присваивания с семантикой перемещения (move-семантикой). Внутри этой операции меняем местами указатели _p. В список методов класса добавляем строки:

```
TT& operator = (TT&& t) {
    char * p = t._p; t._p = _p; _p = p;
    cout << "Hello, " << _p << endl;
    return *this;
 }
```

Задача 1-3

Объясните, что означает понятие «сопрограммы Кнута». В чем отличие сопрограмм языка Python от сопрограмм Кнута? Приведите пример языка, который ближе, чем Python, реализует сопрограммы Кнута (с примерами соответствующих языковых конструкций).

Ответ

Главное свойство сопрограммы состоит в том, что управление всегда передается в точку, непосредственно следующую за местом, где оно покинуло сопрограмму (в начальный момент — на первый оператор сопрограммы). В сопрограммах Кнута существует явный оператор передачи управления на сопрограмму (обычно называемый resume или transfer), в котором указывается имя или ссылка на вызываемую сопрограмму. Этот оператор является обобщением оператора вызова подпрограммы.
*Поскольку информация о точке возврата в сопрограмму не может храниться в общем стеке, если любая сопрограмма может вызвать любую другую сопрограмму, то иногда говорят о том, что сопрограммы Кнута реализуют бесстековый механизм. НАДО ПОНИМАТЬ, что речь идет о реализации механизма хранения информации о точках возврата, и что отсутствует здесь **общий** стек возвратов (как в случае подпрограмм) и только, а при этом каждая сопрограмма может обладать своим стеком — вызывать функции и т. п. Но из любой функции сопрограммы можно вызвать оператор передачи*

управления на другую сопрограмму, и вот тут-то информация о том, куда надо будет вернуться, уже не может быть сохранена в общем стеке, а нужно использовать более общий механизм (чаще всего это — продолжения или континуации, о которых мы в курсе ничего не говорили). Однако нельзя говорить о том, что-де в сопрограммах Кнута стека нет (какого именно стека?), а в сопрограммах Питона стек есть — или наоборот — в Питоне стека нет, а в СК — есть. Это неправильная постановка вопроса. Кроме того существуют облегченные варианты сопрограмм, реализация которых использует общий стек (что накладывает на них ряд ограничений). Опять-таки здесь снова речь о механизме реализации хранилища точек возврата.

Отличие сопрограмм Кнута от сопрограмм Питона (*и Шарпа*) состоит в том, что в Питоне предложение yield, с помощью которого реализуются сопрограммы (точнее — генераторы), не содержит информации о том, куда передается управление. *Поэтому некоторые источники (например, David Beasley) утверждают, что сопрограммы Питона — это не просто генераторы, а такие генераторы, управление на которые передается с помощью send, а сами сопрограммы явно принимают управление и данные с помощью специальной формы предложения yield.*

К концепции сопрограмм Кнута ближе, чем Питон, подошла реализация языка Модула-2. Там есть специальная процедура TRANSFER(VAR FROM,TO: ADDRESS), которая служит для передачи управления на сопрограмму, контекст которой, включая и адрес возврата, хранится в TO, а контекст текущей сопрограммы упрятывается в FROM. Начальное значение контекста задается процедурой NEWPROCESS(P:PROC; VAR CONTEXT:ADDRESS; N:CARDINAL), где P – процедура, по которой будет работать сопрограмма, N – размер области хранения контекста.

Заметим, что и Модула-2 не дает полной реализации сопрограмм Кнута, поскольку детали организации хранилища контекстов (включая даже размер!) полностью возлагает на программиста.

Задача 1-4

На языке Java написан класс **`Generator`** (не компилируется из-за отсутствия объявления имени и знаков вопроса в вызовах метода *`generate`*):

```
public class Generator {
  public static void main(String[] args) {
        generate(???, 10); // заменить ??? на правильное значение
        generate(???, 10); // заменить ??? на правильное значение
  }
  static void generate(Compute func, int n) {
    for (int i = 1; i <= n; ++i) { System.out.print(func.compute(i)); System.out.print(' ');}
  }
}
```

Объявить тип данных `Compute` и заменить вопросы в вызовах метода *`generate`* на правильные лямбда-выражения так, чтобы в канал вывода было выдано:

```
1 4 9 16 25 36 49 64 81 100 1 8 27 64 125 216 343 512 729 1000
```

Ответ

Во-первых, надо определить функциональный интерфейс `Compute` с единственным методом `compute(int x)`:

```
public interface Compute {
      int compute(int x);
}
```

Во-вторых, вставить вместо вопросов лямбда-выражения квадрата и куба:

```
generate(x->x*x, 10);
generate(x->x*x*x, 10);
```

Замечание: на удивление много человек пыталось решать эту задачу без лямбд, хотя в условии ПРЯМО указывалось на них. Также намекалось, что нужно объявить ТИП данных (а не класс!) Собственно, лямбда-функции и появились в языках типа Java для того, чтобы можно было избежать появления классов, единственным назначением которых являлась обертка функций в методы (поскольку двадцать лет назад большинство программистов вдруг решило, что все в программировании можно выразить объектами...). Задача является демонстрацией того, насколько проще использовать (лямбда-)функции, когда нужны именно функции. Так что люди расписавшие классы типа типа Compute (или еще хуже Printer из второго варианта) могут утешаться тем, что у них есть готовый ответ на вопрос, зачем же все-таки нужны лямбда-функции в Java. Чтобы не плодить джи-код.

Задача 1-5

Переписать программу из задачи №4 на языке C# так, чтобы она выдавала в стандартный вывод те же самые значения (набор и сигнатуры методов класса `Generator` должны остаться без изменения). Для вывода использовать **`System.Console.Write(…)`**.

Ответ

В отличие от Java, в которой «пришивание» лямбд пришлось делать слегка через задний ...вход — то есть придумав концепцию «функционального интерфейса» введения callable-объектов, в C# нужная заготовка была с самого начала — это делегаты. Не случайно анонимные делегаты(то есть типизированные лямбды) появились в языке уже со второй версии, а вскоре поспели и настоящие лямда-выражения. Поэтому в C# программа должна выглядеть более естественно, чем в Java. И не надо «тупо» транслировать то, что не транслируется (например, функциональные интерфейсы).

Вместо функционального интерфейса просто описываем делегатский тип
`delegate int Compute(int i);`
Объекты-делегаты — уже callables – то есть их можно вызывать (точнее применять к ним операцию вызова «()») напрямую без всяких методов-посредников типа `compute(i)`.

Окончательно получаем

```
 public class Generator
    {
        public delegate int Compute(int i);
        static void Main(String[] args) {
               generate(x=>x*x, 10);
               generate(x=>x*x*x, 10);
        }
        static void generate(Compute func, int n) {
            for (int i = 1; i <= n; ++i) {
                System.Console.Write(func(i)); // обратите внимание на отсутствие вызова
func.compute(i) - он больше не нужен
                System.Console.Write(' ');
            }
        }
    }
```

И все!

Задача 1-6

Объявление шаблонного класса **`CalcSort`** на языке C++ предполагает, что над параметром шаблона T выполняются 4 арифметические операции (+,-,*,/) и операции сравнения:

```
template <typename T> class CalcSort { … };
```

Напишите эквивалентное объявление обобщенного класса **`CalcSort`** на языке Java (опустив, как и выше, все объявления членов класса), добавив при необходимости нужные объявления типов вне этого класса.

Ответ

См. ответ и замечания по поводу задачи 2-6 из второго варианта, только в Java ограничения могут быть ТОЛЬКО в форме наследования класса или интерфейса (еще проще)

```
interface Arithmetic<T> {
        T Add(T x);
        T Mult(T x);
        T Sub(T x);
        T Div(T x);
}
class CalcSort<T extends Comparable<T>, Arithmetic<T> { … }
```

Задача 1-7

Дайте определение и пример локального анонимного класса в языке Java. Есть ли аналогичное понятие в языках C++ или C#?

Ответ

Коротко: детали ответа можно найти в книге Гослинга, Арнольда и Холмса «Язык программирования Java» 3-е издание, п.5.4. Анонимные внутренние классы, стр.148.

В C# и C++ такого понятия нет.

Замечание. Постоянная (и грубая) ошибка: путать внутренние и вложенные классы. Это - «две большие разницы». Подробно и обстоятельно написано там же в 5 главе «Вложенные классы и интерфейсы».

Задача 1-8

Пусть программа на языке C# содержит следующий ошибочный фрагмент:

```
class D { }
interface IFace <T, R>
{
   R Generate();
   void Process(T x);
}
IFace<D, Object> f1 = new ObjectFace(); //error!
IFace<D, Object> f2 = new DFace(); //error!

class ObjectFace : IFace<Object, Object> {
  public Object Generate() { return new Object();}
  public void Process(Object x) {}
}
class DFace : IFace<D, D> {
  public D Generate() { return new D(); }
  public void Process(D x) { }
}
```

Объясните, в чем состоит ошибка, и исправьте РОВНО ОДНУ строчку во фрагменте так, чтобы ошибка исчезла.

Ответ

Ошибка состоит в том, что по правилам языка конкретизации обобщенных интерфейсов (без указания вариантности по параматрам) инвариантны, если их фактические типы-параметры различны (независимо от вариантности типов-параметров). Поэтому IFace<D, Object> не приводится ни к IFace<Object, Object>, ни к IFace<D, D> (это и есть инвариантность).

В Java, кстати, такая же история. Но решается проблема в C# и Java по-разному. В C# вариантность интерфейса указывается при определении самого интерфейса по каждому типу — ковариантность указывается как out, а контравариантность — как in, что хорошо согласуется с тем, как типы-параметры используются в методах обобщенного интерфейса.

Интерфейс IFace ковариантен по типу R, и контравариантен по типу T. Поэтому вместо

```
interface IFace <T, R>
```

надо поставить

```
interface IFace <in T, out R>
```