

Министерство образования и науки Российской Федерации

Федеральное государственное автономное  
образовательное учреждение высшего образования  
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

**М. Э. Абрамян**

# **СТРУКТУРЫ ДАННЫХ В PASCALABC.NET**

## **Выпуск 1**

Массивы и последовательности. Запросы

*УЧЕБНОЕ ПОСОБИЕ ПО КУРСУ  
«ОСНОВЫ ПРОГРАММИРОВАНИЯ»  
ДЛЯ СТУДЕНТОВ ЕСТЕСТВЕННОНАУЧНЫХ  
И ТЕХНИЧЕСКИХ СПЕЦИАЛЬНОСТЕЙ*

Ростов-на-Дону 2016

УДК 004.438.NET

ББК 32.973.202

А 13

Печатается по решению Редакционно-издательского совета  
Института математики, механики и компьютерных наук им. И. И. Воровича  
Южного федерального университета (протокол № 4 от 1 сентября 2016 г.)

**Рецензенты:**

*доктор технических наук, профессор кафедры «Информатика»  
Ростовского государственного университета путей сообщения (РГУПС)*

**М. А. Бутакова**

*кандидат физико-математических наук, доцент кафедры алгебры  
и дискретной математики Института математики, механики  
и компьютерных наук им. И. И. Воровича Южного федерального университета*

**С. С. Михалкович**

**Абрамян М. Э.**

А 13 Структуры данных в PascalABC.NET. Выпуск 1. Массивы и  
последовательности. Запросы. — Ростов н/Д : Изд-во ЮФУ,  
2016. — 119 с.: ил.

ISBN

Учебное пособие содержит полное описание возможностей языка PascalABC.NET версии 3.1, связанных с динамическими массивами и последовательностями, включая обзор всех запросов для последовательностей, как входящих в стандартную библиотеку платформы .NET (в рамках интерфейса LINQ to Objects), так и разработанных специально для стандартной библиотеки PascalABC.NET. Особое внимание уделяется средствам PascalABC.NET, не имеющим прямых аналогов в стандартной библиотеке .NET, в частности, подпрограммам для генерации, ввода и вывода массивов и последовательностей. Детально обсуждаются особенности последовательностей как структур, выполняющих отложенную обработку данных. Изложение сопровождается многочисленными примерами, причем основная часть примеров представляет собой решения задач из электронного задачника Programming Taskbook, встроенного в систему PascalABC.NET. Пособие снабжено подробным указателем.

Для преподавателей программирования, старшеклассников и студентов.

УДК 004.438.NET

ББК 32.973.202

ISBN

© М. Э. Абрамян, 2016

## Предисловие

Предлагаемое учебное пособие является первым в серии пособий, посвященных структурам данных в языке PascalABC.NET. В нем дается подробное описание возможностей, связанных с использованием динамических массивов и последовательностей. В следующем выпуске [1] рассматриваются другие структуры данных, перенесенные в PascalABC.NET из стандартной библиотеки платформы .NET, а также многомерные структуры. В дальнейшем планируется продолжить серию, рассмотрев средства PascalABC.NET, связанные с обработкой текстовых строк и файловых данных.

Система программирования PascalABC.NET, разрабатываемая под руководством доцента С. С. Михалковича в Институте математики, механики и компьютерных наук Южного федерального университета, в настоящее время является одной из наиболее популярных систем, основанных на языке Паскаль, в русскоязычном образовательном пространстве. Язык PascalABC.NET, сохранив все базовые возможности таких распространенных реализаций Паскаля, как Delphi Pascal и Free Pascal, существенно расширил набор средств, доступных при разработке программ, позаимствовав наиболее востребованные (и не конфликтующие с идеологией Паскаля) возможности из языков C# и Python. Важнейшей особенностью языка PascalABC.NET является его интегрированность с платформой .NET 4.0, что позволяет использовать в нем все средства, включенные в ее огромную стандартную библиотеку. При этом многие из средств стандартной библиотеки .NET адаптированы в PascalABC.NET таким образом, чтобы их применение было более простым и «дружественным» для программиста, особенно начинающего; по этой же причине наиболее важные классы библиотеки .NET дополнены новыми полезными возможностями.

Хотя при разработке программ на PascalABC.NET можно ограничиться только традиционными средствами Паскаля (и в некоторых редких случаях, например, при подготовке к ЕГЭ по информатике, это может считаться оправданным), такой подход не позволяет программисту использовать всю мощь современных программных технологий, обеспечивающих как краткость и наглядность программного кода, так и его эффективность. Применение подобных технологий уже на начальных этапах обучения

представляется очень важным, так как позволяет сформировать навыки и «стиль мышления», характерные для современных языков программирования. Заметим, что именно такой подход реализован при обучении программированию в Воскресной компьютерной школе и затем на первом курсе направления «Фундаментальная информатика и информационные технологии» в Институте математики, механики и компьютерных наук ЮФУ.

В то же время практически отсутствуют книги, в которых давалось бы систематическое изложение языка PascalABC.NET с учетом всех его новых возможностей. Имеющиеся пособия начального уровня, например [6, 7], содержат лишь описание возможностей «традиционного» Паскаля. Часть важных нововведений языка используется в пособии [5], однако это пособие охватывает лишь базовый уровень и, кроме того, описывает одну из первых версий системы PascalABC.NET более чем 8-летней давности. Пособие [2], в котором излагается материал продвинутого уровня, использует язык PascalABC.NET наряду с другими современными реализациями Паскаля, но в этом пособии упор делается на возможностях, общих для разных реализаций. Разумеется, те возможности системы PascalABC.NET, которые перешли в нее из платформы .NET, могут быть изучены по соответствующим пособиям (например, [3, 4]), однако, во-первых, для этого придется освоить новый язык (как правило, C#) и, во-вторых, останутся неосвещенными многие важные дополнения PascalABC.NET к средствам стандартной библиотеки .NET.

Некоторым препятствием к разработке пособий, ориентированных на современные возможности PascalABC.NET, служит то обстоятельство, что этот язык очень интенсивно развивается и совершенствуется. В частности, полная поддержка технологии LINQ, обеспечивающей новую для Паскаля технологию разработки программ в стиле функционального программирования, была реализована всего год назад (в версии 3.0). И с этого времени язык уже пополнился такими принципиально новыми конструкциями, как кортежи, срезы и операторы `yield`.

Тем не менее, необходимость в создании подобных пособий уже назрела, и предлагаемая серия является первой такой попыткой. Структуры данных выбраны предметом изучения в этой серии не случайно. Именно в области обработки сложных наборов данных PascalABC.NET существенно превосходит традиционный Паскаль, объединяя расширения Delphi Pascal и стандартные средства платформы .NET и при этом дополняя их новыми возможностями.

Целью первого выпуска является максимально полное описание возможностей PascalABC.NET, связанных с динамическими массивами и последовательностями, включая обзор всех запросов для последовательностей, как входящих в стандартную библиотеку .NET (в рамках интерфейса

LINQ to Objects), так и разработанных специально для стандартной библиотеки PascalABC.NET. Особое внимание уделяется средствам PascalABC.NET, не имеющим прямых аналогов в стандартной библиотеке .NET, в частности, подпрограммам для генерации, ввода и вывода массивов и последовательностей. Детально обсуждаются особенности последовательностей как структур, выполняющих отложенную обработку данных, поскольку четкое понимание этих особенностей необходимо для правильного использования последовательностей в программах.

Изложение сопровождается многочисленными примерами, причем большая часть примеров представляет собой решения задач из электронного задачника Programming Taskbook (<http://ptaskbook.com>), встроенного в систему PascalABC.NET. Задачник автоматически генерирует тестовые наборы данных для каждого задания и проверяет правильность их обработки, поэтому его использование оказывается особенно удобным при изучении структур данных, избавляя учащегося как от ручного ввода или специальной генерации их элементов, так и от последующего анализа выведенных результатов. Кроме того, в задачнике предусмотрены дополнительные средства для ввода, вывода и отладочной печати, что существенно сокращает усилия учащихся при программировании и отладке программ с решениями задач.

Обширный набор задач, включенных в задачник, дает возможность читателю после ознакомления с решением какой-либо задачи закрепить изученные приемы, решая другие, аналогичные задачи из той же группы. Для большинства задач приводится несколько вариантов решения, что позволяет продемонстрировать многообразие средств PascalABC.NET и обсудить преимущества и недостатки каждого из использованных средств. В ряде случаев исследуется быстрдействие различных вариантов решения, при этом подробно описывается методика проведения подобных численных экспериментов. Приводимые в книге результаты, связанные с быстрдействием алгоритмов, были получены на компьютере со следующей конфигурацией (в скобках указывается индекс производительности, используемый Windows для оценки компонентов системы):

- процессор AMD A10-6700 3,70 GHz (7,3),
- оперативная память 6 Г (7,3),
- 64-разрядная операционная система Windows 7.

Пособие состоит из 5 глав. Первые две из них являются вводными. В главе 1 описываются основные нововведения PascalABC.NET, связанные с описанием и вводом-выводом данных, а также с применением таких конструкций языка, как лямбда-выражения и кортежи. Глава 2 посвящена знакомству с электронным задачником Programming Taskbook; кроме того, в ней на решениях задач из группы Proc демонстрируются средства, описанные в главе 1. Глава 3 содержит базовые сведения по работе с динамиче-

скими массивами и последовательностями, в главе 4 дается полное описание всех запросов для последовательностей, которое сопровождается решениями задач из группы задачника `LinqBegin`, а глава 5 посвящена дополнительным возможностям по обработке динамических массивов; в ней, как и в главе 3, рассматриваются задачи из группы `Array`.

Работа со статическими массивами в пособии рассматривается очень кратко; для более детального изучения особенностей обработки статических массивов можно обратиться, например, к [2]. Заметим, что книга [2] также содержит указания к задачам группы `Array`, которые могут оказаться полезными и при их решении с применением динамических массивов. Много дополнительных сведений, связанных с обработкой последовательностей, содержится в книгах [3, 4]; в частности, в [4] обсуждаются решения задач из группы `LinqObj`, позволяющие закрепить навыки по использованию сложных запросов на группировку и объединение данных.

При изложении материала предполагается, что читатель знаком с основами языка Паскаль, например, в объеме книги [5] (включая знания о базовых типах данных, управляющих операторах и процедурах и функциях). Знакомства с объектно-ориентированным программированием не требуется; достаточно понимать смысл использования точечной нотации для вызова методов.

Включенный в книгу указатель позволяет использовать ее как справочник по всем средствам работы с одномерными динамическими массивами и последовательностями в `PascalABC.NET`. Кроме того, указатель содержит ссылки на все разобранные в книге задачи и варианты их решения.

В книге описываются возможности системы `PascalABC.NET` по состоянию на **июль 2016 г. (версия 3.1)**. Скачать последнюю версию системы можно на ее сайте <http://pascalabc.net>.

## Глава 1. Некоторые расширения Паскаля в языке PascalABC.NET

При разработке программ на языке PascalABC.NET можно использовать синтаксис традиционных реализаций Паскаля (таких как Delphi Pascal или Free Pascal). Однако язык PascalABC.NET включает ряд новых возможностей, отсутствующих в традиционном Паскале. Эти возможности позволяют записывать программы более компактным и наглядным способом. В данной главе описываются наиболее существенные нововведения PascalABC.NET, без использования которых не обходится практически ни одна программа и которые будут активно применяться на протяжении всей книги.

### 1.1. Описание переменных

PascalABC.NET позволяет описывать переменные не только в специальном *разделе описаний*, но и в любом месте *раздела операторов*. Область действия такой переменной продолжается до окончания того блока, в котором она описана. Еще одной важной особенностью описания переменных в PascalABC.NET является *вывод типов* (type inference): если в момент описания переменная инициализируется некоторым значением, то тип переменной можно не указывать: он будет автоматически выведен компилятором из типа инициализирующего выражения. Таким образом, в любом месте программы можно, например, описать переменную `a`, инициализировав ее значением `0`, и в результате с этой переменной будет связан целочисленный тип `integer`:

```
var a := 0;
```

Преимуществом подобного способа описания является то, что требуемые переменные вводятся в программу непосредственно перед их использованием, причем наиболее краткая форма описания (с инициализирующим выражением) является одновременно и наиболее надежной, так как обеспечивает явную инициализацию описываемой переменной (заметим, что использование переменных без их явной инициализации очень часто приводит к ошибкам в программе).

Важным частным случаем подобного способа инициализации является инициализация параметра цикла `for`, например:

```
for var i := 1 to 10 do
  ...
```

Подобный вариант заголовка цикла описывает переменную `i`, тип которой выводится из инициализирующего выражения `1`, т. е. является целочисленным. Эта переменная будет существовать до конца того цикла, в котором она определена. Если попытаться обратиться к переменной `i` после выхода из цикла, то возникнет ошибка компиляции. Таким образом, подобная форма заголовка цикла `for` защищает программиста от очень распространенной ошибки — обращения к параметру цикла после его завершения, которая в традиционном Паскале не выявляется на этапе компиляции и в дальнейшем, при выполнении программы, обычно приводит к ее неверной работе.

Не следует считать, что описание вспомогательной переменной внутри блока, который выполняется многократно, приводит к неэффективной работе программы. Предположим, например, что переменная `a` описана в теле цикла:

```
for var i := 1 to 10 do
begin
  var a := 0;
  ...
end;
```

Это не означает, что на каждой итерации цикла будет выполняться последовательность действий, обеспечивающая выделение памяти для переменной `a` и ее инициализацию. Компилятор `PascalABC.NET` обеспечивает эффективное выделение памяти для всех используемых в программе переменных, независимо от области их видимости.

## 1.2. Ввод и вывод данных

Наряду с традиционной процедурой ввода `Read` язык `PascalABC.NET` позволяет использовать *функции ввода* для базовых типов данных: `ReadInteger`, `ReadReal`, `ReadChar`, `ReadString`, `ReadBoolean`. Применение этих функций позволяет совместить в одном операторе и описание переменной, и ввод ее значения, причем тип переменной можно не указывать, так как он *выводится* из типа возвращаемого значения функции. Например, для описания целочисленной переменной `a` и немедленного ввода ее значения с клавиатуры достаточно использовать единственный оператор:

```
var a := ReadInteger;
```



Предусмотрен вариант функций ввода со строковым параметром `prompt` — приглашением к вводу, которое выводится на экран при вызове функции ввода:

```
var a := ReadInteger('Введите длину отрезка:');
```

В традиционном Паскале для организации аналогичного ввода с приглашением потребовалось бы *три* оператора: описание переменной в разделе описаний, вывод приглашения с помощью процедуры `Write` и, наконец, ввод переменной с помощью процедуры `Read`.

Заметим также, что программист не должен предусматривать вывод пробела после приглашения; об этом заботится сама функция ввода.

Имеются модификации функций ввода, обеспечивающие немедленный переход к следующей строке вводимых данных; в этих вариантах, как и в соответствующем варианте процедуры `Read`, используется суффикс `ln` после слова `Read`, например `ReadlnInteger`.

Особое поведение реализовано для функции `ReadString`. Эта функция, в отличие от функций ввода других типов данных, обеспечивает автоматический переход к следующей строке вводимых данных (т. е. выполняется так же, как и функция `ReadlnString`). Подобное поведение функции `ReadString` защищает программиста от ошибок, возникающих при использовании процедуры `Read` вместо `Readln` для ввода *нескольких* строковых данных.

Кроме процедуры вывода `Write` и ее модификации `Writeln` в языке PascalABC.NET предусмотрены процедуры `Print` и `Println`, имеющие небольшое, но полезное отличие от традиционных аналогов: после вывода каждого элемента данных эти процедуры автоматически выводят символ пробела. Следует, однако, иметь в виду, что в процедурах `Print` и `Println` нельзя использовать атрибуты форматирования вида `:width` и `:width:digits`, доступные в процедурах `Write` и `Writeln`. Напомним, что запись `a:10` означает, что значение `a` будет выведено в 10 экранных позициях (и при необходимости дополнено *слева* пробелами), а запись `r:10:2`, допустимая только для вещественных данных, означает, что число `r` будет выведено в 10 экранных позициях с 2 дробными знаками.

Имеются также процедуры *форматного вывода* `WriteFormat` и `WritelnFormat`, обеспечивающие существенно более гибкие средства форматирования выводимых данных, чем описанные в предыдущем абзаце формирующие атрибуты традиционного Паскаля. В процедурах `WriteFormat` и `WritelnFormat` форматные настройки указываются в специальной *форматной строке*, которая является первым параметром этих процедур (после форматной строки можно указывать произвольное количество выводимых данных любого скалярного типа). Правила задания форматной строки ана-

логичны стандартным правилам, используемым в языках платформы .NET (см., например, [3, п. 4.3.3]). Приведем их краткое описание.

В форматной строке можно указывать обычный текст и *форматные настройки* для каждого из форматируемых параметров. Эти настройки заключаются в фигурные скобки и состоят из трех атрибутов, первый из которых является обязательным, а любой из двух последующих может отсутствовать: `{ind[,width][:spec]}` (для большей наглядности необязательные атрибуты заключены в полужирные квадратные скобки; разумеется, эти скобки *не следует* указывать в форматных настройках). Смысл атрибутов следующий:

- `ind` — целое число, которое определяет индекс форматируемого элемента в последующем списке выводимых данных (индексация ведется от 0);
- `width` — целое число, модуль которого задает минимальную *ширину поля вывода* (т. е. минимальное число позиций, отводимое для форматируемого элемента), а знак определяет *способ выравнивания* элемента в пределах поля вывода;
- `spec` — строка, которая задает *спецификатор формата* для данного элемента.

Между двоеточием и строкой `spec` не должно быть пробелов. Например, для вывода вещественного числа с 2 дробными знаками достаточно указать спецификатор формата `f2` (таким образом, если первый элемент выводимых данных является вещественным числом и его надо вывести в 10 экранных позициях с 2 дробными знаками, то соответствующая форматная настройка должна иметь вид `{0,10:f2}`). Если атрибут `spec` отсутствует, то выбирается вариант форматирования по умолчанию.

Если атрибут `width` отсутствует, то используется ширина поля вывода, минимально необходимая для отображения отформатированного элемента. Если отформатированный элемент не занимает всего поля вывода, то он дополняется пробелами: пробелы добавляются слева, если атрибут `width` положителен, и справа, если атрибут `width` отрицателен. Таким образом, при положительном значении `width` выполняется *выравнивание по правой границе* поля вывода, а при отрицательном — *по левой границе*.

Функции ввода и процедуры `Print/Println` можно использовать не только для ввода данных с клавиатуры и вывода на экран, но и для ввода-вывода данных, содержащихся в текстовых файлах. В этом случае первым параметром функции ввода или процедуры `Print/Println` должна быть файловая переменная типа `Text`. Кроме того, эти новые средства ввода-вывода можно применять при выполнении учебных заданий с применением электронного задачника `Programming Taskbook`, встроенного в среду `PascalABC.NET` (особенности использования задачника описываются в главе 2).

В PascalABC.NET предусмотрены также специальные средства для ввода и вывода массивов и последовательностей. Эти средства не имеют аналогов в традиционном Паскале; они будут описаны в главе 3, посвященной соответствующим структурам. Часть подобных средств можно использовать и при выполнении заданий из электронного задачника, связанных с обработкой структур данных.

### 1.3. Лямбда-выражения

Подобно большинству современных языков программирования, язык PascalABC.NET имеет средства для работы с *лямбда-выражениями* — создаваемыми «на лету» подпрограммами (функциями и процедурами), которые можно либо сохранить в процедурных переменных, либо (что требуется гораздо чаще) передать в качестве параметров процедурного типа<sup>1</sup>.

Синтаксис определения лямбда-выражений является чрезвычайно наглядным и кратким. Краткость достигается, прежде всего, за счет уже упоминавшегося ранее *вывода типов*, т. е. способности компилятора самостоятельно определять типы, связанные с определяемым лямбда-выражением. Аналогичный синтаксис можно использовать при описании процедурных переменных, что также существенно повышает наглядность.

Приведем пример. Опишем процедурную переменную *f* для хранения вещественных функций с одним вещественным параметром:

```
var f: real -> real;
```

После этого с переменной *f* можно связать лямбда-выражение, определяющее функцию возведения в куб:

```
f := x -> x * x * x;
```

Обратите внимание на то, что после «стрелки» *->* сразу указывается возвращаемое значение. Тип параметра *x* указывать не требуется, так как он может быть выведен из описания переменной *f*.

Операторы описания и инициализации можно объединить:

```
var f: real -> real := x -> x * x * x;
```

Теперь можно выполнять операцию возведения в куб, используя процедурную переменную *f*, например:

```
Print(f(1), f(2), f(3)); // будет выведено 1 8 27
```

Если процедурная переменная или лямбда-выражение имеют несколько параметров, то их, как обычно, надо заключать в скобки. Если проце-

---

<sup>1</sup> В дальнейшем под словом «подпрограмма» будем подразумевать как функцию, так и процедуру (аналогичным образом, выражение «процедурный тип» традиционно используется для типов, определяющих как процедуры, так и функции).

дурная переменная предназначена для хранения процедур (а не функций), то при ее описании после стрелки  $\rightarrow$  надо указать парные скобки  $()$ . Например, так выглядит определение процедурной переменной, с которой связывается лямбда-выражение для печати суммы двух целых чисел:

```
var p: (integer, integer) -> () := (a, b) -> Print(a + b);
p(2, 3); // будет выведено 5
```

Как правило, лямбда-выражения используются для оформления коротких функций или процедур, состоящих из одного возвращаемого выражения или, соответственно, одного выполняемого оператора. Однако ничто не препятствует описывать лямбда-выражения из нескольких операторов; в этом случае лишь требуется, как обычно, заключать эти операторы в операторные скобки `begin-end`, а для определения возвращаемого значения использовать стандартную переменную `Result`. Например, с переменной `f` можно связать следующее лямбда-выражение, которое не только вычисляет куб числа, но и печатает результат вычислений на экране:

```
f := x ->
begin
  var y := x * x * x;
  WritelnFormat('{0}*{0}*{0}={1}', x, y);
  Result := y;
end;
```

Данный фрагмент содержит также пример использования форматирующей процедуры вывода `WritelnFormat` с простейшим вариантом форматной строки, в которой позиции выводимых данных помечаются как `{0}` и `{1}` (обратите внимание на то, что значение `x` будет указано в трех местах форматной строки).

Теперь вызов оператора

```
Print(f(1), f(2), f(3));
```

приведет к выводу следующего текста:

```
1*1*1=1
2*2*2=8
3*3*3=27
1 8 27
```

Здесь первые три строки выводятся внутри вызванных функций `f`, а возвращаемые ими значения печатаются в последней строке (которая выводится процедурой `Print`).

Следует упомянуть еще одну важную особенность лямбда-выражений: возможность *захвата переменных*. В лямбда-выражении можно использовать не только параметры, но и внешние по отношению к этому выражению переменные. Это часто оказывается удобным в ситуации, когда лямб-

да-выражение требуется передавать в качестве параметра. Предположим, например, что имеется процедура `Tab`, которая выводит значения некоторой вещественной функции  $f$ , зависящей от одной вещественной переменной, в точках отрезка  $[a, b]$ , разбивающих его на  $n$  равных частей (такое действие называется *табуляцией* функции  $f$ ). Естественно ожидать, что ее заголовок будет иметь следующий вид (вариант реализации этой функции будет приведен далее, в п. 3.5):

```
procedure Tab(f: real -> real; a, b: real; n: integer);
```

Предположим далее, что мы хотим выполнить табуляцию функции, зависящей от дополнительного параметра (одного или нескольких), — например, квадратного трехчлена  $f(x) = ax^2 + bx + c$ . Мы не можем использовать лямбда-выражение, зависящее от четырех переменных ( $x, a, b, c$ ), так как процедура `Tab` принимает только лямбда-выражения типа `real -> real`. Однако мы можем описать требуемое лямбда-выражение с параметром  $x$ , «захватив» в нем внешние переменные  $a, b, c$ :

```
var a, b, c: real;
// при создании переменные a, b, c получают нулевые значения
var f : real -> real;
f := x -> a * x * x + b * x + c;
a := 1;
Tab(f, 0, 1, 10); // табуляция функции x * x
b := -2;
c := 1;
Tab(f, 0, 1, 10); // табуляция функции (x - 1) * (x - 1)
```

Необходимо подчеркнуть, что при вычислении функции  $f$  будут использоваться те значения захваченных переменных, которые они имеют в *момент вычисления* функции (а не в момент ее определения).

## 1.4. Кортежи

Подобно тому как лямбда-выражения позволяют «на лету» создавать функции или процедуры для передачи их в подпрограммы или присваивания процедурным переменным, *кортежи* (tuples) позволяют «на лету» создавать составные наборы данных, аналогичные *записям* (records), используя при этом очень краткий и наглядный синтаксис.

Например, кортеж `t`, состоящий из двух полей, первое из которых имеет целый, а второе — строковый тип, можно описать следующим образом:

```
var t: (integer, string);
```

После этого переменной `t` можно присвоить значение, состоящее из полей указанных типов, также заключая эти поля в круглые скобки:

```
t := (10, 'Sample');
```

Поскольку по типу каждого выражения в правой части присваивания можно *вывести* тип кортежа, инициализацию переменной `t` можно выполнить в момент ее описания, не указывая при этом ее тип:

```
var t := (10, 'Sample');
```

Полям кортежа автоматически присваиваются имена `Item1`, `Item2` и т. д.; кроме того, с ними связываются индексы, начиная от 0. Таким образом, для обращения к полям кортежа можно использовать либо точечную нотацию (например, `t.Item1`), либо индексированное выражение (например, `t[0]`). Однако индекс должен быть константным выражением (это необходимо для того, чтобы компилятор уже на этапе компиляции смог определить тип того поля, к которому выполняется обращение).

Кортеж может содержать от 1 до 7 полей, причем поля тоже могут быть кортежами.

Поля кортежа доступны только для чтения; изменять поля по отдельности запрещено. Можно представлять себе кортеж как некоторую «замороженную» сущность, которую нельзя изменять по частям. Однако изменять кортеж как единое целое вполне допустимо; при этом для изменения, например, только первого из двух имеющихся полей можно использовать следующий оператор:

```
t := (20, t.Item2);
```

Важным средством работы с кортежами является так называемое *кортежное присваивание*. Если в правой части оператора присваивания указывается кортеж, то в левой можно указать *в круглых скобках через запятую* набор переменных, типы которых совпадают с типами полей кортежа, например (предполагаем, что переменная `n` имеет тип `integer`, а переменная `s` — тип `string`):

```
(n, s) := t;
```

В результате подобного присваивания в переменные, указанные в левой части, будут записаны значения соответствующих полей кортежа, указанного справа.

Интересным примером применения кортежного присваивания является *обмен значениями* двух одностипных переменных `a` и `b`, который можно оформить в виде единственного оператора:

```
(a, b) := (b, a);
```

Кортежи позволяют наглядно записывать лямбда-выражения, возвращающие не одно, а несколько значений. Например, допустимо описать процедурную переменную такого типа:

```
var f: (integer, integer) -> (integer, integer);
```

Затем эту переменную можно связать с лямбда-выражением, которое принимает два целых числа и возвращает их сумму и произведение:

```
f := (a, b) -> (a + b, a * b);
```

Стандартные процедуры вывода PascalABC.NET Write/Writeln и Print/Println позволяют указывать кортежи в качестве своих параметров. Поля кортежа при выводе разделяются запятыми, а весь кортеж заключается в круглые скобки. Например, можно выполнить в процедуре вывода вызов ранее определенного лямбда-выражения f:

```
Write(f(10, 20));
```

В результате будет выведен следующий текст:

```
(30,200)
```


Два кортежа считаются имеющими один и тот же тип, если они имеют одинаковое количество полей, причем поля с одинаковыми индексами имеют одинаковые типы. Кортежи одного типа можно сравнивать не только операциями «равно»—«не равно», но и операциями «меньше»—«больше»; при этом используется так называемое *лексикографическое* сравнение: вначале сравниваются значения первого поля; если они различны, то меньшим кортежем считается тот, у которого первое поле имеет меньшее значение; если первые поля совпадают, то сравниваются вторые поля, и т. д.


Для кортежей предусмотрен метод Add. Вызов a.Add(b) возвращает кортеж, содержащий все поля кортежа a и дополнительное поле, которому присваивается значение b (новое поле располагается последним). Того же самого результата можно достигнуть, используя *операцию* + вида a + b (где первый операнд является кортежем). Таким образом, выражение (1, 2) + 3 вернет кортеж (1, 2, 3). Заметим, что выражение (1, 2) + (3, 4) тоже вернет кортеж из трех полей (1, 2, (3, 4)), поскольку второй операнд операции + (и параметр b при вызове метода a.Add(b)) всегда добавляется к кортежу в виде *единственного* поля.

## Глава 2. Электронный задачник Programming Taskbook


На протяжении всей книги изучаемые возможности языка PascalABC.NET будут иллюстрироваться примерами решений задач, входящих в электронный задачник Programming Taskbook. Это позволит приводить краткие, но законченные фрагменты программного кода, которые можно сразу запускать на выполнение. При этом исходные данные для обработки будут предоставляться самим задачиком, и он же будет автоматически проверять правильность полученных результатов. Кроме того, все используемые в программе данные будут в наглядном виде отображаться в окне задачника. В настоящей главе подробно описываются все этапы решения задачи в системе PascalABC.NET с применением электронного задачника. В качестве примера выбрана задача, решение которой позволит проиллюстрировать применение большинства особенностей языка PascalABC.NET, рассмотренных в предыдущей главе.

### 2.1. Создание заготовки для выбранного задания

Процесс решения задачи из электронного задачника Programming Taskbook начинается с создания программы-заготовки для этой задачи. Хотя программа-заготовка имеет очень простой вид и ее можно создать вручную, удобнее воспользоваться для этих целей специальной утилитой задачника PT4Load. Эта утилита запускается непосредственно из среды PascalABC.NET с помощью либо команды меню «Модули | Создать шаблон программы», либо кнопки быстрого доступа  на панели инструментов, либо (самый быстрый способ) клавиатурной комбинации Ctrl+Shift+L. В результате на экране появится окно утилиты PT4Load, в котором требуется ввести имя задания (см. рис. 1).

Как правило, для решения задач используется специальный рабочий каталог C:\PABCWork.NET, в котором автоматически размещается файл результатов с данными об учащемся. Имя рабочего каталога и данные учащегося (обычно фамилия и имя) отображаются в окне программы PT4Load. При желании можно изменить как рабочий каталог, так и данные учащегося. Для смены каталога можно использовать либо кнопку , ли-



бо клавишу F12. Для изменения данных надо использовать команду контекстного меню «Изменить данные в файле результатов»; отобразить контекстное меню можно, либо выполнив щелчок правой кнопкой мыши в окне программы, либо нажав кнопку , либо используя комбинацию клавиш Shift+F10. Следует иметь в виду, что *при изменении данных учащегося в файле результатов все прежнее содержимое файла результатов стирается.*

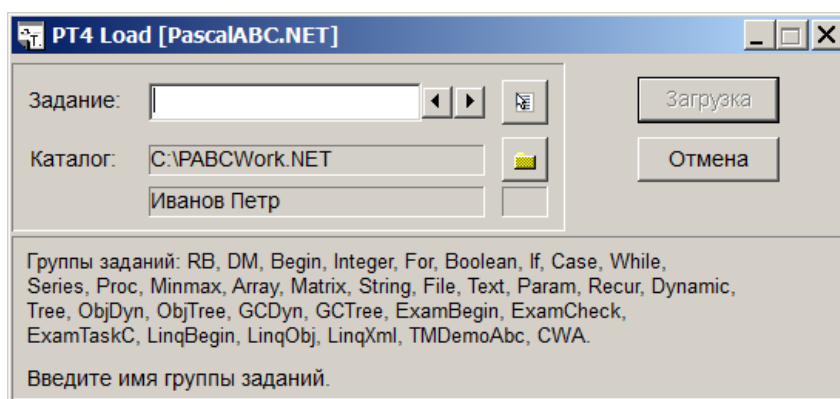


Рис. 1. Окно утилиты PT4Load

В нижней части окна PT4Load выводится список доступных групп с заданиями. Если ввести имя одной из групп, то в окне будет выведено краткое описание выбранной группы и количество входящих в нее задач. После ввода одного из допустимых номеров задач кнопка «Загрузка» станет доступной (см. рис. 2); щелкнув на этой кнопке или нажав клавишу Enter, мы создадим заготовку для выбранной задачи.

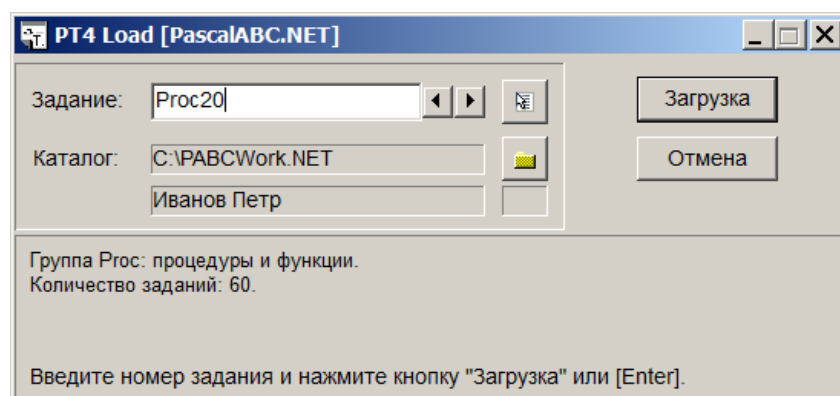


Рис. 2. Окно утилиты PT4Load после ввода имени задания

Мы выбрали задачу **Proc20** — одну из задач группы Proc, посвященной процедурам и функциям, их описанию и использованию. Созданная заготовка будет иметь имя, совпадающее с именем задания (в нашем случае Proc20.pas). После создания она сразу загрузится в редактор среды PascalABC.NET.

Приведем вид заготовки для задачи Proc20:

```

uses PT4;

begin
  Task('Proc20');

end.

```

В первой строке заготовки к программе подключается модуль PT4, который содержит вспомогательные процедуры и функции задачника, предназначенные для инициализации задания, ввода исходных данных, вывода результатов и отображения на экране отладочной информации. Затем следует раздел операторов, содержащий вызов единственной процедуры Task, которая выполняет инициализацию указанного задания.

## 2.2. Окно задачника

Запустив созданную заготовку (самый быстрый способ это сделать — нажать клавишу F9), мы увидим на экране окно задачника с данными, соответствующими выбранной задаче (см. рис. 3). Это окно содержит формулировку задачи, набор тестовых исходных данных, а также пример правильного решения для этих данных. Данные могут выводиться на белом или на черном фоне; для изменения фонового цвета достаточно нажать клавишу F3 или щелкнуть на метке «Цвет» в правой верхней части окна.

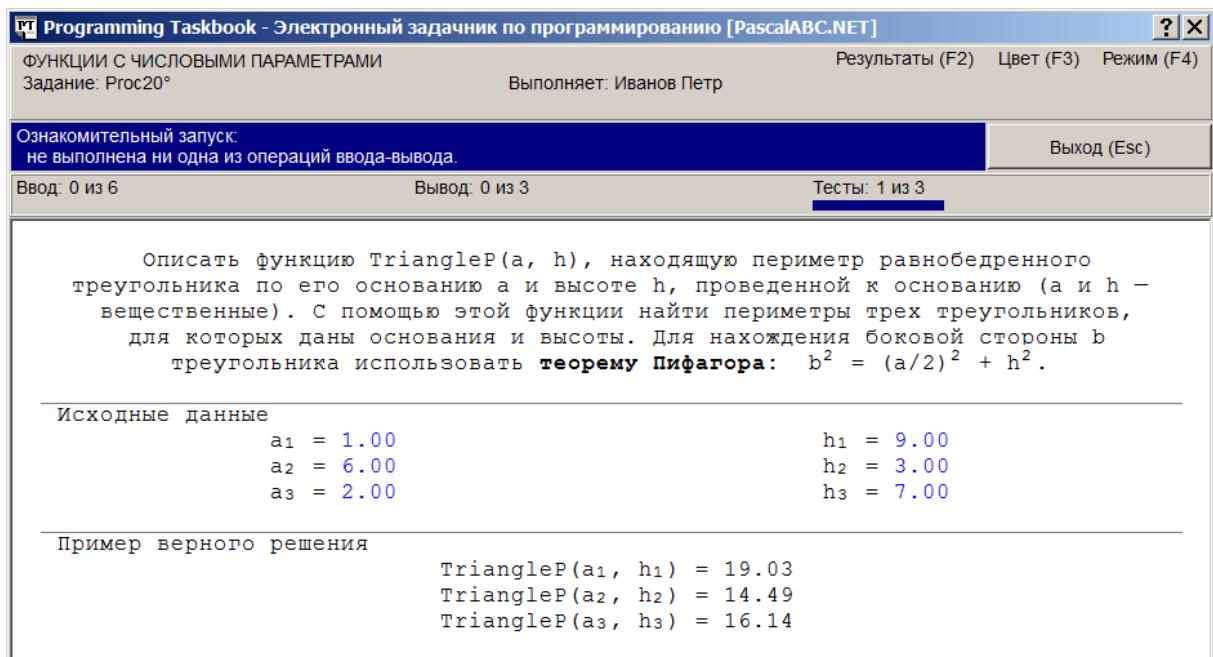


Рис. 3. Ознакомительный запуск задания Proc20

Верхняя часть окна содержит *информационную панель* с описанием результата запуска программы и *панель индикаторов*, в которой указывается число введенных исходных данных и выведенных результатов, а так-

же количество пройденных тестов. В данном случае информационная панель содержит текст «*Ознакомительный запуск: ни выполнена ни одна из операций ввода-вывода*».

Ознакомившись с условиями задачи, мы можем закрыть окно, щелкнув на кнопке «Выход (Esc)» или нажав клавишу Esc. Для завершения программы можно также нажать клавишу F9, т. е. ту же клавишу, которая использовалась для запуска программы.

### 2.3. Решение задачи

Решение задачи надо ввести в заготовку после процедуры Task. Разумеется, мы можем описывать переменные (а также константы, типы, процедуры и функции) в традиционном для Паскаля разделе описаний, однако в данной книге мы будем активно пользоваться возможностью PascalABC.NET, разрешающей описывать переменные непосредственно в разделе операторов (см. п. 1.1). Поскольку наши решения всегда будут целиком располагаться в разделе операторов, договоримся для краткости приводить только содержимое этого раздела, опуская обязательную директиву uses подключения модуля PT4, а также обрамляющие раздел операторов операторные скобки begin-end.

Начнем с реализации функции TriangleP. Чтобы проиллюстрировать еще одну рассмотренную ранее особенность PascalABC.NET, связанную с лямбда-выражениями (см. п. 1.3), реализуем эту функцию в виде процедурной переменной, присвоив ей соответствующее лямбда-выражение:

```
Task('Proc20');  
var TriangleP: (real, real) -> real;  
TriangleP := (a, h) -> 2 * Sqrt(Sqr(a / 2) + Sqr(h)) + a;
```

Вначале мы описали процедурную переменную TriangleP, определив ее тип как функцию, принимающую два вещественных параметра и возвращающую вещественный результат. Затем мы присвоили этой переменной лямбда-выражение, которое по двум параметрам  $a$  и  $h$  — основанию и высоте равнобедренного треугольника — вычисляет периметр этого треугольника. Для нахождения боковой стороны мы использовали формулу, приведенную в условии задачи, а для операций возведения в квадрат и извлечения квадратного корня — стандартные функции Паскаля Sqr и Sqrt.

Чтобы проверить, что данный фрагмент программы не содержит ошибок, запустим программу еще раз. Поскольку мы по-прежнему не вводим и не выводим данные, текст информационной панели не изменится, хотя набор исходных данных окажется другим. При каждом запуске программы задачник генерирует новый набор тестовых данных (используя при этом датчик случайных чисел), поэтому для решения задачи необходимо запрограммировать алгоритм, правильно обрабатывающий *любые* допустимые

наборы исходных данных. Заметим, что при компиляции программы будет выведено предупреждение: «Переменной 'TriangleP' присвоено значение, но оно нигде далее не используется». Подобные предупреждения не препятствуют компиляции и запуску программы, однако обычно свидетельствуют о каких-либо недочетах в ее реализации.

Нам осталось проверить правильность полученной функции, вычислив с ее помощью периметры трех треугольников, основания и высоты которых будут предложены задачиком. Для получения исходных данных будем использовать *функции* ввода (см. п. 1.2). Результат, возвращенный функцией TriangleP, мы можем сразу переслать задачику для проверки, используя одну из процедур вывода (допустимо использовать как Write, так и Print, а также их модификации Writeln и Println; при выполнении заданий с применением задачника все эти варианты процедур вывода работают одинаково). Приведем фрагмент, который надо добавить в конец предыдущего варианта решения:

```
var a := ReadReal;  
var h := ReadReal;  
Write(TriangleP(a, h));
```

**Замечание.** Может возникнуть вопрос, почему вместо приведенного фрагмента не был использован *единственный* оператор, в котором ввод выполняется при вызове функции: Write(TriangleP(ReadReal, ReadReal)). Причина заключается в том, что в PascalABC.NET *не определен порядок вычисления параметров подпрограмм*, и значит, нельзя быть уверенным в том, что в данном случае *вначале* будет вызвана первая функция ReadReal, а *затем* вторая. В случае обработки параметров в направлении *справа налево* в приведенном операторе основание *a* окажется вторым параметром функции TriangleP, а высота *h* — первым, что, разумеется, приведет к неверному вычислению периметра. Проверка показывает, что в версии 3.1 системы PascalABC.NET параметры вычисляются *слева направо* (и поэтому в данной версии значения параметров будут располагаться в правильном порядке), однако это может измениться в последующих версиях.

Полученный вариант решения является не вполне правильным, так как он обрабатывает не три треугольника, а лишь один. Поэтому при запуске программы в информационной панели окна задачника будет выведено сообщение об ошибке «Введены не все требуемые исходные данные. Количество прочитанных данных: 2 (из 6)», причем фон информационной панели станет оранжевым (см. рис. 4). Оранжевый цвет используется для индикации ошибки ввода, связанной с тем, что были введены не все требуемые данные. Этот же цвет используется в случае вывода не всех требуемых результатов. При попытке ввести или вывести *лишние* данные сообщение об ошибке выводится на малиновом фоне, а ошибки, связанные с использова-

нием *неверных типов* при вводе или выводе, выделяются фиолетовым цветом. Все прочие ошибки отображаются на красном фоне.

На панели индикаторов отображаются данные о количестве фактически введенных и выведенных данных. Заголовок раздела с исходными данными также выделяется оранжевым цветом, чтобы подчеркнуть, что ошибка связана именно с вводом данных. Следует также обратить внимание на то, что в случае ошибочного решения в окне задачника отображается как раздел с результатами, полученными программой, так и раздел с примером правильного решения.

Сравнивая содержимое этих разделов, мы можем убедиться, что периметр первого из данных треугольников найден правильно.

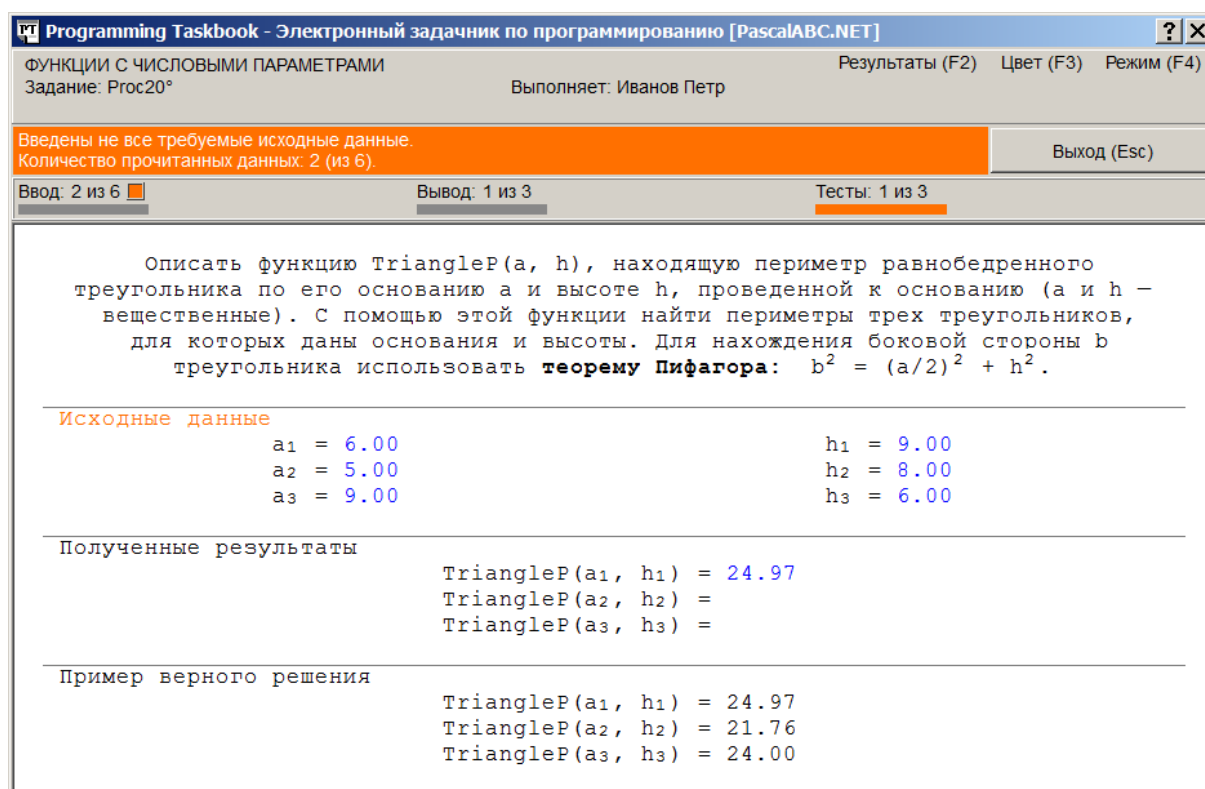


Рис. 4. Ошибочное решение задачи Proc20

Для завершения программы нам осталось повторить три раза действия по вводу данных, их передаче функции TriangleP и выводу результата. Оформим это повторение в виде цикла for, воспользовавшись еще одной возможностью PascalABC.NET, отсутствующей в традиционном Паскале: описанием параметра цикла непосредственно в его заголовке (см. п. 1.1). В результате решение задачи примет следующий вид:

```
Task('Proc20');
var TriangleP: (real, real) -> real;
TriangleP := (a, h) -> 2 * Sqrt(Sqr(a/2) + Sqr(h)) + a;
for var i := 1 to 3 do
```

```

begin
  var a := ReadReal;
  var h := ReadReal;
  Write(TriangleP(a, h));
end;

```

При запуске этого (правильного) варианта решения на экране появится вспомогательное окно с индикацией пройденных тестов (см. рис. 5), а после успешного прохождения требуемого числа тестов (в зависимости от сложности задачи и разнообразия возможных исходных данных это число может изменяться от 3 до 9) — окно задачника с сообщением о правильном решении (см. рис. 6).

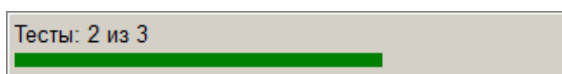


Рис. 5. Индикация успешно пройденных тестов

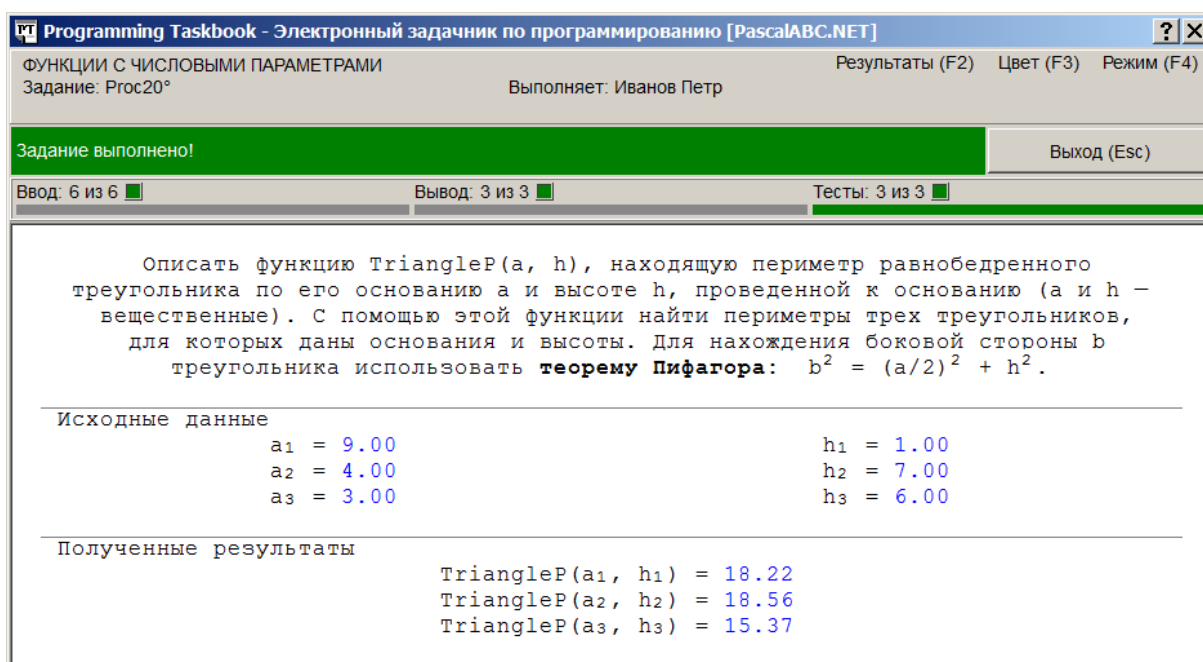



Рис. 6. Правильное решение задачи Proc20

В данном случае раздел с примером правильного решения отсутствует, так как его содержимое совпадает с содержимым раздела с полученными результатами.

Щелкнув на метке «Результаты (F2)» в правом верхнем углу окна или нажав клавишу F2, можно просмотреть протокол выполнения задания, записанный в файле результатов. Для просмотра содержимого файла результатов используется специальная утилита PT4Results, которую можно вызвать и непосредственно из среды PascalABC.NET с помощью команды меню «Модули | Просмотреть результаты», кнопки  или клавиатурной

комбинации Ctrl+Shift+R. В окне этой программы (в разделе «Полная информация») мы увидим примерно такой текст:

```
= Иванов Петр          (C:\PABCWork.NET)
Proc20  A03/05 15:34 Ознакомительный запуск.--2
Proc20  A03/05 15:40 Введены не все требуемые исходные данные.
Proc20  A03/05 15:47 Задание выполнено!
```

Каждая строка содержит название задания, дату и время запуска программы и описание результата ее выполнения. Число «2» в конце первой строки с результатами означает, что было произведено два ознакомительных запуска (время соответствует последнему из них). Буква «A» перед датой означает, что задание выполнялось в среде PascalABC.NET.

В заключение данного пункта приведем пример решения еще одной задачи из группы Proc, которая похожа на только что решенную, но имеет особенности, позволяющие использовать при ее решении не только лямбда-выражения, но и кортежи (см. п. 1.4). Это задача **Proc4**; в ней требуется описать процедуру TrianglePS, которая по стороне *a* равностороннего треугольника вычисляет его периметр и площадь. Для проверки правильности процедуры надо обработать с ее помощью три данных треугольника.

При использовании традиционных средств Паскаля в разделе описаний необходимо описать процедуру, содержащую два выходных параметра (снабженных атрибутом var):

```
procedure TrianglePS(a: real; var P, S: real);
begin
  P := 3 * a;
  S := Sqrt(3) * a * a / 4;
end;
```

Применение процедуры для решения задачи будет выглядеть следующим образом (напомним, что мы для краткости не указываем операторные скобки begin-end, обрамляющие раздел операторов):

```
Task('Proc4'); // Вариант 1
var P, S: real;
for var i := 1 to 3 do
begin
  TrianglePS(ReadReal, P, S);
  Write(P, S);
end;
```

Однако аналогичной функциональности можно добиться, оформив требуемый алгоритм в виде *функции* (лямбда-выражения), принимающей один параметр (сторону треугольника) и возвращающей кортеж из двух найденных значений — площади и периметра:

```
Task('Proc4'); // Вариант 2
```

```
var TrianglePS: real -> (real, real);
TrianglePS := a -> (3 * a, Sqrt(3) * a * a / 4);
var P, S: real;
for var i := 1 to 3 do
begin
  (P, S) := TrianglePS(ReadReal);
  Write(P, S);
end;
```

Для сохранения результатов, возвращенных функцией, мы использовали кортежное присваивание. Поместить вызов функции `TrianglePS` непосредственно в процедуру вывода `Write` нельзя, так как варианты процедур вывода `Write` и `Print` для электронного задачника не поддерживают вывод кортежей (и других структур данных).

Мы видим, что завершающий фрагмент второго варианта решения (в котором выполняется тестирование функции) отличается от аналогичного фрагмента первого варианта (в котором тестируется процедура) единственным оператором, содержащим вызов соответствующей функции (процедуры). При этом оператор с вызовом функции оказывается более наглядным, так как по его виду, в отличие от оператора вызова процедуры, можно сразу определить, что переменные `P` и `S` будут содержать *результат* выполнения соответствующей подпрограммы.

## 2.4. Отладочные средства задачника

При решении задач часто бывает желательно вывести на экран дополнительную информацию, связанную с ходом решения. Поскольку задачник рассматривает все данные, выводимые процедурами `Write` и `Print`, как результаты, полученные программой, попытка вывести с их помощью какие-либо дополнительные данные приведет к сообщению об ошибочном решении. Кроме того, эти дополнительные данные могут вообще не появиться на экране, так как задачник отображает в разделе результатов только те данные, которые соответствуют типу ожидаемых результатов.

Для возможности вывода отладочной информации в задачнике предусмотрены специальные процедуры `Show` и `ShowLine`. Каждая из них выводит в особый *раздел отладки* требуемый элемент данных (снабженный, при необходимости, строкой-комментарием), а процедура `ShowLine` дополнительно осуществляет переход на новую строку в разделе отладки. Раздел отладки отображается в окне задачника только в случае, если он содержит какие-либо данные.

На примере ранее решенной задачи **Proc20** проиллюстрируем использование отладочных процедур, добавив их вызов и в разработанную функцию `TriangleP`, и в цикл, в котором эта функция вызывается. Заодно мы продемонстрируем еще раз, как описывать лямбда-выражение, тело кото-



рого состоит из нескольких операторов (ранее подобный пример мы приводили в конце п. 1.3):

```
Task('Proc20');
var TriangleP: (real, real) -> real;
TriangleP := (a, h) ->
begin
  Show('a = ', a);
  Show('h = ', h);
  Result := 2 * Sqrt(Sqr(a / 2) + Sqr(h)) + a;
  ShowLine('P = ', Result);
end;
loop 3 do
begin
  Show(i);
  var (a,h) := ReadReal2;
  Write(TriangleP(a, h));
end;
```

В этом варианте на каждой итерации цикла в раздел отладки выводится номер итерации (значение параметра цикла *i*); кроме того, функция `TriangleP` выводит в раздел отладки значения своих параметров, снабженные комментариями, а также значение результата — найденного периметра, после чего переходит в разделе отладки на новую строку.

При запуске программы на экране появится окно задачника, содержащее раздел отладки (см. рис. 7). Как мы видим, вывод отладочных данных никак не повлиял на проверку полученных результатов.

Вещественные данные отображаются в окне отладки с двумя дробными знаками (как и в других разделах окна задачника). Число дробных знаков *n* можно указать явно, используя процедуру `SetPrecision(n)`. При *n* = 0 используется формат с плавающей точкой.

В задачнике предусмотрены дополнительные возможности, связанные с отладочной печатью *последовательностей* (последовательности рассматриваются в п. 3.3). Эти возможности будут описаны далее, при обсуждении заданий группы `LinqBegin` (см. п. 4.2).

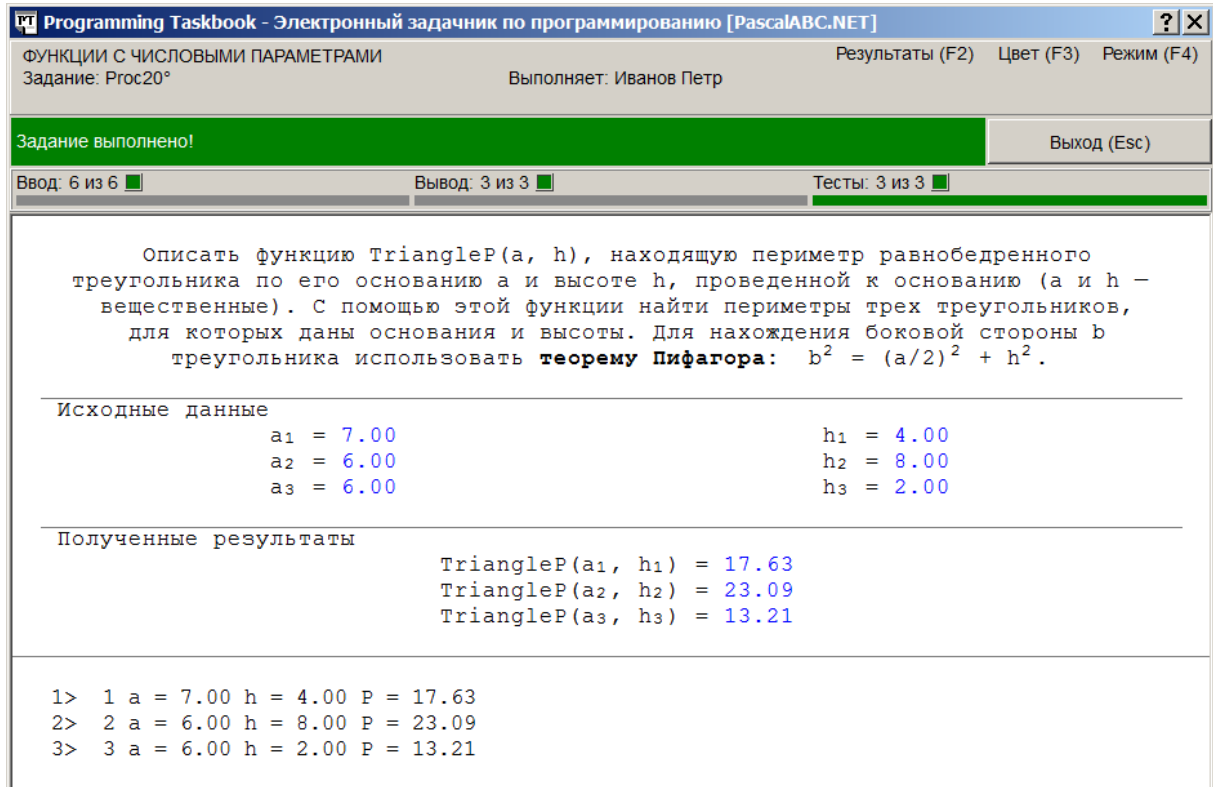


Рис. 7. Окно задачника с разделом отладки

Краткое описание средств отладки задачника содержится в разделе «Отладка» окна справки (см. рис. 8), которое можно вызвать из окна задачника, щелкнув на кнопке со знаком «?» в заголовке окна или нажав клавишу F1.

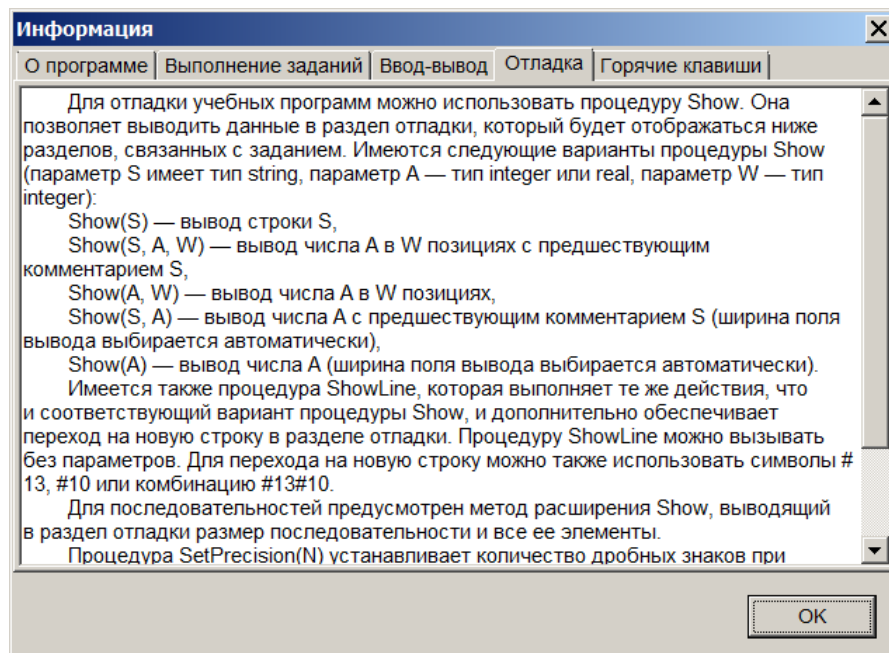


Рис. 8. Окно со справочной информацией

## Глава 3. Массивы и последовательности

### 3.1. Статические и динамические массивы в PascalABC.NET

Массивы представляют собой структурный тип данных, состоящий из элементов *одного и того же типа*, размещенных на *непрерывном* участке памяти. Доступ к требуемому элементу массива осуществляется по его *индексу*, который указывается в квадратных скобках после имени массива, например `a[5]`. Массивы обеспечивают очень быстрый доступ к своим элементам, так как по адресу начала массива и значению индекса можно легко вычислить адрес, по которому располагается в памяти элемент с указанным индексом.

В современных реализациях языка Паскаль, в том числе и в PascalABC.NET, можно использовать статические и динамические массивы. *Статические массивы* — это традиционные массивы Паскаля, в которых можно указывать диапазон изменения индексов, причем в качестве индексов допускается использовать любой перечислимый тип (например, символьный). Приведем три примера описания статических массивов:

```
var a: array[0..19] of string;  
    b: array['A'..'Z'] of integer;  
    c: array[1..10,1..20] of real;
```

Массив `a` является одномерным массивом строк. Он содержит 20 элементов, которые индексируются числами от 0 до 19. Массив `b` содержит целые числа и индексируется символами — заглавными латинскими буквами. Таким образом, он состоит из 26 элементов; его первый элемент имеет индекс 'A', а последний — индекс 'Z'. Массив `c` является двумерным массивом; каждый его элемент задается парой индексов, причем первый индекс может изменяться в диапазоне от 1 до 10, а второй — в диапазоне от 1 до 20. Всего в массиве `c` содержится 200 ( $= 10 \cdot 20$ ) элементов. Двумерные массивы часто интерпретируются как прямоугольные таблицы элементов, причем номера *строк* определяются первым индексом, а номера *столбцов* — вторым. Например, в нашем случае удобно считать, что элемент `c[4,12]` находится на пересечении 4 строки и 12 столбца.

Основным недостатком статических массивов является то, что их размер нельзя задать с учетом текущих обрабатываемых данных. В качестве диапазона индексов необходимо указывать числа или именованные константы. Поэтому приходится описывать массивы «по максимуму», выделяя для них столько памяти, сколько может потребоваться для хранения самой большой из возможных наборов исходных данных. Помимо очевидной неэффективности такого подхода он порождает еще одну проблему: как правило, статические массивы заполняются не полностью, и поэтому требуется дополнительно хранить информацию о «фактической заполненности» статического массива. Следует также отметить бедность средств обработки статических массивов в Паскале. Всё, что предоставляет программисту Паскаль в отношении статического массива, — это возможность его описывать и обращаться к его элементам по индексу. Все алгоритмы обработки массивов программист вынужден реализовывать самостоятельно (или подключать дополнительные библиотеки с этими алгоритмами).

*Динамические массивы* являются гораздо более гибкими. Главной особенностью динамических массивов является возможность выделения для них памяти *на любом этапе* выполнения программы, причем размер выделяемой памяти может определяться текущими значениями обрабатываемых данных. Таким образом, используя динамические массивы, не нужно заранее ограничивать их возможный размер; кроме того, при создании динамического массива можно выделить ровно столько памяти, сколько требуется для конкретного набора данных, поэтому не требуется дополнительно отслеживать уровень «заполненности» массива. Имеется возможность корректировки размера динамического массива как в сторону его уменьшения, так и в сторону увеличения, причем в откорректированном массиве будут сохранены значения всех оставшихся элементов (эта возможность будет подробно рассмотрена в п. 5.2).

Впервые такой вид массивов появился в языке Delphi Pascal. Язык PascalABC.NET поддерживает синтаксис динамических массивов Delphi Pascal, но вместе с тем предоставляет для них альтернативный синтаксис в стиле языка C#, обладающий большей наглядностью. Стандартная библиотека языка PascalABC.NET содержит большое количество подпрограмм, связанных с динамическими массивами, что существенно упрощает их обработку.

### 3.2. Создание массивов и их вывод

Здесь и далее для краткости под *массивом* (без уточняющего прилагательного «статический» или «динамический») мы будем всегда подразумевать *динамический* массив. В данной книге рассматриваются только *одномерные* массивы (особенности многомерных массивов описываются во втором выпуске настоящей серии — см. [1, гл. 6]).

Для *описания* одномерного массива достаточно указать ключевые слова `array of` и имя типа элементов массива, например:

```
var a: array of integer;
```

При этом, в отличие от описания статических массивов, не указывается диапазон индексов, поскольку на этапе описания динамического массива память для его элементов еще не выделяется.

Аналогичным образом массивы могут описываться в качестве параметров подпрограмм, например:

```
procedure p(a: array of integer);
```

Для возможности обращения к элементам массива его необходимо *инициализировать*. Именно на этапе инициализации происходит выделение памяти для элементов массива и ее заполнение начальными значениями. Инициализирующее выражение для массива начинается со слова `new`, используемого в `PascalABC.NET` для вызова *конструктора* некоторого класса (фактически при инициализации массива вызывается конструктор класса `Array` из стандартной библиотеки платформы `.NET`; дополнительные сведения о классе `Array` приведены в [1, гл. 8]). После слова `new` указывается тип элементов массива и, в квадратных скобках, их количество:

```
a := new integer[10];
```

Созданные элементы сразу получают начальное значение, равное нулевому значению соответствующего типа (для чисел это целый или вещественный нуль, для символов — символ с кодом 0, для строк и других ссылочных типов данных — нулевая ссылка `nil`).

Инициализацию массива целесообразно объединить с его описанием. При этом, в силу *вывода типов* (см. п. 1.1), имя типа можно не указывать, поскольку компилятор может вывести этот тип по виду инициализирующего выражения, например:

```
var a := new integer[10];
```

Элементы динамического массива *всегда* индексируются от 0 (в отличие от статических массивов, для которых можно указывать диапазон изменения индексов). *Размер* динамического массива (т. е. количество его элементов) можно определить с помощью его свойства `Length` (для обращения к свойствам используется точечная нотация, например `a.Length`). Подчеркнем, что к свойству `Length` (как и к элементам массива по их индексам) можно обращаться только после инициализации массива.

Для динамического массива определены еще два свойства: `Low` и `High`, определяющие соответственно нижнюю и верхнюю границу диапазона изменения индекса. Свойство `a.Low` всегда возвращает 0, а свойство `a.High` связано с размером `a.Length` массива соотношением `a.High = a.Length - 1`. Интересно отметить, что эти свойства (в отличие от свойства `Length`) можно

вызывать даже для неинициализированного динамического массива; в этом случае свойство `Low` будет, как обычно, возвращать 0, а свойство `High` вернет  $-1$ . Свойство `Low` практически не используется; свойство `a.High` может применяться в качестве более краткой альтернативы для выражения `a.Length - 1` (часто возникающего в заголовке цикла `for` при переборе элементов динамического массива).

Свойства `Length`, `Low` и `High` доступны только для чтения.

Кроме *свойств* `a.Length`, `a.Low`, `a.High` для динамического массива `a` можно использовать одноименные *функции* `Length(a)`, `Low(a)`, `High(a)`, возвращающие те же характеристики массива (размер, нижнюю и верхнюю границы индексов). Функции `Low(a)` и `High(a)` можно также использовать для *статических* массивов, в то время как функция `Length(a)` для статических массивов не определена. Функция `Low(a)` для статического массива может возвращать значение, отличное от 0. Например, для массива `a` типа `array[2..10] of real` функция `Low(a)` вернет значение 2, а функция `High(a)` — значение 10.

Переменные, связываемые со статическими и динамическими массивами, имеют важные различия, которые проявляются, в частности, при выполнении для них операции присваивания. В данной главе нам не потребуется учитывать эти различия, поэтому мы обсудим их позднее (см. п. 5.1).

Для *вывода* элементов массива можно использовать несколько конструкций. Наиболее традиционным является вывод в цикле `for`:

```
for var i := 0 to a.Length - 1 do
  Print(a[i]);
```

Обратите внимание на то, что последним допустимым индексом массива является `a.Length - 1`; если бы в качестве конечного значения параметра цикла было ошибочно указано значение `a.Length`, то при выполнении программы произошел бы выход за границы массива. Подобная ошибка является очень распространенной; поскольку она может приводить к непредсказуемым последствиям при последующей работе программы, в PascalABC.NET, как и в большинстве других современных языков программирования, выполняется автоматическая проверка индексов массивов, и при их выходе за допустимый диапазон программа немедленно завершается с соответствующим сообщением об ошибке.

Более безопасным (и чуть более коротким) является использование свойства `a.High`:

```
for var i := 0 to a.High do
  Print(a[i]);
```

Отметим также, что благодаря использованию процедуры `Print` выводимые значения отделяются друг от друга пробелами. Стандартная процедура `Write` этого не делает, и поэтому при ее использовании приходится яв-

но выводить пробелы или применять форматированный вывод данных; в противном случае все выводимые значения целого типа сольются в одно длинное «число».

Все описываемые далее дополнительные возможности, связанные с выводом массивов, применимы не только к массивам, но и к любым *последовательностям* (см. их описание в п. 3.4), а также к другим структурам данных, которые могут быть неявно преобразованы в последовательности (будем называть такие структуры *коллекциями*; различные виды коллекций описываются во втором выпуске данной серии — см. [1, гл. 2–5]).

Для перебора элементов любой коллекции в PascalABC.NET предусмотрен специальный вариант оператора цикла — цикл `foreach`. Использование цикла `foreach` позволяет при выводе элементов массива обойтись без применения индексов:

```
foreach var e in a do
    Print(e);
```

В данном случае параметром цикла выступает переменная `e`, имеющая тот же тип, что и элементы массива `a` (выбор имени параметра цикла объясняется тем, что с буквы «e» начинается слово «element»). На каждой итерации в переменную `e` записывается очередной элемент массива `a`. Обратите внимание на то, что тип переменной `e` (как и тип переменной `i` в цикле `for`) указывать не требуется: он выводится из типа связываемых с ней данных. Поскольку переменная `e` описана в заголовке цикла, она существует только на протяжении его работы и уничтожается после завершения цикла.

Необходимо отметить ограничения цикла `foreach`: во-первых, он всегда перебирает *все* элементы массива, причем в порядке возрастания их индексов, и, во-вторых, он не позволяет *изменять* элементы массива.

В PascalABC.NET предусмотрены более простые способы вывода элементов массива или других коллекций, не требующие применения циклов.

Во-первых, любая коллекция может быть непосредственно указана в *процедуре* `Write/Writeln` или `Print/Println`:

```
Writeln(a); // пример вывода: [1,5,3,13,20]
```

В этом случае вывод массива или последовательности выполняется в специальном формате: элементы разделяются запятыми, а весь список элементов заключается в квадратные скобки.

Напомним, что аналогичным образом оформляется в процедурах `Write` и `Print` вывод *кортежей* (см. п. 1.4); единственное отличие состоит в том, что список полей кортежа заключается не в квадратные, а в *круглые* скобки (следует заметить, что кортеж не является коллекцией, так как его нельзя преобразовать в последовательность). Имеются особые виды коллекций, например *множества* и *словари* [1, гл. 3–4], при выводе элементов которых используются *фигурные* скобки `{ }`.

Во-вторых, можно использовать *методы* Print и Println, вызываемые для массивов и других коллекций с помощью *точечной нотации*. Например, обеспечить вывод всех элементов массива **a** в том же виде, в каком они были бы выведены при использовании любого из двух вариантов с операторами цикла, можно так:

```
a.Print; // пример вывода: 1 5 3 13 20
```

Вызов `a.Println` дополнительно обеспечит переход в окне вывода на новую строку. В методах Print и Println (в отличие от одноименных процедур) можно настроить разделитель, который выводится между элементами; для этого его достаточно указать в качестве *параметра* этих методов. Например, для того чтобы вывести каждый элемент массива **a** на новой строке, можно использовать следующий вариант метода:

```
a.Print(#13); // Сейчас это a.PrintLines
```

В качестве параметра здесь указана константа `#13` — символ с кодом 13, который интерпретируется системой как переход на новую строку.

Кроме того, можно изменить системную переменную `PrintDelimDefault`, содержащую значение разделителя по умолчанию (в начале программы в ней записан символ пробела). Методы Print/Println без параметров используют текущее значение этой переменной в качестве разделителя, за исключением случая, когда выводится *массив символов* (`array of char`): для такого массива разделителем по умолчанию всегда считается *пустая строка* (т. е. символы выводятся без разделителей). Методы Print и Println печатают разделитель только *между* выводимыми элементами коллекции, а после последнего элемента разделитель не выводится.

Для вывода коллекций (в частности, массивов) при решении задач с использованием задачника Programming Taskbook предусмотрены *методы* Write и WriteAll, реализованные в модуле PT4. Первый из них пересылает задачнику все элементы коллекции, а второй перед этим действием дополнительно пересылает ее размер (поскольку в некоторых задачах требуется вывести не только элементы, но и, предварительно, размер полученного набора данных). Вместо метода Write можно использовать методы WriteLn, Print и Println (для программ с подключенным модулем PT4 эти методы выполняют одинаковые действия); вместо метода WriteAll можно использовать его синоним PrintAll.

*Процедуры* Write и Print при использовании совместно с задачником *не позволяют* указывать коллекции и другие составные данные (например, кортежи) в качестве параметров; допускаются только данные базовых скалярных типов: логического, целочисленного, вещественного, символьного и строкового (а также данные типа PNode и Node для групп заданий, связанных с динамическими структурами — см. [2]).



### 3.3. Функции генерации массивов

PascalABC.NET обладает богатым набором функций, позволяющих формировать массивы с нужным содержимым. Благодаря этим функциям во многих случаях при определении массива можно обойтись не только без указания его типа, но и без вызова конструктора.

Имеются две основные функции генерации массивов: `ArrFill` и `ArrGen`, причем вторая функция имеет несколько перегруженных вариантов, отличающихся своими параметрами. Во всех функциях генерации массива первым параметром является его размер `count` типа `integer`. Вторым параметром функции `ArrFill` является значение, которое присваивается всем элементам созданного массива. Этот параметр может иметь произвольный тип; именно по типу второго параметра определяется тип элементов массива, т. е. тип возвращаемого значения функции `ArrFill`.

Подобное «гибкое» поведение функции `ArrFill` возможно благодаря тому, что эта функция (как и практически все подпрограммы, связанные с обработкой массивов, и очень многие другие подпрограммы языка PascalABC.NET) является *обобщенной функцией*. Обобщенные подпрограммы позволяют использовать в качестве одного или нескольких типов имена-«заменители» (обычно `T` или `T1`, `T2`, и т. д.), означающие *любой* возможный тип. Имена-заменители должны указываться в заголовке подпрограммы в угловых скобках сразу после ее имени и в дальнейшем могут применяться, как и «обычные» имена типов, при описании параметров, возвращаемого значения функции и локальных переменных. Например, заголовки функции `ArrFill` выглядят следующим образом:

```
function ArrFill<T>(count: integer; x: T): array of T
```

С помощью этой функции можно очень легко определить массив, состоящий, например, из 10 целых чисел, равных 1:

```
var a := ArrFill(10, 1);
```

Если в качестве второго параметра указать `1.0`, то будет создан массив *вещественных* чисел того же размера и с тем же значением.

Гораздо больше возможностей по созданию массивов предоставляют различные варианты функции `ArrGen`. Во всех этих вариантах ключевую роль играют параметры — лямбда-выражения, которые подробно обсуждались в п. 1.3. Приведем заголовки для всех вариантов функции `ArrGen` (здесь и далее в **полужирных** квадратных скобках будут указываться **необязательные** параметры, для которых предусмотрены значения по умолчанию; кроме того, для упрощения записи мы не будем в дальнейшем указывать текст `<T>` после имени обобщенной подпрограммы):

```
function ArrGen(count: integer; f: integer -> T  
[; from: integer]): array of T
```

```
function ArrGen(count: integer; first: T; next: T -> T): array of T
function ArrGen(count: integer; first, second: T;
  next: (T,T) -> T): array of T
```

Как уже было отмечено, первый параметр определяет размер массива. Если вторым параметром является лямбда-выражение  $f$  типа  $\text{integer} \rightarrow T$ , то в массив последовательно записываются значения  $f(i)$ , начиная от  $i$ , равного параметру `from`. Если параметр `from` не указан, то он полагается равным 0; в этом случае значение  $f(i)$  записывается в элемент массива с индексом  $i$ . Данный вариант функции `ArrGen` удобно использовать, если имеется явная формула, определяющая значение элемента по его индексу.

В качестве примера использования данного варианта функции `ArrGen` приведем решение задачи **Array1**, в которой требуется сформировать массив, состоящий из  $n$  первых положительных нечетных чисел (1, 3, 5, ...):

```
Task('Array1'); // Вариант 1
var a := ArrGen(ReadInteger, i -> 2 * i + 1);
a.Write;
```

Для вывода полученного массива мы использовали вспомогательный метод `Write` из модуля `PT4`. Поскольку метод `Write` можно применить к массиву сразу после его создания, в решении можно обойтись без переменной `a`:

```
Task('Array1'); // Вариант 1a
ArrGen(ReadInteger, i -> 2 * i + 1).Write;
```

Разумеется после выполнения данного оператора мы потеряем связь с созданным массивом, однако на правильность решения это не повлияет.

Обратимся к другим вариантам функции `ArrGen`. Эти варианты удобны в ситуациях, когда для определения значений элементов массива используются не явные, а *рекуррентные* формулы, т. е. выражения, в которых значение очередного элемента определяется через значение одного или нескольких *предыдущих* элементов. При этом надо явно определить значения соответствующего количества *начальных* элементов массива. Функция `ArrGen` позволяет использовать рекуррентные соотношения с одним или двумя параметрами. Иными словами, с ее помощью можно создавать массивы, в которых каждый последующий элемент определяется по одному или двум предыдущим элементам. Рекуррентная формула указывается в виде лямбда-выражения `next` и является последним параметром функции. Перед ней указывается один или два параметра, задающие значения начальных элементов массива.

Используя рекуррентное соотношение, мы можем запрограммировать решение задачи `Array1`, не содержащее операции умножения:

```
Task('Array1'); // Вариант 2
ArrGen(ReadInteger, 1, e -> e + 2).Print;
```

Классическим примером последовательности, в которой элемент определяется по двум предыдущим, является *последовательность Фибоначчи*. В ней два первых элемента равны 1, а следующие равны сумме двух предыдущих: 1, 1, 2, 3, 5, 8, 13, 21, 34 и т. д. В задаче **Array5** требуется сформировать и вывести массив, содержащий  $n$  первых чисел Фибоначчи. С использованием последнего варианта функции `ArrGen` эта задача также решается в одну строку:

```
Task('Array5'); // Вариант 1
ArrGen(ReadInteger, 1, 1, (e1, e2) -> e1 + e2).Print;
```

При программировании рекуррентной формулы с двумя параметрами надо учитывать, что ближайшим соседом к формируемому элементу считается *второй* параметр  $e2$  лямбда-выражения; иными словами, если через  $e3$  обозначить значение вычисляемого элемента, то порядок значений в массиве будет следующим:  $e1, e2, e3$ .

Завершая обзор средств формирования массивов, отметим две дополнительные возможности. Во-первых, в массив можно превратить любой набор *однотипных* значений, указав эти значения в качестве параметров функции `Arr` (количество параметров может быть произвольным):

```
var a := Arr(1, 10, 2, 9, 3, 8);
```

При выполнении этого оператора мы получим массив `a` из 6 указанных целых чисел (располагающихся в массиве в том же порядке). Вызов `Arr(x)` формирует *одноэлементный* массив, единственный элемент которого имеет значение  $x$  (в некоторых ситуациях требуется использовать одноэлементные массивы, и указанный вызов является простейшим способом создать такой массив). Функция `Arr` входит в группу так называемых *коротких функций* (названных так из-за краткости их имен), позволяющих быстро сформировать различные структуры с требуемыми значениями. Другие примеры коротких функций приводятся в п. 3.4 и 4.6.

Второй возможностью, оказывающейся полезной, в частности, при тестировании программ, является создание числовых массивов, заполненных случайными значениями. Для этого предназначены функции `ArrRandomInteger` (имеет краткий синоним `ArrRandom`) и `ArrRandomReal`:

```
function ArrRandomInteger(n: integer := 10; a: integer := 0;
  b: integer := 100): array of integer
function ArrRandomReal(n: integer := 10; a: real := 0;
  b: real := 10): array of real
```

В каждой из этих функций можно указывать от 0 до 3 необязательных параметров. Первый параметр  $n$ , как и в случае функций `ArrFill` и `ArrGen`, задает размер массива (по умолчанию создается массив из 10 элементов), второй и третий параметры  $a$  и  $b$  задают диапазон, из которого выбираются случайным образом значения элементов. В случае целых чисел в диапазон

входят все целые числа от  $a$  до  $b$  включительно, по умолчанию используется диапазон от 0 до 100; в случае вещественных чисел диапазон представляет собой полуинтервал  $[a, b)$ , т. е. левый конец диапазона включается, а правый нет, по умолчанию используется полуинтервал  $[0; 10)$ . В случае вещественных чисел при  $a \neq b$  можно считать, что случайные числа выбираются из *интервала*  $(a, b)$ , так как вероятность случайного выбора числа, равного  $a$ , практически равна нулю.

Интересно отметить, что параметр  $a$  может быть *больше* параметра  $b$ , границы диапазона при этом будут установлены правильно. Эта особенность справедлива и для полезной функции `Random(a, b: integer)`, которая возвращает случайное целое число от  $a$  до  $b$  включительно: допускается ситуация, когда  $a > b$ , в этом случае используется диапазон от  $b$  до  $a$  (напомним, что имеется также вариант функции `Random` без параметров, возвращающий случайное *вещественное* число из полуинтервала  $[0; 1)$ ). Отмеченную особенность надо учитывать при указании в функциях `ArrRandomInteger` и `ArrRandomReal` *только* параметра  $a$  (без параметра  $b$ ). Например, вызов `ArrRandomInteger(20, 5)`, как и следовало ожидать, вернет массив из 20 элементов со случайными значениями в диапазоне от 5 до 100 (значение  $b = 100$  используется по умолчанию). Если же выполнить вызов `ArrRandomInteger(20, 150)`, то мы получим массив из 20 элементов со случайными значениями в диапазоне от 100 до 150 (значение  $b$  по-прежнему считается равным 100, поэтому границы  $a = 150$  и  $b = 100$  «меняются местами»). Аналогично будет вести себя и функция `ArrRandomReal`, вызванная с двумя параметрами  $n$  и  $a$ .

### 3.4. Последовательности

*Последовательность* (sequence) представляет собой тип данных, пришедший в `PascalABC.NET` из стандартной библиотеки платформы `.NET` (в которой он реализован в виде *обобщенного интерфейса* `IEnumerable<T>` — см. [3, п. 9.7, 10.1]). В традиционном Паскале подобный тип отсутствует. Последовательности обладают рядом замечательных свойств, которые позволяют во многих случаях использовать их более эффективно, чем другие структуры данных, в частности, массивы. В то же время массивы, строки и большинство других структур данных могут быть неявно преобразованы в последовательности (напомним, что мы условились называть такие структуры *коллекциями*). Поэтому средства, предназначенные для обработки последовательностей, можно применять к любым коллекциям; следует лишь учитывать, что в результате применения этих средств будут получены не структуры исходного типа, а последовательности.

Последовательности в языке `PascalABC.NET` описываются подобно динамическим массивам, за исключением того, что вместо ключевого слова `array` надо использовать слово `sequence`, например:

```
var a: sequence of integer;
```

Таким образом, все элементы последовательности (как и массива) имеют одинаковый тип.

Особой конструкции для инициализатора последовательности не предусмотрено, однако для последовательностей можно использовать тот же набор функций-генераторов, что и для массивов (эти функции подробно описывались в предыдущем пункте); требуется лишь изменить префикс «Arr» в именах этих функций на префикс «Seq». Таким образом, для генерации последовательностей можно использовать функции Seq (еще один пример *короткой функции*, допускающей указание произвольного количества однотипных параметров), SeqFill, SeqGen, SeqRandomInteger (имеет короткий синоним SeqRandom) и SeqRandomReal с теми же наборами параметров. Дополнительные возможности, связанные с генерацией последовательностей, будут рассмотрены в следующем пункте.

Главной особенностью последовательностей является то, что они не выделяют памяти для хранения своих элементов. Точнее, память не выделяется для *каждого* элемента, вместо этого последовательность хранит в памяти информацию о том, как получить каждое значение при необходимости. Проиллюстрируем эту особенность на примере. Предположим, что последовательность *a* определена следующим образом:

```
var a := SeqFill(1000000000, 1);
```

Если бы вместо SeqFill была использована функция ArrFill, то программа попыталась бы выделить память для хранения миллиарда целочисленных элементов, что привело бы при выполнении программы к ее аварийному завершению из-за нехватки памяти. В то же время функция SeqFill с такими же параметрами будет успешно выполнена, и последовательность *a* будет создана. Однако храниться в этой последовательности будет не миллиард единиц, а информация (в специальном формате) о том, что *при необходимости* из этой последовательности можно извлечь миллиард элементов, каждый из которых будет равен единице. Подчеркнем, что сами элементы нигде не хранятся; они будут созданы тогда, когда потребуется их использовать, но и в этом случае они будут создаваться не одновременно, а по очереди, т. е. по мере их использования.

Простейший способ использования элементов последовательности — их перебор в цикле foreach. Например, к предыдущему оператору можно добавить следующий фрагмент:

```
var sum := 0;
foreach var e in a do
    sum += e;
sum.Print;
```

После запуска такой программы через некоторое время (порядка 10–30 секунд) она выведет число 1000000000. Таким образом, в цикле были обработаны все элементы последовательности, причем одновременно в памяти они не размещались (как мы знаем, это привело бы к аварийному завершению программы).

Еще одной важной особенностью последовательностей является их *неизменяемость* (ранее мы отмечали, что эта особенность имеется и у кортежей — см. п. 1.4). У последовательности нельзя изменить какой-либо отдельный элемент; можно лишь преобразовать всю последовательность, получив в результате новую последовательность.

Основным способом преобразования последовательности является применение к ней некоторого *запроса* — метода, обычно возвращающего новую последовательность (хотя имеются запросы, возвращающие скалярные данные или другие структуры, например массивы). Запросы, преобразующие одну последовательность в другую, как правило, не обрабатывают сразу каждый элемент исходной последовательности; вместо этого они в специальном формате *описывают* то преобразование, которое надо применить к элементам, «откладывая» фактическое выполнение преобразования «на потом» — к моменту, когда, например, потребуется организовать перебор полученной последовательности в цикле `foreach`. Запросы для последовательностей будут подробно обсуждаться далее (см. главу 4), а сейчас для иллюстрации их особенностей мы используем один из самых распространенных запросов — `Select`, который преобразует все элементы исходной последовательности в соответствующие элементы новой последовательности по определенному правилу (задаваемому в виде лямбда-выражения).

Вернемся к предыдущему примеру и применим к созданной последовательности запрос `Select`. Кроме того, уменьшим размер последовательности в 10 раз (с 1 миллиарда до 100 миллионов элементов) и добавим в конец фрагмента вывод времени работы программы (в миллисекундах), используя для этого стандартную функцию `Milliseconds`:

```
// Вариант 1
var a := SeqFill(100000000, 1).Select(e -> 2 * e);
var sum := 0;
foreach var e in a do
    sum += e;
Print(sum, Milliseconds);
```

**Замечание.** При анализе быстродействия программ следует правильно выбрать режим их выполнения. Если программа запущена по нажатию клавиши F9 (этот вариант запуска связан с командой меню «Программа | Выполнить»), то она будет выполняться в *режиме отладки* (`debug mode`). В этом режиме компилятор добавляет к программе дополнительный код,

который облегчает выявление и исправление ошибок времени выполнения, но замедляет работу программы. Чтобы этого избежать, следует запустить программу в *режиме релиза* (release mode), в котором отладочный код к программе не добавляется. В системе PascalABC.NET для запуска программы в режиме релиза необходимо предварительно откорректировать настройки системы. Окно настроек вызывается командой «Сервис | Настройки...»; в этом окне надо перейти в раздел «Опции компиляции» и в этом разделе *снять* флажок «Генерировать отладочную информацию (Debug-версия)», после чего нажать кнопку «ОК». Кроме того, для запуска в режиме релиза необходимо использовать комбинацию Shift+F9 (связанную с командой меню «Программа | Выполнить без связи с оболочкой»). В этом случае для программы создается *консольное окно*, в котором выполняется ввод-вывод данных. Приводимые в книге результаты, связанные с измерением времени, получены при запуске программ в режиме релиза.

Приведем вариант выведенных данных (первое число всегда будет одинаковым, а второе зависит от быстродействия компьютера):

```
200000000 2594
```

В исходной последовательности содержатся сто миллионов единиц. Метод `Select` преобразует каждую единицу в двойку. Полученная последовательность сохраняется в переменной `a`, и затем ее элементы суммируются в цикле `foreach`. Важно понять, что перебор элементов выполняется *только* в цикле `foreach`. Казалось бы, для удвоения элементов в запросе `Select` их тоже надо перебрать, однако это излишне: достаточно к информации о содержимом исходной последовательности *добавить* новую информацию о том, как преобразовать эти элементы *в тот момент, когда они потребуются*. Ясно, что это действие выполняется быстро и не зависит от фактического размера последовательности. Указанный фрагмент, как и предыдущий, практически не использует оперативную память, так как все элементы исходной последовательности создаются, преобразуются и наконец суммируются *по очереди*, в цикле `foreach`.

Поскольку мы уменьшили размер исходного набора в 10 раз, нам удастся создать *массив* требуемого размера и выполнить с его помощью такие же действия:

```
// Вариант 2  
var a := ArrFill(100000000, 1);  
for var i := 0 to a.Length - 1 do  
    a[i] *= 2;  
var sum := 0;  
foreach var e in a do  
    sum += e;  
Print(sum, Milliseconds);
```

При запуске этой программы на том же компьютере был получен следующий результат:

**200000000 1638**

Время меньше, чем для варианта 1. Следует, однако, учитывать, что в варианте 2 выделяется память для хранения *всех* элементов. Время, требуемое на размещение в памяти массива и его преобразование, компенсируется ускорением при его обработке в цикле `foreach`, поскольку теперь программа не генерирует элементы исходной последовательности «на лету», а просто читает их значения, размещенные в памяти.

Модифицируем последний вариант, заменив цикл `foreach` на цикл `for`:

// **Вариант 3**

```
var a := ArrFill(100000000, 1);
for var i := 0 to a.Length - 1 do
  a[i] := a[i] * 2;
var sum := 0;
for var i := 0 to a.Length - 1 do
  sum += a[i];
Print(sum, Milliseconds);
```

Теперь результаты программы будут следующими:

**200000000 1648**

Таким образом, скорость работы вариантов 2 и 3 совпадает.

Вернемся к варианту 1. При работе с последовательностями надо активно использовать *запросы*, поскольку именно они позволяют выполнять обработку последовательностей наиболее эффективным образом. В нашем случае для суммирования элементов последовательности следовало использовать не цикл, а специальный запрос `Sum`:

// **Вариант 4**

```
var a := SeqFill(100000000, 1).Select(e -> 2 * e);
var sum := a.Sum;
Print(sum, Milliseconds);
```

Теперь программа выведет следующие данные:

**200000000 1347**

Мы получили наилучшее быстродействие из всех рассмотренных вариантов. Заметим, что запрос `Sum` можно было применить непосредственно к результату, возвращенному запросом `Select`, сформировав *цепочку запросов* (это стандартный способ обработки последовательностей):

// **Вариант 4а**

```
var sum := SeqFill(100000000, 1).Select(e -> 2 * e).Sum;
Print(sum, Milliseconds);
```



Скорость выполнения вариантов 4 и 4а будет одинаковой.

В качестве еще одного варианта реализации данной программы рассмотрим следующий:

**// Вариант 5**

```
var sum := ArrFill(100000000, 1).Select(e -> 2 * e).Sum;  
Print(sum, Milliseconds);
```

Он отличается от варианта 4а только тем, что вместо функции SeqFill используется функция ArrFill. Результат работы программы:

**200000000 2211**

Время хуже, чем для вариантов 2 и 3 с применением циклов. Таким образом, хотя к массиву допустимо применять запросы (поскольку массивы могут быть неявно преобразованы в последовательности), они не дают такого же эффекта ускорения, как при применении к последовательностям.

Как мы выяснили, анализируя варианты 2 и 3, цикл `for` для массивов работает с той же скоростью, что и цикл `foreach`. Что произойдет, если мы попытаемся использовать цикл `for` для *последовательностей*? Для этого необходимы запросы, которые позволили бы определять размер последовательности и значение ее *i*-го элемента. Такие запросы есть: это `Count` и `ElementAt(i)` соответственно. Однако их *никогда* не следует использовать для организации перебора элементов последовательности, поскольку запрос `ElementAt(i)` работает *крайне медленно* (и, кроме того, *время его работы увеличивается с увеличением значения индекса i*). Поэтому окончания работы следующего варианта программы мы не дождемся<sup>2</sup>:

**// Вариант 6 (НЕДОПУСТИМЫЙ)**

```
var a := SeqFill(100000000, 1).Select(e -> 2 * e);  
var sum := 0;  
for var i := 0 to a.Count-1 do  
    sum += a.ElementAt(i);  
Print(sum, Milliseconds / 1000);
```

Итак, на основе анализа различных вариантов нашей программы можно сделать вывод о том, что как массивы, так и последовательности являются эффективным средством обработки данных, однако при работе с ними надо учитывать их особенности. Нетрудно заметить, что программы, использующие последовательности, оказываются чрезвычайно наглядными. В то же время, следует иметь в виду, что нередко алгоритмы, основанные на обработке последовательностей, работают медленнее, чем традиционные алгоритмы, использующие циклы (поскольку программный код с последовательностями сложнее оптимизировать). Кроме того, имеются

---

<sup>2</sup> По приблизительным подсчетам, данный вариант программы выполнялся бы на том же компьютере, на котором запускались предыдущие варианты, более двух лет.

группы задач, для решения которых последовательности плохо приспособлены (например, из-за очень медленного доступа к элементу по его индексу). И в первом, и во втором выпуске настоящего пособия все особенности последовательностей мы проиллюстрируем на многочисленных примерах.

### 3.5. Вывод последовательностей и их генерация. Бесконечные последовательности

При описании средств вывода элементов массивов в конце п. 3.2 было отмечено, что многие из этих средств можно использовать не только для массивов, но и для последовательностей. Фактически единственным средством, которым *не следует* пользоваться для вывода последовательности, является обычный цикл `for`.

Напомним, что для *вывода* последовательностей можно использовать цикл `foreach`, процедуры `Write/WriteLn` и `Print/PrintLn`, а также *методы* `Print/PrintLn`. В методах `Print/PrintLn`, как и в одноименных методах для массивов, можно указывать необязательный строковый параметр `delim` — разделитель, который будет выводиться между значениями элементов. По умолчанию разделителем считается пробел (исключением является последовательность символов `sequence of char`, для которой разделителем по умолчанию *всегда* считается пустая строка). Изменить разделитель по умолчанию можно, записав его новое значение в системную переменную `PrintDelimDefault`.

При выводе последовательностей в программах, выполняющих задания из электронного задачника `Programming Taskbook`, следует использовать методы `Write` и `WriteAll` (метод `WriteAll` перед выводом элементов дополнительно выводит размер последовательности). Вместо метода `Write` можно использовать метод `Print`, который в данном случае выполняет те же самые действия.

В предыдущем пункте мы отмечали, что для *генерации* последовательностей можно использовать функции `Seq`, `SeqFill`, `SeqGen`, `SeqRandomInteger` (синоним `SeqRandom`) и `SeqRandomReal`. Эти функции аналогичны по своему действию функциям для генерации массивов. Кроме них для последовательностей предусмотрены еще две функции-генератора: `Range` и `SeqWhile`.

Функция `Range` позволяет задавать диапазоны целых чисел или символов, а также диапазоны равномерно распределенных вещественных чисел. Приведем заголовки трех перегруженных вариантов этой функции:

```
function Range(k1, k2: integer [; step: integer]):  
    sequence of integer  
function Range(c1, c2: char): sequence of char  
function PartitionPoints(a1, a2: real; n: integer): sequence of real
```

Первый вариант возвращает последовательность целых чисел в диапазоне от  $k_1$  до  $k_2$  включительно, второй — последовательность символов в диапазоне от  $c_1$  до  $c_2$  включительно, третий — последовательность вещественных чисел, определяющих точки разбиения отрезка  $[a_1, a_2]$  на  $n$  равных частей (включая и концы этого отрезка).

Первый вариант может иметь дополнительный ненулевой параметр `step` — шаг, с которым перебираются целые числа, включаемые в диапазон (начиная с  $k_1$ ). По умолчанию параметр `step` равен 1. Для любых положительных значений шага должно выполняться условие  $k_1 \leq k_2$ ; в противном случае будет возвращена пустая последовательность. Для отрицательных значений шага `step`, наоборот, должно выполняться условие  $k_1 \geq k_2$ , при этом элементы полученной последовательности будут располагаться по убыванию. Если шаг отличен от 1 и  $-1$ , то значение  $k_2$  может не попасть в полученную последовательность.

Для второго варианта метода символ  $c_1$  должен иметь код, меньший или равный коду символа  $c_2$ , в противном случае будет возвращена пустая последовательность.

В третьем варианте число  $n$  должно быть положительным. В полученной последовательности будет содержаться  $n + 1$  вещественное число; если  $a_1 < a_2$ , то числа будут располагаться по возрастанию (от  $a_1$  до  $a_2$ ), если  $a_1 > a_2$ , то числа будут располагаться по убыванию (также от  $a_1$  до  $a_2$ ). Допускается ситуация  $a_1 = a_2$ , в этом случае все элементы последовательности будут одинаковыми (и равными  $a_1$ ).

При использовании функции `Range` следует очень внимательно относиться к указанию типов ее параметров, поскольку изменение типа единственного параметра может привести к совершенно иному результату. В качестве примера рассмотрим два вызова функции `Range`:

```
Range(0, 10, 5).Println;  
PartitionPoints(0.0, 10, 5).Println;
```

Результат выполнения этих операторов будет следующим:

```
0 5 10  
0 2 4 6 8 10
```

В первом случае, поскольку все параметры являются целыми, используется первый вариант функции `Range`, для которого третий параметр определяет шаг изменения значений. Поэтому перебираются и выводятся все целые числа от 0 до 10 с шагом 5. Во втором случае, поскольку первый параметр является вещественным, выбирается третий вариант функции `Range` (и при этом второй параметр также приводится к вещественному типу). Для третьего варианта третий параметр означает число отрезков, на которые надо разбить исходный отрезок, поэтому в этом случае выполняется разбиение отрезка  $[0; 10]$  на 5 равных отрезков, и печатаются концы

полученных отрезков в порядке возрастания (на самом деле вторая последовательность состоит из вещественных чисел, однако все они имеют нулевую дробную часть, которая при печати не указывается).

Используя третий вариант функции `Range`, легко реализовать процедуру табуляции `Tab`, которая упоминалась в п. 1.3. Напомним, что эта процедура печатает значения функции  $f$  в точках разбиения отрезка  $[a, b]$  на  $n$  равных частей и имеет следующий заголовок:

```
procedure Tab(f: real -> real; a, b: real; n: integer);
```

Простейший вариант реализации этой процедуры, выводящей каждое значение табулируемой функции на новой строке и не использующий специальные средства форматирования выводимых данных, может быть представлен в виде единственного оператора:

```
PartitionPoints(a, b, n).Select(x -> f(x)).PrintLines;
```

Вначале мы генерируем последовательность точек, в которых требуется вычислить функцию, затем применяем к этой последовательности запрос `Select`, преобразующий аргументы  $x$  в значения функции  $f(x)$ , а затем выводим полученную последовательность значений с помощью метода `Println`, явно указывая разделитель `#13` — разрыв строки.

Если указанным образом описать процедуру `Tab`, а затем выполнить фрагмент с ее вызовами, приведенный к концу п. 1.3, то на экране будут выведены следующие данные:

```
0
0.01
0.04
0.09
0.16
0.25
0.36
0.49
0.64
0.81
1
1
0.81
0.64
0.49
0.36
0.25
0.16
0.09
0.04
0.01
0
```

Завершив на этом обсуждение функции `Range`, обратимся к функции `SeqWhile`. Она имеет два перегруженных варианта со следующими заголовками (напомним, что мы условились для краткости не указывать текст `<T>` после имени обобщенной функции):

```
function SeqWhile(first: T; next: T -> T; pred: T -> boolean):  
    sequence of T  
function SeqWhile(first, second: T; next: (T,T) -> T;  
    pred: T -> boolean): sequence of T
```

Данная функция имеет в числе своих параметров два лямбда-выражения. Первое из них, `next`, определяет правило, по которому следующий элемент последовательности будет сгенерирован на основе одного или двух предыдущих, а второе, `pred` (от слова *predicate* — *предикат*), определяет, когда следует прекратить заполнение последовательности. Последовательность заполняется до тех пор, пока предикат `pred` возвращает `True` для очередного значения, полученного функцией `next`; только в этом случае полученное значение добавляется в последовательность. Как только значение, возвращенное функцией `next`, перестанет удовлетворять предикату, построение последовательности завершится; таким образом, можно утверждать, что *все* элементы полученной последовательности будут удовлетворять предикату `pred`. Параметры `first` и `second` функции `SeqWhile` задают начальные значения формируемой последовательности (одно, если рекуррентная формула `next` использует одно предыдущее значение, и два, если формула `next` использует два предыдущих значения). Эти начальные значения также проверяются предикатом `pred`; если, например, для параметра `first` предикат `pred` вернет значение `False`, то функция `SeqWhile` вернет пустую последовательность.

Интересной особенностью функции `SeqWhile` является то, что при ее вызове *неизвестен* размер той последовательности, которую она создаст. Это отличает ее от всех функций-генераторов, которые являются общими для массивов и последовательностей (`ArrFill` и `SeqFill`, `ArrGen` и `SeqGen`, `ArrRandom` и `SeqRandom` и т. п.). Во всех функциях-генераторах массивов *вначале* выделяется память для хранения элементов массива, а *затем* они заполняются по тому или иному алгоритму. Именно поэтому в наборе генераторов для массивов отсутствует аналог функции `SeqWhile`: любая реализация подобного генератора для массива была бы неэффективной.

В случае же последовательностей никаких проблем с реализацией функции `SeqWhile` не возникает, поскольку при генерации последовательности ее элементы не размещаются в памяти. Фактический перебор элементов последовательности выполняется либо в цикле `foreach`, либо при выполнении некоторых видов запросов (например, уже использовавшегося

нами запроса Sum), либо при ее выводе с помощью соответствующих процедур или методов.

При анализе функции SeqWhile можно задаться таким вопросом: что произойдет, если предикат *pred* *никогда* не вернет значение False? Если бы предикат использовался в качестве условия обычного цикла while, то произошло бы заикливание программы. В нашем случае ситуация будет интереснее: последовательность будет успешно создана (поскольку для ее создания не требуется генерировать все ее элементы; достаточно сохранить каким-либо образом способ их генерации). Проблемы возникнут при попытке перебрать все элементы этой «бесконечной» последовательности.

Предположим, например, что мы создадим с помощью функции SeqWhile бесконечную последовательность Фибоначчи:

```
var fib := SeqWhile(1.0, 1.0, (a, b) -> a + b, e -> True);
```

Для того чтобы в этой последовательности правильно вычислялись числа Фибоначчи с большими номерами, мы указали в качестве начальных значений *вещественные* числа.

Данный оператор успешно откомпилируется и не приведет к каким-либо проблемам при запуске программы. Однако при попытке вывести элементы этой последовательности в цикле *foreach* мы, как и следовало ожидать, получим заиклившуюся программу. Заикливание произойдет и при попытке подсчитать, например, сумму всех элементов нашей последовательности с помощью вызова *fib.Sum*.

**Замечание.** Казалось бы, программа, которая печатает или находит сумму быстро растущих элементов последовательности скоро должна аварийно завершиться из-за ошибки, связанной с переполнением. Однако в нашем случае переполнения не происходит, так как в языке PascalABC.NET и других языках платформы .NET тип вещественных чисел реализован таким образом, что для него переполнение невозможно. Если в результате очередной операции с вещественными числами будет получено значение, превышающее максимально допустимое (это значение равно  $1.79769313486232E+308$ ; его можно получить с помощью свойства *real.MaxValue*), то в качестве результата будет возвращено особое значение *real.PositiveInfinity*, отображаемое на экране как «бесконечность». В этом легко убедиться, запустив для нашей последовательности цикл *foreach*, печатающий ее элементы: очень скоро все выводимые элементы будут иметь значение «бесконечность».

Итак, несмотря на то что бесконечную последовательность Фибоначчи можно создать, попытка перебора ее элементов приводит к заикливанию. Однако это не означает, что созданная нами последовательность является бесполезной. Дело в том, что среди стандартных запросов, которые можно применять к последовательностям, есть группа запросов, позволяющих из-

влекать из последовательности *часть* ее элементов. Простейшим из таких запросов является запрос `Take(n)`, который возвращает последовательность, состоящую из первых  $n$  элементов исходной последовательности. Наличие подобных запросов делает бесконечные последовательности не только возможным, но и удобным в ряде ситуаций средством. Например, мы можем «заранее» создать некоторую бесконечную последовательность, а затем получать и обрабатывать ее различные начальные фрагменты. В частности, после создания нашей последовательности Фибоначчи мы можем вывести ее первые 15 членов, выполнив следующий оператор:

```
fib.Take(15).Print;
```

В результате на экране будет выведен следующий набор чисел:

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

Интересно отметить, что в *процедуры* `Write` и `Print` (не методы!) встроены специальные средства, позволяющие корректно выводить очень длинные или даже бесконечные последовательности. Эти процедуры просто прерывают обработку последовательности после вывода ее первых ста элементов. Например, следующий способ вывести нашу бесконечную последовательность `fib` к заикливанию программы не приведет:

```
Write(fib);
```

Вместо этого он выведет следующий текст и продолжит выполнение программы (обратите внимание на то, что список элементов завершается многоточием):

```
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,46368,75025,121393,196418,317811,514229,832040,1346269,2178309,3524578,5702887,927465,14930352,24157817,39088169,63245986,102334155,165580141,267914296,433494437,701408733,1134903170,1836311903,2971215073,4807526976,7778742049,12586269025,20365011074,32951280099,53316291173,86267571272,139583862445,225851433717,365435296162,591286729879,956722026041,1548008755920,2504730781961,4052739537881,6557470319842,10610209857723,17167680177565,27777890035288,44945570212853,72723460248141,117669030460994,190392490709135,308061521170129,498454011879264,806515533049393,1.30496954492866E+15,2.11148507797805E+15,3.41645462290671E+15,5.52793970088476E+15,8.94439432379146E+15,1.44723340246762E+16,2.34167283484677E+16,3.78890623731439E+16,6.13057907216116E+16,9.91948530947555E+16,1.60500643816367E+17,2.59695496911123E+17,4.2019614072749E+17,6.79891637638612E+17,1.1000877783661E+18,1.77997941600471E+18,2.88006719437082E+18,4.66004661037553E+18,7.54011380474635E+18,1.22001604151219E+19,1.97402742198682E+19,3.19404346349901E+19,5.16807088548583E+19,8.36211434898484E+19,1.35301852344707E+20,2.18922995834555E+20,3.54224848179262E+20,...]
```

Поскольку бесконечные последовательности можно использовать в программах, для их генерации предусмотрены особые средства, не требующие применения «фиктивных» предикатов, всегда возвращающих значение `True`. Большинство этих средств представляют собой особые методы, которые применяются к *элементу*, используемому при генерации беско-

нечной последовательности. Имеются три таких метода: Repeat, Step и lterate.

Метод Repeat является самым простым. В результате его применения к некоторому элементу e (произвольного типа) мы получаем бесконечную последовательность, состоящую из одинаковых элементов e, например:

```
var a := 2.Repeat; // [2,2,2,2,2,2,2,...]
```

Заметим, что имя метода Repeat совпадает с одним из ключевых слов Паскаля, однако это не препятствует его использованию в программе.

Метод Step может применяться только к числовым типам и позволяет создавать бесконечные арифметические прогрессии. При этом число, для которого вызывается метод, считается первым элементом прогрессии, а ее разность указывается в качестве единственного параметра метода, например:

```
var b := 2.Step(3); // [2,5,8,11,14,17,20,...]
```

Параметр в методе Step можно не указывать, в этом случае разность прогрессии считается равной 1.

Метод lterate позволяет создавать бесконечные последовательности, основанные на применении рекуррентных формул. Он имеет два перегруженных варианта, аналогичных вариантам рассмотренной ранее функции SeqWhile (тоже использующей рекуррентную формулу). Отличия метода lterate от функции SeqWhile заключаются в том, что в нем отсутствует завершающий параметр-предикат, а также в том, что этот метод должен вызываться для начального элемента последовательности (если вычисление по рекуррентной формуле требует использования двух предыдущих элементов, то второй элемент указывается в качестве первого параметра метода lterate). Рекуррентная формула, как обычно, оформляется в виде лямбда-выражения и указывается в качестве последнего параметра метода lterate. Например, последовательность Фибоначчи, созданную ранее с помощью второго варианта функции SeqWhile, можно создать более простым способом, используя соответствующий вариант метода lterate:

```
var fib := 1.0.lterate(1.0, (a, b) -> a + b);
```

Наконец, еще одним вариантом создания бесконечной последовательности является «заикливание» уже имеющейся, конечной последовательности. Для того чтобы получить на основе последовательности s бесконечную циклическую последовательность, достаточно вызвать для исходной последовательности метод Cycle без параметров: s.Cycle. Можно также использовать вариант вызова в виде функции с параметром: Cycle(s). Пример:

```
var c := Seq(1, 2, 3).Cycle; // [1,2,3,1,2,3,1,2,3,1,2,3,...]
```

В заключение данного пункта проиллюстрируем изученные возможности, связанные с бесконечными последовательностями, решив с их по-



мощью задачи **Array1** и **Array5** (решения этих задач с применением функций-генераторов массивов были приведены в п. 3.3).

Для генерации начальной части последовательности нечетных чисел удобно использовать метод `Step`:

```
Task('Array1'); // Вариант 3
1.Step(2).Take(ReadInteger).Print;
```

Для генерации начальной части последовательности Фибоначчи можно, как уже было отмечено выше, использовать метод `Iterate`:

```
Task('Array5'); // Вариант 2
1.Iterate(1, (a, b) -> a + b).Take(ReadInteger).Print;
```

В данном случае надо указать целочисленные значения начальных элементов последовательности, так как в задаче `Array5` требуется вывести данные целого типа.

### 3.6. Дополнение. Генерация последовательностей с помощью конструкции `yield`

Начиная с версии 3.1, в `PascalABC.NET` реализован еще один, наиболее универсальный механизм генерации последовательностей, основанный на применении специального оператора `yield` (это слово может переводиться как «выход, выработка»; читается «йилд»). Хотя для решения большинства задач, связанных с обработкой последовательностей, вполне достаточно рассмотренных выше средств генерации, этот механизм полезно изучить хотя бы потому, что он позволяет более наглядно представить себе те действия, которые выполняются при создании и использовании последовательностей, и лучше понять, каким образом можно создать последовательность, не сохраняя в памяти все ее элементы.

Оператор `yield` указывается в функциях, возвращающих последовательность, в следующем формате:

```
yield выражение;
```

Выражение, указанное после слова `yield`, считается очередным элементом формируемой последовательности. Понятно, что если функция возвращает последовательность некоторого типа `T` (например, `integer`), то все выражения, указанные после `yield`, также должны иметь тип `T` или неявно приводиться к этому типу.

Операторы `yield` являются заменителями операторов, определяющих возвращаемые значения функции; в частности, в функции, использующей `yield`, запрещено обращаться к переменной `Result`.

Если в функции имеется *последовательность*, элементы которой надо вернуть, то можно использовать следующую модификацию оператора `yield`:

```
yield sequence последовательность;
```

Функция может содержать любое количество операторов `yield` и `yield sequence`; требуется лишь, чтобы выражения, указываемые в этих операторах, имели допустимые типы.

В качестве простого примера приведем функцию, генерирующую последовательность, содержащую первые  $n$  положительных нечетных чисел:

```
function OddGen(n: integer): sequence of integer;
begin
  for var i := 1 to n do
    yield 2 * i - 1;
  end;
```

С помощью этой функции можно реализовать еще один вариант решения задачи **Array1**:

```
Task('Array1'); // Вариант 4
OddGen(ReadInteger).Write;
```

При анализе функции, использующей `yield`, необходимо учитывать, что эта конструкция является «отложенной». В тот момент, когда вызывается функция, никакие действия по вычислению элементов не выполняются, а в созданной последовательности сохраняется лишь «программа» их вычисления. Активизируется эта программа только при переборе элементов последовательности в цикле `foreach` или в других случаях, требующих создания и обработки ее элементов. Но и в этом случае элементы не будут размещаться в памяти одновременно: они вычисляются последовательно и сразу разрушаются после обработки. Обратите внимание на то, что в самой функции `OddGen` никак не сохраняются найденные нечетные числа. Они пересылаются «наружу» оператором `yield`, после чего сразу «забываются». Именно так и работает механизм определения и использования последовательности.

Нет никаких препятствий к тому, чтобы определить с помощью `yield` функцию, возвращающую бесконечную последовательность. Например, реализовать генератор последовательности всех положительных нечетных чисел можно следующим образом:

```
function AllOddGen: sequence of integer;
begin
  var i := 1;
  while True do
    begin
      yield i;
      i += 2;
    end;
  end;
```

Для использования нужной начальной части этой последовательности надо, как обычно, использовать соответствующий вспомогательный запрос, например уже известный нам запрос `Take`:

```
Task('Array1'); // Вариант 5
AllOddGen.Take(ReadInteger).Write;
```

Пример функции `AllOddGen` еще более наглядно демонстрирует особенности функций с конструкцией `yield`. Несмотря на то что в теле функции присутствует бесконечный цикл, ее вызов не приводит к заикливанию. Фактически, как уже было сказано, ее вызов вообще не приведет к выполнению содержащихся в ней операторов. Ее тело будет выполняться только тогда, когда последовательности потребуется перебрать свои элементы, и итерации будут повторяться лишь до тех пор, пока не будут получены те элементы, которые требуются.

Если в функции, возвращающей последовательность, не будет вызвано ни одной конструкции `yield`, то будет возвращена пустая последовательность. Подобная ситуация возникнет, например, при вызове `OddGen(0)`.

### 3.7. Ввод массивов и последовательностей.

#### *Инвертирование. Срезы*

Рассмотрев различные способы создания массивов и последовательностей, мы пока еще не обсудили варианты организации ввода этих структур.

Очевидным вариантом организации ввода массива является цикл `for`. Вначале запрашивается размер массива, затем в цикле вводятся его элементы. Для большей наглядности ввод данных следует сопровождать приглашающими сообщениями. Благодаря тому, что в `PascalABC.NET` для организации ввода предусмотрены не только традиционные процедуры, но и функции (см. п. 1.2), ввод размера можно выполнить непосредственно при инициализации массива. Кроме того, инициализацию можно объединить с описанием массива. В результате ввод, например, массива целых чисел можно организовать следующим образом:

```
var a := new integer[ReadInteger('Введите размер массива:');
Print('Введите элементы массива:');
for var i := 0 to a.Length - 1 do
    a[i] := ReadInteger;
```

При выполнении этого фрагмента вначале будет выведено приглашение «Введите размер массива», после которого пользователь должен ввести размер (целое число). Введенный размер сразу будет передан в конструктор массива, что обеспечит выделение памяти под требуемое число элементов. Затем выводится приглашение на ввод элементов, после чего в цикле выполняется сам ввод. Заметим, что элементы можно вводить либо в одной строке, разделяя их пробелами, либо в различных строках, нажимая

после ввода каждого элемента клавишу Enter. Можно также комбинировать эти варианты.

Для проверки правильности ввода можно использовать процедуру Write:

```
Write(a);
```

Приведем результат выполнения предыдущего фрагмента, дополненного процедурой Write:

**Введите размер массива: 5**

**Введите элементы массива: 2 3 4 5 6**

**[2,3,4,5,6]**

Аналогичным образом можно организовать ввод массива при выполнении заданий с использованием задачника Programming Taskbook. Необходимо лишь убрать из приведенного выше кода все приглашения к вводу.

Для ввода массивов основных скалярных типов (целочисленного, вещественного и строкового) в стандартной библиотеке PascalABC.NET, а также в модуле РТ4 задачника Programming Taskbook предусмотрены специальные функции. Эти функции имеют имена ReadArrInteger, ReadArrReal и ReadArrString и возвращают введенный массив соответствующего типа. Функции из стандартной библиотеки имеют один обязательный параметр целого типа — размер  $n$  вводимого массива. Имеется также вариант с двумя параметрами: (prompt,  $n$ ), где первый параметр prompt (строкового типа) определяет текст приглашения к вводу.

Используя соответствующую функцию, ввод целочисленного массива вместе со всеми приглашающими сообщениями можно организовать в виде *единственного* оператора:

```
var a := ReadArrInteger('Введите элементы массива:',  
    ReadInteger('Введите размер массива:'));
```

Вид окна вывода при выполнении этого оператора (дополненного процедурой Write) будет в точности совпадать с видом, полученным при выполнении первого варианта ввода, использующего цикл.

Обратите внимание на то, что во всех приглашающих сообщениях не нужно указывать завершающий пробел, поскольку он будет добавлен автоматически.

У функций ввода для массивов, предусмотренных в модуле РТ4 задачника Programming Taskbook, отсутствует вариант с приглашающими сообщениями. Вместо него предусмотрен вариант *без параметров*, обеспечивающий, наряду с вводом элементов массива, предварительный ввод его размера. Вариант без параметров является наиболее удобным и используется чаще всего, так как в большинстве задач размер массива должен вво-

даться непосредственно перед вводом самих элементов. В подобной ситуации организация ввода становится предельно простой, например:

```
var a := ReadArrInteger;
```

Ситуации, в которых при решении задач требуется использовать вариант функций ввода с параметром  $n$  — размером массива, возникают достаточно редко: например, если в задаче надо использовать массив, размер которого (скажем, 10) явно указан в формулировке:

```
var a := ReadArrInteger(10);
```

Параметр  $n$  необходим и в ситуации, когда в задаче даются два массива *одинакового* размера, причем размер указывается один раз — перед первым массивом. В этом случае можно использовать два варианта ввода. В первом варианте вначале вводится размер  $n$ , а затем он указывается при вводе каждого массива:

```
var n := ReadInteger;  
var a := ReadArrInteger(n);  
var b := ReadArrInteger(n);
```

Можно, однако, обойтись без использования переменной  $n$ , если ввести первый массив с помощью функции без параметров, а затем указать размер *уже введенного* массива в функции для ввода второго массива:

```
var a := ReadArrInteger;  
var b := ReadArrInteger(a.Length);
```

Обратимся к последовательностям. Можно ли организовать ввод так, чтобы введенные данные сразу помещались в последовательность? На первый взгляд, ответ должен быть отрицательным, поскольку последовательность является *неизменяемой* структурой: создав последовательность с помощью одного из генераторов, мы уже не сможем изменить ее элементы *по отдельности*. Однако мы можем применить к последовательности *запрос*, формирующий на ее основе другую последовательность. Мы уже знаем несколько подобных запросов, в том числе `Select`. А ведь в запросе `Select` (в указываемом в нем лямбда-выражении) мы вполне можем организовать ввод данных! При этом для создания исходной последовательности мы можем использовать какую-либо простейшую функцию-генератор, например, `SeqFill` или `Range` (или `Repeat` с последующим запросом `Take`). Реализуем эту идею для ввода целочисленной последовательности:

```
var a := SeqFill(ReadInteger('Введите размер и элементы:'), 0)  
.Select(e -> ReadInteger);
```

Следует обратить внимание на пустые круглые скобки, указанные после функции `ReadInteger`. В данном случае они необходимы, поскольку при их отсутствии компилятор решит, что в новой последовательности надо

сохранять *саму функцию* ReadInteger (как значение процедурного типа), а не *результат* ее выполнения.

**Замечание.** Как правило, при вызове функций без параметров указанных проблем не возникает, так как компилятор по контексту верно определяет нужное действие. Однако в некоторых ситуациях (как в приведенной выше) по контексту определить требуемое действие нельзя, и поэтому приходится явно вводить скобки, однозначно указывающие на то, что требуется выполнить вызов функции. Следует иметь в виду данную особенность вызовов функций без параметров, и если при компиляции или выполнении оператора, содержащего подобный вызов, возникают проблемы, первым делом добавлять к коду круглые скобки. Можно также применять стиль программирования, при котором после имен подпрограмм при их вызове *всегда* указываются скобки (в том числе пустые). Заметим, что в большинстве языков программирования указание скобок при вызове подпрограмм является обязательным.

Если добавить к этому оператору оператор отладочной печати вида Write(a), то при запуске данного фрагмента и вводе тех же данных, что и для предыдущего примера с массивом, мы получим такой текст:

```
Введите размер и элементы: 5 2 3 4 5 6  
[2,3,4,5,6]
```

Для большей наглядности можно перейти на новую строку после ввода размера последовательности; это никак не повлияет на результат.

Следует, однако, учитывать, что ввод данных в последовательность принципиальным образом отличается от ввода данных в массив. Чтобы в этом убедиться, достаточно после ввода данных попытаться *дважды* вывести эти данные, например:

```
Writeln(a);  
Write(a);
```

В случае массива ничего неожиданного не произойдет: содержимое массива просто выведется дважды. Однако при обработке последовательности результат будет иным: после первого вывода содержимого введенной последовательности программа *опять попытается ввести такое же количество элементов последовательности* (при этом никакого приглашающего сообщения выведено не будет). И если проанализировать данную ситуацию, то можно понять, что именно такое поведение и является для последовательностей наиболее естественным. Ключевая идея состоит в том, что последовательность *не хранит в памяти свои элементы*. Она генерирует их (тем или иным образом) лишь тот момент, когда эти элементы требуется обработать (например, вывести на экран). Если алгоритм генерации задан явным образом, то последовательность всегда генерирует одинаковый набор данных, и по поведению программы невозможно опре-

делить, в какой момент эта генерация производится. Однако при вводе данных с клавиатуры ситуация иная: для того чтобы повторно сгенерировать последовательность необходимо ее повторно *ввести*, так как прежние значения элементов уже обработаны и забыты (последовательность помнит лишь свой размер).

Необходимо также учитывать следующее. Ввод массива выполняется при выполнении того фрагмента, в котором ввод запрограммирован. Даже если бы мы не указали после этого фрагмента процедуру Write, память под массив была бы выделена и в нее занесены введенные значения элементов. Действия же по вводу последовательности фактически производятся только в момент перебора ее элементов. Поэтому если бы мы включили в программу только приведенный выше единственный оператор, определяющий последовательность *a*, то при выполнении этого фрагмента были бы выполнены только действия по вводу единственного числа — *размера* последовательности (в чем можно легко убедиться, закомментировав идущие далее вызовы процедуры Write: если в этом случае после ввода первого числа нажать Enter, то программа немедленно завершится).

Описанная особенность ввода последовательностей означает, что организовывать подобный ввод имеет смысл только если *сразу после ввода* последовательность будет обработана требуемым образом (обычно с применением цепочки запросов) и получен результат, после чего в программе не будет производиться обращений к этой последовательности. Заметим, что в такой ситуации обычно не требуется связывать последовательность с какой-либо переменной, так как в дальнейшем эта переменная в программе не будет использоваться.

Как и для массивов, для последовательностей с числовыми и строковыми элементами в PascalABC.NET предусмотрены специальные функции, упрощающие их ввод. Их имена аналогичны именам функций для ввода массивов, за исключением того, что текст «Arr» надо в них заменить на «Seq»: ReadSeqInteger, ReadSeqReal, ReadSeqString.

Например, использованный ранее оператор для ввода целочисленной последовательности можно записать гораздо короче:

```
var a := ReadSeqInteger  
    (ReadInteger('Введите размер и элементы:'));
```

Можно также указать отдельные приглашения для ввода размера и элементов, как в ранее рассмотренном примере с массивом:

```
var a := ReadSeqInteger('Введите элементы:',  
    ReadInteger('Введите размер:'));
```

Еще раз подчеркнем, что ввод элементов будет выполняться только в момент перебора элементов последовательности.

Варианты функций `ReadSeqInteger`, `ReadSeqReal`, `ReadSeqString` реализованы и в модуле PT4 электронного задачника. Как и в случае функций для ввода массивов, имеются варианты этих функций без параметров и с одним параметром — размером вводимой последовательности.

**Замечание.** При реализации вариантов функций `ReadSeqInteger`, `ReadSeqReal`, `ReadSeqString` для электронного задачника `Programming Taskbook` было решено отказаться от «отложенного» характера их выполнения: ввод элементов последовательности выполняется ими *немедленно*, и введенные элементы сохраняются в памяти, подобно введенным элементам массивов. Отличие этих функций от аналогичных функций для ввода массивов состоит лишь в том, что возвращаемый ими результат имеет тип *последовательности* (а не массива). Подобная модификация не дает проявиться особенностям ввода последовательностей, описанным выше, и благодаря этому избавляет начинающего программиста от многих ошибок, которые ему на начальном этапе изучения последовательностей было бы трудно понять и исправить.

Обсудив средства ввода массивов и последовательностей, мы можем обратиться к задачам, в которых дается некоторый набор данных, требующий обработки. Одной из простейших подобных задач является задача **Array7**, в которой дается размер массива и его элементы и требуется вывести элементы массива в обратном порядке. Очевидное решение, использующее цикл, выглядит следующим образом:

```
Task('Array7'); // Вариант 1
var a := ReadArrReal;
for var i := a.Length - 1 downto 0 do
  Write(a[i]);
```

Чтобы получить более краткое решение, можно использовать запрос, *инвертирующий* последовательность, т. е. располагающий ее элементы в обратном порядке (точнее, данный запрос возвращает *новую* последовательность, которая является инвертированной по отношению к исходной). Этот запрос имеет имя `Reverse` и не требует параметров. Полученную в результате применения этого запроса инвертированную последовательность можно сразу переслать задачнику, поэтому в решении не потребуется использовать ни одной переменной:

```
Task('Array7'); // Вариант 2
ReadSeqReal.Reverse.Write;
```

Запрос `Reverse` можно применить и к массиву, так как любой массив может быть неявно преобразован в последовательность:

```
Task('Array7'); // Вариант 2a
ReadArrReal.Reverse.Write;
```



С другой стороны, если у нас уже есть массив, то его можно инвертировать с помощью специальной *процедуры* `Reverse`:

```
Task('Array7'); // Вариант 3
var a := ReadArrReal;
Reverse(a);
a.Write;
```

Данное решение состоит из трех операторов и выглядит длиннее предыдущих. Собственно говоря, этот вариант программы выполняет не совсем те действия, которые требуются в задании: вместо того чтобы просто вывести элементы массива в обратном порядке, программа *изменяет порядок расположения его элементов в памяти*.

У *процедуры* `Reverse` имеется важная особенность, отсутствующая у одноименного запроса: процедура может иметь два дополнительных параметра `start` и `count`, при наличии которых инвертируется не весь массив, а только `count` его элементов, начиная с элемента с индексом `start`.

Запрос `Reverse` является примером запроса, при котором из набора элементов в определенном порядке извлекается некоторая часть. В других ситуациях может потребоваться извлечь элементы через один или извлечь только первую или вторую половину элементов. Подобные фрагменты исходных наборов называются его *срезами*. Многие современные языки программирования обладают развитыми средствами получения срезов. Такие средства есть и в `PascalABC.NET`.

Для массивов (а также для списков `List`, рассматриваемых во втором выпуске настоящей серии [1, гл. 2], и текстовых строк `string`) срезы можно получить с помощью *расширенного варианта операции индексирования*. Наиболее общая форма среза для массива `a` имеет вид `a[from:to:step]`, где `from` определяет индекс первого элемента, включаемого в срез, а `step` — шаг изменения индекса. Параметр `to` имеет несколько более сложный смысл. Если шаг положителен, то параметр `to` равен индексу, *следующему* за индексом последнего элемента, включаемого в срез, а если шаг отрицателен, то параметр `to` равен индексу, *предшествующему* индексу последнего элемента, включаемого в срез<sup>3</sup>.

Если массив имеет размер `n`, то параметр `from` может принимать целочисленные значения от 0 до `n - 1`, а параметр `to` — значения от `-1` до `n`. Параметр `step` может быть любым ненулевым целым числом. При нарушении любого из этих правил возникает ошибка времени выполнения.

Если `from` больше или равен `to` и `step` положителен или если `from` меньше или равен `to` и `step` отрицателен, то возвращается *пустой срез* (не содержащий ни одного элемента).

---

<sup>3</sup> Подобное определение параметра `to` объясняется тем, что именно так определяются срезы в языке `Python`, из которого был позаимствован синтаксис среза.

Любой из параметров среза может отсутствовать. Если не указывается параметр `step`, то не указывается также и предшествующее ему двоеточие. Двоеточие между параметрами `from` и `to` указывается всегда, даже если отсутствуют оба этих параметра. При отсутствии параметра `step` его значение полагается равным 1. Смысл отсутствующих параметров `from` и `to` зависит от знака параметра `step`. Если шаг положителен, то отсутствующий параметр `from` полагается равным 0, а отсутствующий параметр `to` — равным `n` (мы по-прежнему через `n` обозначаем размер массива). Если шаг отрицателен, то отсутствующий параметр `from` полагается равным `n - 1`, а отсутствующий параметр `to` — равным `-1`.

Все перечисленные правила полностью определяют способ задания любых срезов. Приведем примеры, иллюстрирующие эти правила:

```
var a := Arr(0,1,2,3,4,5,6,7,8,9);
Writeln(a[::-1]); // [9,8,7,6,5,4,3,2,1,0]
Writeln(a[:2]); // [0,2,4,6,8]
Writeln(a[:-2]); // [9,7,5,3,1]
Writeln(a[5:2]); // [0,2,4]
Writeln(a[5::-2]); // [5,3,1]
Writeln(a[:5]); // [0,1,2,3,4]
Writeln(a[5:-1]); // [9,8,7,6]
Writeln(a[5:]); // [5,6,7,8,9]
Writeln(a[5::-1]); // [5,4,3,2,1,0]
Writeln(a[5:0:-1]); // [5,4,3,2,1]
Writeln(a[5:5]); // []
```

Заметим, что конструкция `a[:]` также не нарушает никаких из перечисленных выше правил; она может использоваться для получения *копии* всего массива `a` (см. п. 5.1).

Используя срезы, мы можем привести еще один короткий вариант решения задачи **Array7**:

```
Task('Array7'); // Вариант 4
ReadArrReal[::-1].Print;
```

Рассмотрим также задачу **Array12**, в которой требуется вывести элементы массива с четными порядковыми номерами в порядке возрастания этих номеров. При анализе этой задачи необходимо учитывать, что под *порядковым номером* элемента понимается номер, отсчитываемый от единицы. Поэтому для динамических массивов, всегда индексируемых от нуля, порядковый номер любого элемента будет на 1 больше его индекса. Это означает, что в задаче **Array12** требуется вывести все элементы массива с *нечетными индексами* (в порядке их возрастания):

```
Task('Array12'); // Вариант 1
ReadArrReal[1::2].Print;
```

Для последовательностей аналогичную конструкцию для срезов применять нельзя, так как у последовательностей отсутствует операция индексирования. Однако срез для последовательности (в виде новой последовательности) можно получить с помощью специального запроса `Slice` со следующими параметрами:

```
a.Slice(from, step: integer[; count: integer]);
```

Здесь `from` и `step`, как и раньше, означают соответственно индекс начального элемента среза и шаг, а необязательный параметр `count` означает наибольшее количество элементов, добавляемых в срез (размер полученного среза может быть меньше, чем `count`, если в исходной последовательности окажется недостаточно элементов). Важным дополнительным ограничением запроса `Slice` для последовательностей является то, что шаг `step` всегда должен быть *положительным*.

Запрос `Slice` (как и любые другие запросы для последовательностей) можно также применять к массивам и другим коллекциям. При этом для массивов, списков `List` и текстовых строк предусмотрены особые реализации этого запроса, в которых шаг `step` может быть как положительным, так и отрицательным.

Поскольку для последовательностей шаг `step` в запросе `Slice` не может быть отрицательным, мы не сможем использовать этот запрос при решении задачи **Array7** с помощью последовательностей. Однако применить аналогичный запрос для массивов вполне допустимо, хотя полученное решение будет более длинным, чем то, которое использовало срезы в виде индексов (прежде всего, из-за необходимости явного указания начального индекса, равного индексу последнего элемента массива):

```
Task('Array7'); // Вариант 5
var a := ReadArrReal;
a.Slice(a.Length - 1, -1).Write;
```

Задачу **Array12**, в которой требуется получить срез с положительным шагом, вполне можно решить с помощью последовательности:

```
Task('Array12'); // Вариант 2
ReadSeqReal.Slice(1, 2).Write;
```

## Глава 4. Запросы

При обсуждении в предыдущей главе базовых средств работы с массивами и последовательностями мы неоднократно отмечали, что основным способом обработки последовательностей является применение к ним *запросов* — методов, возвращающих преобразованную последовательность или какие-либо характеристики исходной последовательности. В качестве примеров мы использовали запросы `Select`, `Sum`, `Take`, `Reverse`, `Slice` и убежились, что применение подобных запросов приводит к краткому и наглядному коду.

Однако для эффективной обработки последовательностей (а также любых коллекций) с помощью запросов необходимо знать, какие запросы предусмотрены в `PascalABC.NET`, каковы их особенности и дополнительные возможности. Этим вопросам и посвящена данная глава. В ней рассматриваются как запросы, входящие в стандартную библиотеку платформы `.NET`, так и дополнительные запросы, реализованные в библиотеке `PascalABC.NET`.

Исключены из рассмотрения лишь запросы `AsEnumerable`, `Cast` и `OfType`, входящие в стандартную библиотеку `.NET`, поскольку два первых запроса используются крайне редко, а описание особенностей применения запроса `OfType` требует привлечения понятий объектно-ориентированного программирования, выходящих за рамки настоящего пособия (по поводу этих запросов см., например, [3, п. 10.3.7, 10.3.8]). Кроме того, не обсуждаются особенности вариантов запросов, содержащих параметры типа `IComparer` или `IEqualityComparer`, поскольку эти типы также не рассматриваются в данном пособии (см. [3, п. 8.6, 9.7]). Отметим лишь, что параметры типа `IComparer` входят в варианты запросов `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending` (см. п. 4.2), а параметры типа `IEqualityComparer` — в варианты запросов `Contains`, `SequenceEqual` (п. 4.1), `Distinct`, `Union`, `Intersect`, `Except` (п. 4.2), `Join`, `GroupJoin` (п. 4.4), `GroupBy` (п. 4.5), `ToDictionary`, `ToLookup` (п. 4.6). Эти необязательные параметры всегда указываются *последними* в списке параметров соответствующего запроса.

### 4.1. Поэлементные операции, квантификаторы и агрегирование

К простейшим стандартным запросам можно отнести запросы, которые позволяют:

- обратиться к отдельным элементам последовательности,
- проверить, удовлетворяют ли все или некоторые элементы определенному условию (*запросы-квантификаторы*),
- найти какую-либо общую (*агрегирующую*) характеристику элементов последовательности.

Здесь и далее при описании групп запросов (и других методов) мы будем указывать их имена, параметры и тип возвращаемого значения, заключая необязательные параметры в квадратные скобки. В первой строке описания группы методов слева будет указываться ее название, а справа — тип структуры, к которой должны применяться описываемые методы.

Начнем с первой группы запросов. Обратите внимание на то, что все эти запросы возвращают скалярное значение.

#### **Поэлементные операции**

Методы sequence of T

**First**([pred: T -> boolean]): T

**Last**([pred: T -> boolean]): T

**Single**([pred: T -> boolean]): T

**ElementAt**(index: integer): T

**FirstOrDefault**([pred: T -> boolean]): T

**LastOrDefault**([pred: T -> boolean]): T

**SingleOrDefault**([pred: T -> boolean]): T

**ElementAtOrDefault**(index: integer): T

Запрос **First** возвращает первый элемент последовательности, запрос **Last** — последний элемент, запрос **Single** — единственный элемент последовательности, а запрос **ElementAt** — элемент с индексом *index*. При указании дополнительного параметра — предиката *pred* запросы обрабатывают только те элементы последовательности, которые удовлетворяют указанному предикату.

Запросы, имеющие суффикс **OrDefault**, выполняются аналогично запросам без суффикса, однако при отсутствии требуемого элемента запросы с суффиксом **OrDefault** возвращают нулевое значение для типа T (тогда как запросы без **OrDefault** *возбуждают исключение*<sup>4</sup>). Напомним, что нулевым

<sup>4</sup> Если в программе возбуждается исключение (говорят также, что в программе *возникает исключительная ситуация*), то при отсутствии в ней специальных средств для перехвата и обработки исключения (так называемых *try-блоков*) происходит аварийное завершение программы. Использование *try-блоков* в Паскале кратко описывается, например, в [2, п. 12.5].

значением для числовых типов является число 0 данного типа, для символа — это символ с кодом 0, а для строки — константа `nil`.

Заметим, что запросы `Single` и `SingleOrDefault` всегда возбуждают исключение, если количество требуемых элементов больше одного. Они действуют по-разному, если требуемые элементы отсутствуют: в этом случае запрос `Single` возбуждает исключение, а запрос `SingleOrDefault` возвращает нулевое значение.

Как уже отмечалось в предыдущей главе (см. п. 3.4), запрос `ElementAt`, как правило, выполняется медленно, поэтому его не следует использовать для перебора всех элементов последовательности в цикле.

Перейдем к *запросам-квантификаторам*.

### **Квантификаторы**

Методы `sequence of T`

**All**(pred: T -> boolean): boolean

**Any**([pred: T -> boolean]): boolean

**Contains**(value: T): boolean

**SequenceEqual**(seq2: sequence of T): boolean

Запрос `All` возвращает `True`, если *все* элементы последовательности удовлетворяют предикату `pred` (для пустой последовательности запрос `All` всегда возвращает `True`). Запрос `Any` возвращает `True`, если *какие-либо* элементы последовательности удовлетворяют предикату `pred` (для пустой последовательности запрос `Any` всегда возвращает `False`). Если параметр `pred` отсутствует, то запрос `Any` возвращает `True`, если последовательность является непустой.

Запрос `Contains` возвращает `True`, если в последовательности имеется хотя бы один элемент со значением `value`. Запрос `SequenceEqual` возвращает `True`, если вызвавшая его последовательность совпадает с последовательностью `seq2` (последовательности считаются равными, если они содержат одни и те же элементы в том же самом порядке).

При перечислении *запросов агрегирования* мы впервые встречаемся с запросами, расширяющими возможности стандартных запросов `.NET` и специально добавленными для этой цели в библиотеку `PascalABC.NET`. Перед именами таких запросов здесь и далее будет указываться метка-звездочка `*`.

### **Агрегирование**

Методы `sequence of T`

**Count**([pred: T -> boolean]): integer

**Average**([sel: T -> числовой\_тип]): real

**Sum**([sel: T -> числовой\_тип]): числовой\_тип

**Max**: T

**Max**(sel: T -> TKey): TKey

**Min**: T

**Min**(sel: T -> TKey): TKey

- \* **MaxBy**(sel: T -> TKey): T
- \* **MinBy**(sel: T -> TKey): T
- \* **LastMaxBy**(sel: T -> TKey): T
- \* **LastMinBy**(sel: T -> TKey): T
- \* **JoinIntoString**([delim: string]): string
- Aggregate**(agg: (T, T) -> T): T
- Aggregate**(seed: TRes; agg: (TRes, T) -> TRes): TRes
- Aggregate**(seed: TAcc; agg: (TAcc, T) -> TAcc;  
finalSel: TAcc -> TRes): TRes

Запрос `Count` возвращает размер последовательности (или количество элементов, удовлетворяющих предикату `pred`, если он указан). Имеется также вариант этого запроса — запрос `LongCount`, возвращающий значение типа `int64`. Тип `int64` — это целочисленный тип размера 8 байт, позволяющий хранить очень большие целые числа: от  $-2^{63}$  ( $= -9223372036854775808$ ) до  $2^{63} - 1$  ( $= 9223372036854775807$ ).

Запросы `Average`, `Sum`, `Min` и `Max` находят соответственно среднее арифметическое, сумму, минимум и максимум элементов последовательности или, при наличии лямбда-выражения `sel` (*селектора*), тех значений, в которые эти элементы преобразуются указанным селектором. Запросы `Average` и `Sum` возбуждают исключение, если вызываются без указания селектора для нечисловых последовательностей. Запросы `Min` и `Max` возбуждают исключение, если вызываются для обработки данных, для которых не определены операции отношения «меньше»–«больше». В случае пустых последовательностей запрос `Sum` возвращает нулевое значение, а запросы `Average`, `Min` и `Max` возбуждают исключение.

Запросы `MaxBy` и `MinBy` возвращают первый из элементов последовательности с максимальной или, соответственно, минимальной характеристикой (*ключом*), вычисляемой с помощью параметра-селектора `sel`. Запросы `LastMaxBy` и `LastMinBy` отличаются от `MaxBy` и `MinBy` тем, что возвращают *последний* из элементов последовательности с максимальным или минимальным ключом. Важно понимать отличие этих запросов от запросов `Max` и `Min` с аналогичным параметром-селектором: запросы `Max` и `Min` возвращают максимальный или минимальный *ключ*, тогда как запросы `MaxBy`, `MinBy`, `LastMaxBy` и `LastMinBy` возвращают *элемент* последовательности, соответствующий максимальному или минимальному ключу.

Запрос `JoinIntoString` возвращает строковое представление последовательности, которое формируется следующим образом: все элементы последовательности преобразуются к их стандартному строковому представлению, а между ними вставляется разделитель `delim`. Если параметр `delim` отсутствует, то он считается равным *пробелу* (за исключением случая, когда элементы последовательности имеют тип `char`; в этом случае отсутствующий параметр `delim` считается *пустой строкой*). Следует подчерк-

нуть, что запрос `JoinIntoString` можно применять к последовательностям *любого типа*, так как в `PascalABC.NET` для любого типа определен метод `ToString`, преобразующий значения этого типа в их строковое представление.

Запрос `Aggregate` позволяет определять новые операции агрегирования. Он реализован в трех вариантах. В каждом из них присутствует лямбда-выражение `agg`, определяющее агрегирующее действие. Его первый параметр является *аккумулятором*, накапливающим результат, а второй параметр содержит аккумулялируемое значение (т. е. значение, которое должно добавляться к аккумулятору). Варианты запроса отличаются дополнительными действиями, связанными с заданием аккумулятора.

В простейшем случае, когда лямбда-выражение `agg` является единственным параметром, оно обрабатывает (аккумулярует) все элементы последовательности, начиная со второго, а первый элемент последовательности используется в качестве начального значения аккумулятора.

Если дополнительно указывается параметр `seed`, то он используется в качестве начального значения аккумулятора, а лямбда-выражение `agg` обрабатывает все элементы последовательности, начиная с первого.

Если дополнительно указывается лямбда-выражение `finalSel` (*финальный селектор*), то после нахождения аккумулятора он передается этому селектору, который преобразует значение аккумулятора в результат, возвращаемый запросом.

Проиллюстрируем использование описанных запросов на примерах выполнения некоторых заданий из группы `LinqBegin` электронного задачника `Programming Taskbook`. Эта группа посвящена знакомству со стандартными запросами, поэтому ее название содержит аббревиатуру `LINQ` (`Language Integrated Query`, т. е. «запрос, интегрированный в язык») — название технологии, позволяющей использовать в языках платформы `.NET` последовательности и запросы. Поскольку в данном пособии не обсуждаются вопросы, связанные с обработкой текстовых данных, будем использовать только те задачи, в которых даются числовые последовательности.

В задаче **`LinqBegin1`** требуется вывести первый положительный и последний отрицательный элементы целочисленной последовательности (предполагается, что последовательность обязательно содержит как положительные, так и отрицательные элементы):

```
Task('LinqBegin1');
var a := ReadSeqInteger;
Write(a.First(e -> e > 0), a.Last(e -> e < 0));
```



При решении задачи мы использовали запросы `First` и `Last` с параметрами-предикатами. Поскольку для исходной последовательности требовалось вызвать два запроса, пришлось связать с ней идентификатор `a`.

**Замечание.** Полезно напомнить, что произошло бы, если бы последний оператор был включен в «обычную» программу, не связанную с задачей, в которой последовательность `a` вводится с помощью одного из вариантов функции `ReadSeqInteger`, входящих в стандартную библиотеку `PascalABC.NET` (например, `ReadSeqInteger(10)`). В этом случае потребовалось бы *дважды* вводить элементы исходной последовательности с клавиатуры — при выполнении *каждого* из использованных в последнем операторе запросов. Чтобы подобные проблемы при выполнении заданий не возникали (и не приводили к сообщениям вида «*Попытка ввода лишних исходных данных*»), функции ввода последовательностей в модуле `PT4` были специальным образом модифицированы (см. последнее замечание в п. 3.7).

В задаче **LinqBegin2** дается цифра `d` и целочисленная последовательность. Требуется вывести первый положительный элемент, оканчивающийся цифрой `d`, или число `0`, если таких элементов в последовательности нет:

```
Task('LinqBegin2');
var d := ReadInteger;
Write(ReadSeqInteger.FirstOrDefault(e -> (e > 0
    and (e mod 10 = d))));
```

В данном случае, поскольку требуемый элемент может отсутствовать, необходимо использовать запрос с суффиксом `OrDefault`. Мы не стали связывать введенную последовательность с переменной, сразу применив к функции `ReadSeqInteger` требуемый запрос.

В задаче **LinqBegin9** требуется вывести минимальный положительный элемент целочисленной последовательности или число `0`, если последовательность не содержит положительных элементов.

Один из очевидных вариантов решения этой задачи — предварительная *фильтрация* исходной последовательности (см. следующий пункт), однако можно обойтись и теми средствами, которые описаны в данном пункте. Придется лишь дополнительно проанализировать результат, возвращенный запросом:

```
Task('LinqBegin9');
var a := ReadSeqInteger.Min(e -> e <= 0 ? integer.MaxValue : e);
Write(a = integer.MaxValue ? 0 : a);
```

В запросе `Min` мы использовали селектор, который сохраняет неизменными все положительные элементы последовательности, а отрицательные заменяет на максимальное значение типа `integer` (доступное в виде свойства

`integer.MaxValue`). Для этого мы использовали в селекторе *тернарную операцию* вида условие ? значение1 : значение2, которая вычисляет условие и возвращает значение1, если условие истинно, и значение2, если условие ложно (синтаксис тернарной операции заимствован языком Pascal-ABC.NET из языков семейства C).

Если в последовательности имеется хотя бы один положительный элемент, то запрос `Min` вернет минимальный из таких элементов, а если положительных элементов в последовательности нет, то будет возвращено значение `integer.MaxValue`. В этом особом случае программа выводит число 0 (еще раз используя тернарную операцию).

В задании **LinqBegin15** требуется вычислить *факториал*  $N!$  целого числа  $N$  (равный произведению всех целых чисел от 1 до  $N$ :  $N! = 1 \cdot 2 \cdot \dots \cdot N$ ), используя генератор `Range` (описанный в начале п. 3.5) и запрос `Aggregate`. При этом факториал требуется вычислять в виде вещественного числа, чтобы избежать целочисленного переполнения для больших значений  $N$ :

```
Task('LinqBegin15');
Write(Range(1, ReadInteger).Aggregate(1.0, (a, e) -> a * e));
```

Главной особенностью этого решения является использование второго из вариантов запроса `Aggregate`, в котором указывается начальное значение аккумулятора в виде 1.0 (т. е. *вещественной* единицы). Заметим, что запрос `Aggregate((a, e) -> a * e)` тоже позволяет правильно вычислять факториал (в виде целого числа), но только в ситуации, когда значение факториала не превосходит максимального значения для типа `integer` (в противном случае из-за целочисленного переполнения будет получено неверное число, которое, в частности, может оказаться отрицательным).

## 4.2. Фильтрация, сортировка, комбинирование и расщепление

В данном пункте мы рассмотрим запросы, позволяющие извлекать из исходной последовательности требуемые элементы (*фильтрация*), изменять порядок следования элементов (*сортировка и инвертирование*), *объединять* элементы нескольких последовательностей в одну и, наоборот, *расщеплять* исходную последовательность на две.

Все запросы этих групп возвращают новые последовательности (или кортежи из двух последовательностей), тип элементов которых *совпадает* с типом элементов исходных последовательностей. Как и все запросы, возвращающие последовательности, они характеризуются «отложенным» выполнением: реальное формирование элементов полученных последовательностей будет происходить только при последующем *переборе* этих элементов (в цикле `foreach`, при печати или при преобразовании последова-

тельности в структуру, хранящую свои данные в памяти, — например, в массив).

### Фильтрация

Методы `sequence of T`

**Where**(pred: (T[, integer]) -> boolean): sequence of T

**TakeWhile**(pred: (T[, integer]) -> boolean): sequence of T

**SkipWhile**(pred: (T[, integer]) -> boolean): sequence of T

**Take**(count: integer): sequence of T

\* **TakeLast**(count: integer): sequence of T

**Skip**(count: integer): sequence of T

\* **Slice**(from, step: integer[; count: integer]): sequence of T

**Distinct**: sequence of T

Запрос `Where` является наиболее часто используемым запросом фильтрации. Он возвращает последовательность, содержащую только те элементы исходной последовательности, которые удовлетворяют предикату `pred` (порядок элементов не изменяется). Запрос `TakeWhile` заносит в выходную последовательность элементы исходной последовательности, пока предикат `pred` возвращает значение `True`; запрос `SkipWhile` пропускает начальные элементы исходной последовательности, пока предикат возвращает значение `True`, после чего заносит в выходную последовательность все оставшиеся элементы. Обратите внимание на то, что во всех трех запросах предикат `pred` может содержать дополнительный параметр — *индекс* анализируемого элемента (индексация начинается с нуля).

Запросы `Take`, `TakeLast` и `Skip`, подобно методам `TakeWhile` и `SkipWhile`, возвращают начальную или конечную часть исходной последовательности: `Take` возвращает первые `count` элементов, `TakeLast` — последние `count` элементов, а `Skip` пропускает первые `count` элементов и возвращает оставшиеся. Элементы в полученной последовательности располагаются в том же порядке, что и в исходной. Если параметр `count` больше размера исходной последовательности, то запросы `Take` и `TakeLast` возвращают все элементы исходной последовательности, а запрос `Skip` — пустую последовательность. Запрос `Take` мы уже использовали в предыдущей главе при обсуждении бесконечных последовательностей (см. п. 3.5).

Запрос `Slice` возвращает *срез* исходной последовательности; он подробно обсуждался в п. 3.7. Напомним лишь, что параметр `step` для этого запроса должен быть положительным.

Запрос `Distinct` возвращает последовательность без повторяющихся элементов (в последовательности оставляются только первые вхождения повторяющихся элементов; повторяющиеся элементы не обязаны располагаться в исходной последовательности подряд). Например, для исходной последовательности `[1,2,1,2,4,2,3,1]` запрос `Distinct` вернет `[1,2,4,3]`.

**Сортировка**

Методы sequence of T

**OrderBy**(T -> TKey): sequence of T**ThenBy**(T -> TKey): sequence of T\* **Sorted**: sequence of T**OrderByDescending**(T -> TKey): sequence of T**ThenByDescending**(T -> TKey): sequence of T\* **SortedDescending**: sequence of T**Инвертирование**

Метод sequence of T

**Reverse**: sequence of T

Запрос **OrderBy** возвращает элементы исходной последовательности, отсортированные по указанному *ключу* в порядке возрастания. Ключ определяется лямбда-выражением и должен иметь тип **TKey**, для которого определены операции отношения «меньше»–«больше».

Запрос **ThenBy** используется, если последовательность требуется отсортировать по *набору ключей*; этот метод переупорядочивает (в порядке возрастания своего ключа) только те элементы последовательности, у которых были *одинаковые* ключи на предыдущем этапе сортировки.

Запрос **Sorted** является упрощенным вариантом запроса **OrderBy**, в котором не требуется указывать лямбда-выражение; ключами в этом случае считаются сами элементы последовательности.

Модификации запросов сортировки с суффиксом **ByDescending** выполняются аналогично, но выполняют сортировку по *убыванию* ключа.

Цепочка запросов, обеспечивающая сортировку по набору ключей, должна начинаться с запроса **OrderBy** или **OrderByDescending**, после которого могут следовать запросы **ThenBy** и **ThenByDescending** в любом количестве и любых комбинациях. После запроса **Sorted** или **SortedByDescending** запросы **ThenBy** и **ThenByDescending** указывать нельзя.

Все описанные запросы выполняют *устойчивую сортировку*; это означает, что исходный порядок элементов с *одинаковыми* ключами после сортировки не изменится.

Запрос **Reverse** возвращает последовательность, в которой элементы исходной последовательности располагаются в обратном порядке. Этот запрос уже рассматривался ранее, в п. 3.7.

**Комбинирование**

Методы sequence of T

**Concat**(a2: sequence of T): sequence of T**Union**(a2: sequence of T): sequence of T**Intersect**(a2: sequence of T): sequence of T**Except**(a2: sequence of T): sequence of T\* **Interleave**(a2: sequence of T): sequence of T\* **Interleave**(a2, a3: sequence of T): sequence of T\* **Interleave**(a2, a3, a4: sequence of T): sequence of T

**Расщепление**

Методы sequence of T

\* **Partition**(pred: (T[, integer]) -> boolean):

(sequence of T, sequence of T)

\* **SplitAt**(count: integer): (sequence of T, sequence of T)

Запрос **Concat** возвращает последовательность, содержащую все элементы *первой* последовательности (вызвавшей данный метод), после которых следуют все элементы *второй* последовательности (параметра *a2*).

Запросы **Union**, **Intersect** и **Except** реализуют *теоретико-множественные операции* (соответственно *объединение*, *пересечение* и *разность*) для двух исходных последовательностей (первой, вызвавшей запрос, и второй — параметра *a2*). Последовательность, полученная в результате выполнения любого из этих запросов, не содержит повторяющихся элементов. Порядок следования элементов определяется порядком их *первых* вхождений в *первую* исходную последовательность (в случае операции объединения те элементы второй последовательности, которые отсутствуют в первой, располагаются *после* элементов первой последовательности).

Варианты запроса **Interleave** возвращают последовательность, в которой чередуются элементы исходных последовательностей с одинаковыми индексами (можно объединять 2, 3 или 4 последовательности). Например, запрос *a.Interleave(b,c)* вернет последовательность, в которой элементы исходных последовательностей располагаются следующим образом: *a<sub>0</sub>, b<sub>0</sub>, c<sub>0</sub>, a<sub>1</sub>, b<sub>1</sub>, c<sub>1</sub>, a<sub>2</sub>, b<sub>2</sub>, c<sub>2</sub>* и т. д. Заполнение выходной последовательности закончится, как только будет достигнут конец *самой короткой* из исходных последовательностей. Если, например, последовательность *a* содержит 7 элементов, последовательность *b* — 5 элементов, а последовательность *c* — 10 элементов, то в полученной последовательности будет 15 элементов (по 5 начальных элементов из каждой исходной последовательности в указанном выше порядке).

Запрос **Partition** «расщепляет» исходную последовательность на две части, возвращая кортеж из двух последовательностей (по поводу кортежей см. п. 1.4). В первую последовательность помещаются элементы, удовлетворяющие предикату *pred*; остальные элементы помещаются во вторую последовательность. В каждой полученной последовательности элементы следуют в том же порядке, в котором они располагались в исходной последовательности. Одна из полученных последовательностей может оказаться пустой. В предикате *pred* можно использовать дополнительный параметр — *индекс* обрабатываемого элемента.

Запрос **SplitAt** также расщепляет исходную последовательность на две части. В данном случае в первую часть включаются *count начальных* элементов исходной последовательности, а во вторую — остальные ее элементы. Если *count = 0*, то первая полученная последовательность будет пу-

стой, если значение `count` больше или равно размеру исходной последовательности, то пустой будет вторая полученная последовательность.

К запросам, выполняющим расщепление, можно также отнести специализированный запрос `UnzipTuple`, который будет описан далее, в п. 4.4, совместно с парным к нему запросом объединения `ZipTuple`. Особенность запроса `UnzipTuple` состоит в том, что возвращаемый им кортеж содержит последовательности, тип элементов которых *отличается* от типа элементов исходной последовательности.

Проиллюстрируем некоторые из описанных в этом пункте запросов примерами задач из группы `LinqBegin`.

В задаче **LinqBegin17** требуется извлечь из исходной целочисленной последовательности все нечетные числа, сохранив их исходный порядок и удалив все вхождения повторяющихся элементов, кроме первых:

```
Task('LinqBegin17');  
ReadSeqInteger.Where(e -> Odd(e)).Distinct.WriteAll;
```

Обратите внимание на то, что в задачах группы `LinqBegin`, в которых требуется вывести полученную последовательность, необходимо выводить не только элементы, но и размер последовательности (размер указывается первым). Напомним, что для организации такого вывода предусмотрен специальный запрос `WriteAll` и его синоним `PrintAll`, описанные в модуле `PT4`.

В ситуации, когда при решении задачи к исходной последовательности применяются несколько запросов, часто бывает желательно проверить, к какому результату приводит применение того или иного запроса. Для этого удобно использовать специальный вариант процедуры отладочной печати `Show`, реализованный в виде запроса (об отладочных средствах, включенных в задачник `Programming Taskbook`, см. п. 2.4). Например, можно дополнить цепочку запросов, использованных при решении задачи `LinqBegin17`, следующим образом (полужирным шрифтом выделены добавленные запросы, обеспечивающие отладочную печать):

```
ReadSeqInteger.Where(e -> Odd(e)).Show.Distinct.Show.WriteAll;
```

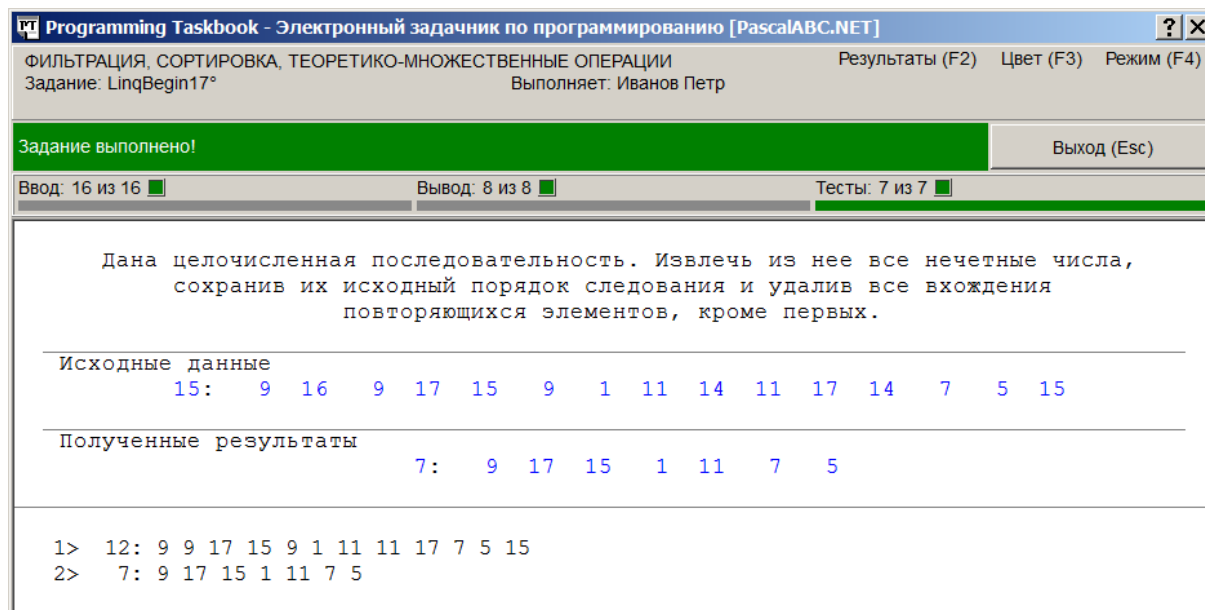


Рис. 9. Окно задачника с отладочным выводом для задачи LinqBegin17

При запуске нового варианта программы в окне задачника появится *раздел отладки*, в котором будут выведены размер и элементы последовательностей, полученных после применения каждого из использованных запросов (см. рис. 9). Легко убедиться, что после первого запроса в последовательности остаются только нечетные элементы, а после второго из нее удаляются все повторения.

В задаче **LinqBegin25** требуется найти сумму всех положительных элементов целочисленной последовательности с порядковыми номерами от  $k_1$  до  $k_2$  включительно (по условию  $1 \leq k_1 < k_2 \leq n$ , где  $n$  — размер последовательности; вначале необходимо ввести числа  $k_1$  и  $k_2$ , затем — исходную последовательность):

```
Task('LinqBegin25');
var (k1,k2) := ReadInteger2;
Write(ReadSeqInteger.Skip(k1 - 1).Take(k2 - k1 + 1)
    .Where(e -> e > 0).Sum);
```

Для отбора «внутренней» части элементов последовательности надо последовательно вызвать запросы `Skip` и `Take` (учитывая при этом, что, в отличие от индексов, порядковые номера начинаются с 1). Просуммировать в полученной последовательности положительные числа можно двумя способами: либо используя запросы `Where` и `Sum` (как в приведенном решении), либо используя единственный запрос `Sum` с параметром-селектором следующего вида: `e -> e > 0 ? e : 0` (этот селектор сохраняет положительные элементы, а отрицательные заменяет на 0, чтобы они не вносили вклад в вычисляемую сумму).

В задаче **LinqBegin29** даются целые числа  $d$  и  $k$  ( $k > 0$ ) и целочисленная последовательность; требуется найти теоретико-множественное объ-

единение двух фрагментов исходной последовательности: первый содержит все элементы до первого элемента, большего  $d$  (не включая его), а второй — все элементы, начиная с элемента с порядковым номером  $k$ . Полученную последовательность (не содержащую одинаковых элементов) требуется дополнительно отсортировать по убыванию:

```
Task('LinqBegin29');
var (d,k) := ReadInteger2;
var a := ReadSeqInteger;
a.TakeWhile(e -> e <= d)
  .Union(a.Skip(k - 1)).SortedDescending.WriteAll;
```

В данном случае требуется получить две части исходной последовательности, после чего объединить их с помощью запроса `Union`. Поэтому необходимо связать последовательность с некоторой переменной и затем дважды обратиться к этой переменной, первый раз применив запрос `TakeWhile`, а второй раз — запрос `Skip`. В конце надо выполнить запрос `SortedDescending`, сортирующий полученную последовательность по убыванию.

Интересно проанализировать процесс формирования итоговой последовательности, добавив в последний оператор несколько запросов отладочной печати `Show`:

```
a.Show('Исходная посл. a: ')
  .TakeWhile(e -> e <= d).Show('b = a.TakeWhile: ')
  .Union(a.Skip(k - 1).Show('c = a.Skip: '))
  .Show('d = b.Union(c): ')
  .SortedDescending.Show('d.SortedDescending:').WriteAll;
```

Для большей наглядности отладочную печать каждой последовательности можно снабдить комментарием. На рис. 10 приводится окно задачника после запуска варианта решения с отладочной печатью.

```
Исходная посл. a: 13: 1 3 3 10 10 8 7 11 2 11 17 3 3
b = a.TakeWhile: 3: 1 3 3
c = a.Skip: 6: 11 2 11 17 3 3
d = b.Union(c): 5: 1 3 11 2 17
d.SortedDescending: 5: 17 11 3 2 1
```



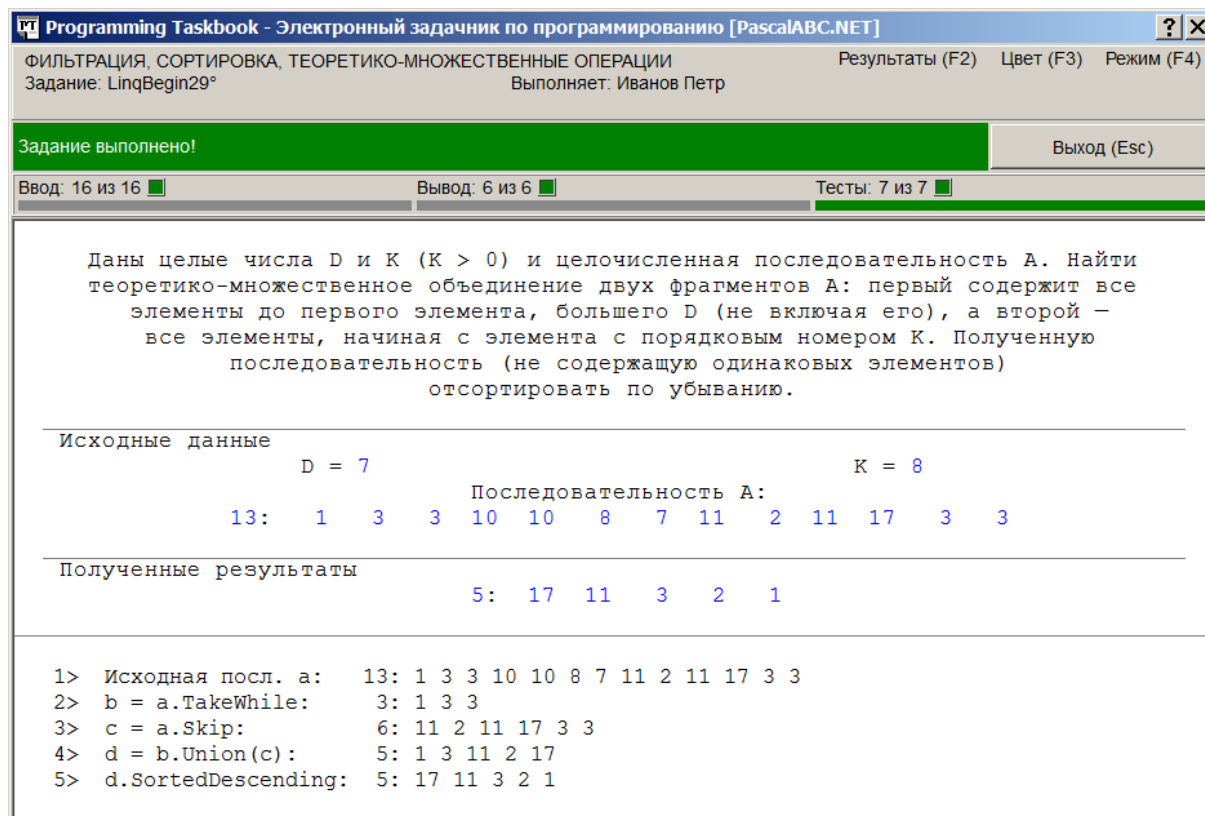


Рис. 10. Окно задачника с отладочным выводом для задачи LinqBegin29

### 4.3. Проецирование

Запросы проецирования формируют на основе исходной последовательности новую последовательность с элементами *другого типа*. Формально к подобным запросам можно отнести и *группирующие запросы*, однако они имеют ряд особенностей и поэтому обычно рассматриваются отдельно (см. п. 4.5). В стандартной библиотеке .NET имеются два запроса проецирования: уже известный нам запрос `Select` (см. п. 3.4) и запрос `SelectMany`. В библиотеку PascalABC.NET добавлены новые запросы проецирования, упрощающие создание последовательностей кортежей специального вида.

#### Проецирование

Методы `sequence of T`

**Select**(sel: (T[, integer]) -> TRes): sequence of TRes

**SelectMany**(sel: (T[, integer]) -> sequence of TRes):  
sequence of TRes

**SelectMany**(sel: (T[, integer]) -> sequence of TMid;  
finalSel: (T, TMid) -> TRes): sequence of TRes

\* **Numerate**([from: integer]): sequence of (integer, T)

\* **Tabulate**(f: T -> TRes): sequence of (T, TRes)

Запрос `Select` преобразует каждый элемент исходной последовательности, используя параметр-селектор `sel` (лямбда-выражение), и заносит пре-

образованное значение в результирующую последовательность. Таким образом, последовательность, полученная в результате применения запроса `Select`, всегда имеет тот же размер, что и исходная последовательность.

Запрос `SelectMany` выполняется более сложным образом. Его параметр-селектор `sel` преобразует каждый элемент исходной последовательности в *последовательность* новых значений. Если в запросе указан только один параметр (`sel`), то все значения из полученных последовательностей объединяются и заносятся в результирующую последовательность. Если в запросе указан дополнительный параметр — *финальный селектор* `finalSel`, то он применяется к паре ( $e_1, e_2$ ), где в качестве  $e_1$  берется очередной элемент из исходной последовательности, а в качестве  $e_2$  — *каждый* из элементов последовательности, полученной путем применения к  $e_1$  селектора `sel` (финальный селектор будет вызываться для каждого элемента  $e_1$  столько раз, сколько элементов входит в последовательность `sel(e1)`). Значения, возвращенные финальным селектором `finalSel`, заносятся в результирующую последовательность.

Таким образом, в любом варианте запроса `SelectMany` по элементам исходной последовательности вначале формируется *последовательность последовательностей* (такая последовательность называется *иерархической*), после чего эти иерархическая последовательность превращается в «плоскую», состоящую из элементов тех последовательностей, которые входили в иерархическую последовательность.

В отличие от запроса `Select`, который всегда возвращает последовательность того же размера, что и исходная, запрос `SelectMany` может вернуть последовательность как большего, так и меньшего размера. Размер может уменьшиться, если какие-либо элементы исходной последовательности будут преобразованы селектором `sel` в *пустые* последовательности.

Приведем пример. Если сформировать последовательность нескольких двузначных целых чисел и «расщепить» каждое число на цифры с помощью соответствующего лямбда-выражения (возвращающего последовательность из двух цифр), то, указав это лямбда-выражение в запросе `Select`, мы получим *иерархическую* последовательность:

```
var a := Range(12,15).Select(e -> Seq(e div 10, e mod 10));
Write(a); // [[1,2],[1,3],[1,4],[1,5]]
```

При выводе полученной последовательности процедурой `Write` легко обнаружить «иерархическую природу» полученной последовательности, поскольку выведенный текст содержит вложенные квадратные скобки, означающие, что каждый элемент полученной последовательности также является, в свою очередь, последовательностью (из двух элементов).

Если же заменить в приведенном фрагменте запрос `Select` на запрос `SelectMany`, то элементы полученных в лямбда-выражении последовательностей запишутся в итоговую последовательность в «плоском» виде:

```
var a := Range(12,15).SelectMany(e -> Seq(e div 10, e mod 10));  
Write(a); // [1,2,1,3,1,4,1,5]
```

Имеет смысл проиллюстрировать и применение второго, более сложного варианта запроса `SelectMany`, использующего два селектора. Изменим предыдущий пример так, чтобы в формируемой последовательности сохранять не отдельную цифру каждого исходного числа, а само исходное число, к которому *справа* приписана одна из его цифр (например, по числу 12 мы хотим получить два числа: 121 и 122). С применением второго варианта запроса `SelectMany` это можно реализовать следующим образом:

```
var a := Range(12,15).SelectMany(e -> Seq(e div 10, e mod 10),  
    (e, d) -> e * 10 + d);  
Write(a); // [121,122,131,133,141,144,151,155]
```

Чтобы подчеркнуть, что в обоих селекторах в качестве первого параметра используется элемент исходной последовательности, мы выбрали для этих параметров одно и то же имя: `e`. Второй параметр второго селектора является одной из тех цифр, которая была получена при «расщеплении» числа `e`; для этого параметра мы выбрали имя `d` (от англ. *digit* — «цифра»).

Имеются модификации запросов `Select` и `SelectMany`, в которых в качестве дополнительного параметра селектора `sel` можно использовать *индекс* обрабатываемого элемента исходной последовательности. Напомним, что аналогичная возможность реализована и для ранее рассмотренных запросов `Where`, `TakeWhile`, `SkipWhile` и `Partition` (см. п. 4.2).

Запрос `Numerate(from)` позволяет снабдить элементы исходной последовательности нумерацией, начиная от номера `from` (при отсутствии параметра `from` он считается равным 1). Полученная последовательность является последовательностью кортежей:

```
var a := Range(12,15).Numerate;  
Write(a); // [(1,12),(2,13),(3,14),(4,15)]
```

Запрос `Tabulate` упрощает формирование данных для табуляции функций (которая ранее обсуждалась нами в пунктах 1.3 и 3.5). С его помощью можно сформировать последовательность пар вида  $(x, f(x))$ , где аргументы  $x$  берутся из исходной последовательности, а функция  $f$  является параметром запроса:

```
var a := Range(1,4).Tabulate(x -> Sqrt(x));  
Write(a);  
// [(1,1),(2,1.4142135623731),(3,1.73205080756888),(4,2)]
```

Как обычно, в конце пункта приведем решения нескольких задач группы LinqBegin, иллюстрирующие применение изученных запросов.

В задаче **LinqBegin33** требуется выделить из исходной целочисленной последовательности положительные числа, извлечь из них последние цифры и удалить в полученной последовательности цифр все вхождения одинаковых цифр, кроме первого, после чего вывести оставшиеся цифры в исходном порядке:

```
Task('LinqBegin33');
ReadSeqInteger.Where(e -> e > 0)
    .Select(e -> e mod 10).Distinct.WriteAll;
```

В задаче **LinqBegin38** требуется обработать элементы исходной целочисленной последовательности следующим образом: если порядковый номер элемента делится на 3 (3, 6, ...), то этот элемент *не включается* в новую последовательность; если остаток от деления порядкового номера на 3 равен 1 (1, 4, ...), то в новую последовательность добавляется *удвоенное значение* этого элемента; в противном случае (для элементов с номерами 2, 5, ...) элемент добавляется в новую последовательность без изменений.

При решении этой задачи следует использовать запросы с лямбда-выражением, включающим *индекс* обрабатываемого элемента; кроме того, надо учитывать, что индекс элемента на 1 меньше его порядкового номера. Если переформулировать условие задачи в терминах индексов, то мы получим, что удваивать надо элементы, индекс которых делится на 3 без остатка, а удалять надо элементы, индекс которых дает при делении на 3 остаток 2.

Поскольку при удалении части элементов индексы оставшихся элементов изменятся, следует вначале изменить (удвоить) нужные элементы с помощью запроса **Select**, а затем выполнить удаление лишних элементов:

```
Task('LinqBegin38'); // Вариант 1
ReadSeqInteger.Select((e, i) -> i mod 3 = 0 ? 2 * e : e)
    .Where((e, i) -> i mod 3 <> 2).WriteAll;
```

Если воспользоваться тем фактом, что в запросе **SelectMany** с элементом допустимо связывать *пустую последовательность*, то можно реализовать решение задачи LinqBegin38, не требующее применения запроса **Where**:

```
Task('LinqBegin38'); // Вариант 2
ReadSeqInteger.SelectMany((e, i) -> i mod 3 = 0 ? Seq(2 * e) :
    i mod 3 = 1 ? Seq(e) : SeqFill(0, 0)).WriteAll;
```

Прокомментируем использованное в запросе лямбда-выражение. Если индекс элемента делится на 3, то возвращается одноэлементная последовательность, содержащая удвоенное значение исходного элемента; в противном случае, если индекс элемента при делении на 3 дает остаток 1, то воз-

вращается последовательность с этим элементом; если и это условие не выполняется, то возвращается пустая последовательность (которую мы создаем с помощью вызова генератора `SeqFill` с нулевым первым параметром).

Данный вариант решения проигрывает в наглядности первому варианту (прежде всего, из-за использования вложенных тернарных операций). Кроме того, в нем приходится создавать вспомогательные последовательности. Однако он интересен как еще одна иллюстрация возможностей запроса `SelectMany`.

#### 4.4. Объединение. Запрос `DefaultIfEmpty`

В данном и следующем пунктах рассматриваются наиболее сложные запросы, которые связаны с *объединением* последовательностей и *группировкой* их элементов.

Основными запросами для объединения являются `Join` и `GroupJoin`. Общим у этих запросов является то, что они обеспечивают объединение или группировку по *ключу*, определяемому по элементам исходных последовательностей. Вариантом объединяющего запроса может также считаться запрос `Zip`. Все эти запросы входят в стандартную библиотеку платформы .NET. Библиотека `PascalABC.NET` содержит также несколько дополнительных запросов, которые можно отнести к категории запросов объединения.

Близкими к запросам объединения могут считаться запросы *комбинирования* последовательностей, рассмотренные ранее (см. п. 4.2). Мы разделили запросы комбинирования и объединения по следующему признаку: комбинирующие запросы возвращают последовательность, тип элементов которой *совпадает* с типом элементов исходных последовательностей, тогда как объединяющие запросы формируют последовательность с типом элементов, отличным от типов элементов исходных последовательностей.

При описании запросов объединения будем предполагать, что последовательность, к которой применяется запрос, имеет тип `sequence of T1`.

##### **Объединение**

##### Методы `sequence of T1`

**Join**(a2: `sequence of T2`; keySel1: `T1 -> TKey`; keySel2: `T2 -> TKey`,  
finalSel: `(T1, T2) -> TRes`): `sequence of TRes`

**GroupJoin**(a2: `sequence of T2`;  
keySel1: `T1 -> TKey`; keySel2: `T2 -> TKey`,  
finalSel: `(T1, sequence of T2) -> TRes`): `sequence of TRes`

\* **Cartesian**(a2: `sequence of T2`): `sequence of (T1, T2)`

\* **Cartesian**(a2: `sequence of T2`; func: `(T1, T2) -> TRes`):  
`sequence of TRes`

**Zip**(a2: `sequence of T2`; func: `(T1, T2) -> TRes`): `sequence of TRes`

\* **ZipTuple**(a2: `sequence of T2`): `sequence of (T1, T2)`

- \* ZipTuple(a2: sequence of T2; a3: sequence of T3):  
sequence of (T1, T2, T3)
- \* ZipTuple(a2: sequence of T2; a3: sequence of T3;  
a4: sequence of T4): sequence of (T1, T2, T3, T4)

В запросах Join и GroupJoin используются две последовательности: первая (*внешняя*) вызывает данные запросы, вторая (*внутренняя*) указывается в качестве их первого параметра a2. Типы элементов у внешней и внутренней последовательностей могут быть различными (выше, в заголовках запросов они обозначены через T1 и T2 соответственно).

Запрос Join выполняет *внутреннее объединение* двух последовательностей по ключу. Ключи определяются первыми двумя лямбда-выражениями — *селекторами ключей* keySel1 и keySel2; к каждой паре элементов внешней и внутренней последовательности, имеющих одинаковые ключи, применяется третье лямбда-выражение — *финальный селектор* finalSel, и его результат заносится в выходную последовательность. Объединение называется *внутренним*, потому что учитываются только те элементы внешней последовательности, для которых найден *хотя бы один* элемент внутренней последовательности с таким же ключом.

Метод GroupJoin тоже выполняет объединение двух последовательностей по ключу, однако результирующий элемент определяется по элементу внешней последовательности и *всем* элементам внутренней последовательности с тем же ключом. Такое объединение называется *левым внешним*; в нем *любой* элемент из внешней («левой») последовательности примет участие в формировании выходной последовательности, даже если для него не найдется «парных» элементов из второй последовательности. Обратите внимание на то, что вторым параметром финального селектора finalSel для метода GroupJoin является *последовательность* (sequence of T2): это последовательность тех элементов исходной внутренней последовательности, ключ которых совпадает с ключом первого параметра селектора finalSel (элемента внешней последовательности).

В полученной последовательности порядок элементов определяется порядком элементов внешней последовательности, а для элементов, построенных по одному и тому же элементу внешней последовательности, — порядком элементов внутренней последовательности.

Рассмотрим пример. Пусть исходные последовательности определены следующим образом:

```
var a1 := Seq(10, 21, 33, 84);
var a2 := Seq(40, 51, 52, 53, 60);
```

Выполним внутреннее объединение этих последовательностей, используя в качестве ключа разряд единиц (т. е. цифру, которой оканчивает-

ся число) и возвращая последовательность пар чисел (каждая пара представляется в виде кортежа):

```
var res := a1.Join(a2, e1 -> e1 mod 10, e2 -> e2 mod 10,  
  (e1, e2) -> (e1, e2));  
Write(res); // [(10,40),(10,60),(21,51),(33,53)]
```

Если мы заменим в предыдущем фрагменте имя запроса на `GroupJoin`, не меняя его параметры, то мы тоже получим последовательность пар, но теперь вторым элементом каждой пары будет *последовательность* (всех элементов `a2`, имеющих одинаковый ключ с первым элементом этой же пары):

```
var res := a1.GroupJoin(a2, e1 -> e1 mod 10, e2 -> e2 mod 10,  
  (e1, e2) -> (e1, e2));  
Write(res); // [(10,[40,60]),(21,[51]),(33,[53]),(84,[])]
```

Мы видим, что в данном случае второе поле каждого кортежа является последовательностью. Обратите внимание на то, что с элементом 84 связывается *пустая последовательность*, поскольку в последовательности `a2` отсутствуют числа, оканчивающиеся на 4 (по этой же причине элемент 84 отсутствовал в последовательности, построенной с помощью запроса `Join`).

Итак, запрос `Join` формирует «плоское» внутреннее объединение (с каждым элементом внешней последовательности в полученном объединении связывается по *одному* элементу внутренней), тогда как внешнее объединение, формируемое запросом `GroupJoin`, является иерархическим (с каждым элементом внешней последовательности связывается *последовательность* элементов внутренней). В некоторых ситуациях требуется сформировать *плоское левое внешнее объединение*, которое отличается от внутреннего объединения только тем, что включает *все* элементы внешней последовательности — в том числе и те, для которых не нашлось пары во внутренней последовательности (подобные элементы в плоском левом внешнем объединении объединяются в пару с каким-либо особым значением, например нулем).

Поскольку для преобразования иерархической последовательности в плоскую предназначен запрос `SelectMany`, естественно использовать для построения плоского левого внешнего объединения комбинацию запросов `GroupJoin` и `SelectMany`:

```
var res := a1.GroupJoin(a2, e1 -> e1 mod 10, e2 -> e2 mod 10,  
  (e1, e2) -> e2.Select(e -> (e1, e))).SelectMany(e -> e);
```

Здесь в последнем лямбда-выражении запроса `GroupJoin` мы преобразуем последовательность `e2` (напомним, что в нее входят все элементы внутренней последовательности, имеющие одинаковый ключ с элементом `e1`) в последовательность пар `(e1, e)`, где `e` пробегает все элементы последовательности `e2`, после чего превращаем набор преобразованных последова-

тельностью (т. е. *иерархическую последовательность*) в плоскую последовательность, используя запрос `SelectMany(e -> e)`.

Однако в результате мы получим обычное внутреннее объединение:

```
Write(res); // [(10,40),(10,60),(21,51),(33,53)]
```

Это вполне естественно, поскольку мы не предусмотрели обработку особой ситуации: когда с элементом `e1` внешней последовательности нельзя связать ни одного элемента внутренней последовательности и, таким образом, в последнем лямбда-выражении запроса `GroupJoin` последовательность `e2` оказывается пустой. Как мы знаем, запрос `SelectMany` просто игнорирует пустые последовательности.

Для правильной обработки этой особой ситуации нам было бы достаточно «превратить» пустую последовательность `e2` в одноэлементную последовательность, содержащую то особое значение, которое в результирующем плоском левом внешнем объединении мы хотим связать с «одиночками» элементами внешней последовательности (в нашем случае таким одиночным элементом является число 84).

В стандартной библиотеке платформы .NET предусмотрен запрос, `DefaultIfEmpty` специально предназначенный для решения подобных задач.

#### **Обработка пустой последовательности**

Метод `sequence of T`

`DefaultIfEmpty(defaultValue: T): sequence of T`

Запрос `DefaultIfEmpty` может применяться к последовательности с элементами любого типа `T` и имеет один необязательный параметр `defaultValue` того же типа `T`. Если исходная последовательность является непустой, то он возвращает эту последовательность в неизменном виде; в противном случае он возвращает последовательность, содержащую единственный элемент со значением `defaultValue` (или с нулевым значением соответствующего типа, если параметр `defaultValue` не указан). Таким образом, в результате применения запроса `DefaultIfEmpty` мы *всегда* будем получать непустую последовательность.

Чтобы предыдущий фрагмент правильно конструировал плоское левое внешнее объединение, достаточно добавить в него запрос `DefaultIfEmpty`:

```
var res := a1.GroupJoin(a2, e1 -> e1 mod 10, e2 -> e2 mod 10,
    (e1, e2) -> e2.DefaultIfEmpty.Select(e -> (e1, e)))
    .SelectMany(e -> e);
Write(res); // [(10,40),(10,60),(21,51),(33,53),(84,0)]
```

Теперь в полученной последовательности будет присутствовать пара с элементом 84, который будет связан с особым значением 0.

Завершая обзор запросов `Join` и `GroupJoin`, заметим, что для них используется эффективная реализация, не сводящаяся к двойному циклу с попарными проверками ключей: внутренняя последовательность предварительно преобразуется к специальной структуре данных — *таблице просмотра*,



индексированной по ключу, что позволяет находить ее элементы, парные к элементам внешней последовательности, не прибегая к их многократному перебору (по поводу таблицы просмотра см. также п. 4.6).

Иногда требуется построить последовательность, состоящую из всевозможных пар, в которых первый элемент берется из первой последовательности, а второй элемент — из второй. Такой набор пар называется *декартовым произведением* исходных последовательностей. Его можно сформировать с помощью запроса `Join`, если задать один и тот же ключ для *всех* элементов обеих последовательностей (например, число 0). Однако такой способ формирования декартова произведения будет неэффективным, поскольку потребует выполнения множества дополнительных действий (в частности, создания таблицы просмотра). Для эффективного формирования декартова произведения можно использовать комбинацию запросов `SelectMany` и `Select`:

```
var a1 := Seq(1, 2, 3);  
var a2 := Seq(10, 20);  
var res := a1.SelectMany(e1 -> a2.Select(e2 -> (e1, e2)));  
Write(res); // [(1,10),(1,20),(2,10),(2,20),(3,10),(3,20)]
```

Чтобы не конструировать каждый раз подобную комбинацию запросов, в библиотеку `PascalABC.NET` включен запрос `Cartesian`, специально предназначенный для формирования декартова произведения, и содержащий единственный параметр — имя второй последовательности:

```
var res2 := a1.Cartesian(a2);
```

Последовательность `res2`, полученная в результате выполнения этого запроса, будет совпадать с последовательностью `res` из предыдущего фрагмента. Запрос `Cartesian` может содержать дополнительный параметр — лямбда-выражение, определяющее преобразование, которое будет применено к каждой паре элементов декартова произведения перед помещением результата в итоговую последовательность.

Оставшиеся запросы объединения имеют в своем названии слово `Zip`; все эти запросы объединяют (тем или иным способом) элементы исходных последовательностей *с одинаковыми индексами*. Если таким образом объединяются последовательности разного размера, то полученная последовательность будет иметь размер, равный *наименьшему* размеру исходных последовательностей («лишние» элементы более длинных последовательностей не обрабатываются).

В запросе `Zip` из стандартной библиотеки платформы `.NET` способ объединения элементов задается в виде лямбда-выражения, например:

```
var a1 := Seq(3, 4, 5, 6);  
var a2 := Seq(50, 60, 70);  
var res := a1.Zip(a2, (e1, e2) -> e1 * e2);
```

```
Write(res); // [150,240,350]
```

С помощью этого же запроса можно получить последовательность пар (кортежей) соответствующих элементов:

```
var res2 := a1.Zip(a2, (e1, e2) -> (e1, e2));
Write(res2); // [(3,50),(4,60),(5,70)]
```

В PascalABC.NET последнюю задачу можно решить проще, используя специальный запрос `ZipTuple` с единственным параметром — второй последовательностью: `a1.ZipTuple(a2)`.

Запрос `ZipTuple` имеет перегруженные варианты, позволяющие объединять соответствующие элементы трех и четырех последовательностей:

```
var a1 := Seq(3, 4, 5, 6);
var a2 := Seq(50, 60, 70);
var a3 := Seq('a', 'b', 'c');
var a4 := Seq(1.1, 2.2, 3.3);
var res := a1.ZipTuple(a2, a3, a4);
Write(res); // [(3,50,a,1.1),(4,60,b,2.2),(5,70,c,3.3)]
```

Имеется запрос `UnzipTuple` без параметров, выполняющий обратное действие, т. е. *расщепляющий* исходную последовательность кортежей (пар, троек или четверок) на соответствующее количество результирующих последовательностей (которые возвращаются в виде *кортежа последовательностей*). Например, имея последовательность `res` из предыдущего примера, можно «восстановить» последовательности `a1`, `a2`, `a3`, `a4` следующим образом (используя кортежное присваивание):

```
(a1, a2, a3, a4) := res.UnzipTuple;
```

Запрос `UnzipTuple` можно отнести к категории специализированных запросов расщепления, рассмотренных в п. 4.2.

Обратимся к задачам из группы `LinqBegin`.

В задаче **LinqBegin51** даются последовательности `a` и `b` положительных целых чисел, причем все числа в последовательности `a` различны. Требуется получить последовательность строк вида «`s:e`», где `s` обозначает сумму тех чисел из `b`, которые оканчиваются на ту же цифру, что и число `e` — один из элементов последовательности `a` (например, «74:23»); если для числа `e` не найдено ни одного подходящего числа из последовательности `b`, то в качестве `s` надо указать 0. Элементы полученной последовательности должны располагаться по возрастанию значений найденных сумм, а при равных суммах — по убыванию значений элементов `a`.

Поскольку в полученную последовательность надо включать также и элементы исходной последовательности `a`, для которых в последовательности `b` отсутствуют соответствующие элементы (оканчивающиеся на ту же цифру), нам необходимо построить *левое внешнее объединение*, исполь-

зую запрос `GroupJoin`. Задача немного упрощается благодаря тому, что пустые последовательности, полученные в результате объединения, не потребуются обрабатывать особым образом: при применении к таким последовательностям запроса `Sum` мы получим нужное по условию значение 0.

В задаче требуется выполнить сортировку полученных данных по *набору ключей*, поэтому результат объединения удобно представить в виде кортежа, из которого будет легко извлекать ключи сортировки. Отсортированную последовательность кортежей останется преобразовать в последовательность строк с помощью запроса `Select`:

```
Task('LinqBegin51');
var a := ReadSeqInteger;
a.GroupJoin(ReadSeqInteger, e1 -> e1 mod 10, e2 -> e2 mod 10,
    (e1, e2) -> (e1, e2.Sum))
    .OrderBy(e -> e[1]).ThenByDescending(e -> e[0])
    .Select(e -> e[1] + ':' + e[0]).WriteAll;
```

Обратите внимание на способ получения результирующей строки в запросе `Select`: достаточно применить операцию `+` к *числовым* полям кортежа и разделителю «:» (двоеточие), при этом числовые поля будут автоматически преобразованы к своим строковым представлениям. Здесь используется следующее правило `PascalABC.NET` для операции `+`: *если один ее операнд является строкой (или символом), то другой операнд преобразуется в свое строковое представление и операция `+` выполняет сцепление полученных строк.*

В задаче **LinqBegin53** также даны две целочисленные последовательности. Требуется получить последовательность *различных* сумм, в которых первое слагаемое берется из первой последовательности, а второе — из второй. Полученную последовательность надо упорядочить по возрастанию:

```
Task('LinqBegin53');
var a := ReadSeqInteger;
a.Cartesian(ReadSeqInteger, (e1, e2) -> e1 + e2)
    .Distinct.Order.WriteAll;
```

В этой задаче надо вначале построить суммы всевозможных комбинаций пар элементов исходных последовательностей, затем отбросить повторяющиеся значения, после чего отсортировать оставшиеся (различные) значения по возрастанию. Для перебора всех комбинаций пар элементов проще всего использовать запрос `Cartesian`.

## 4.5. Группировка

В результате *группировки* исходная последовательность преобразуется в последовательность наборов (*групп*) элементов, обладающих некоторым

общим свойством. Полученные группы элементов обычно являются последовательностями, поэтому можно сказать, что группировка преобразует исходную «плоскую» последовательность в *иерархическую* (в этом отношении ее действие является обратным к действию запроса `SelectMany`).

В стандартной библиотеке платформы .NET имеется один группирующий запрос `GroupBy`, реализованный в четырех вариантах. В библиотеку `PascalABC.NET` добавлено еще несколько специализированных запросов, которые можно отнести к категории группирующих.

### Группировка

Методы `sequence of T`

**GroupBy**(keySel: T -> TKey): `sequence of IGrouping<TKey, T>`

`GroupBy`(keySel: T -> TKey; elemSel: T -> TElem)

: `sequence of IGrouping<TKey, TElem>`

`GroupBy`(keySel: T -> TKey;

finalSel: (TKey, `sequence of T`) -> TRes): `sequence of TRes`

`GroupBy`(keySel: T -> TKey; elemSel: T -> TElem;

finalSel: (TKey, `sequence of TElem`) -> TRes): `sequence of TRes`

\* **Batch**(size: integer): `sequence of sequence of T`

\* `Batch`(size: integer; func: `sequence of T -> TRes`):

`sequence of TRes`

\* **Pairwise**: `sequence of (T,T)`

\* `Pairwise`(func: (T,T) -> TRes): `sequence of TRes`

Первый параметр любого варианта запроса `GroupBy` представляет собой лямбда-выражение `keySel` (*селектор ключа*), определяющее *ключ группировки*. Если этот параметр является единственным, то результатом выполнения метода `GroupBy` является последовательность с элементами специального обобщенного типа `IGrouping<TKey, T>`. Данный тип представляет собой последовательность элементов типа `T` и, кроме того, имеет дополнительное свойство `Key` типа `TKey`, которое можно рассматривать как некоторую общую характеристику (*ключ*) всех элементов, входящих в эту последовательность. Таким образом, в результате выполнения запроса `GroupBy` с одним параметром мы получим последовательность, каждый элемент которой, в свою очередь, тоже является последовательностью, а именно — *группой тех элементов исходной последовательности, которые имеют одинаковый ключ*. При этом значение ключа любой группы элементов можно получить, обратившись к свойству `Key` данной группы.

Приведем пример:

```
var a := Seq(10, 29, 33, 84, 40, 59, 52, 53, 60);
```

```
var res := a.GroupBy(e -> e mod 10);
```

```
Writeln(res); // [[10,40,60],[29,59],[33,53],[84],[52]]
```

Мы сгруппировали исходную последовательность чисел по ключу — последней цифре числа. Таким образом, в каждую группу попали все эле-

менты с одинаковой последней цифрой. При выводе сгруппированной последовательности процедурой `Writeln` она представляется в виде обычной иерархической последовательности (в нашем случае — последовательности групп чисел). Порядок следования групп (и элементов в пределах каждой группы) определяется однозначно: в пределах каждой группы все элементы сохраняют свой исходный порядок, а группы располагаются в том порядке, в котором появляются в исходной последовательности *первые* элементы с данными ключами.

Приведенный фрагмент никак не отражает тот факт, что результат запроса (переменная `res`) является особым типом `IGrouping`. Чтобы продемонстрировать это, дополним фрагмент следующим циклом:

```
foreach var e in res do
    Print(e.Key); // 0 9 3 4 2
```

В результате выполнения этого цикла на экране напечатаются значения ключей, соответствующих группам последовательности `res`. Именно в наличии дополнительного свойства `Key` у элементов и состоит единственное отличие типа `IGrouping` от обычной иерархической последовательности.

Разобравшись с простейшим вариантом запроса `GroupBy`, перейдем к его более сложным вариантам.

Группировка может сопровождаться *проецированием*; в результате полученная последовательность будет содержать элементы типа `IGrouping<TKey, TRes>`, где `TRes` — тип элементов, полученных из элементов исходной последовательности путем некоторого преобразования. Требуемое преобразование задается в качестве второго параметра метода `GroupBy` и представляет собой лямбда-выражение `elemSel` (*селектор элемента*), принимающее элемент исходной последовательности и возвращающее преобразованный элемент (типа `TRes`). Таким образом, в данном случае группы будут состоять из элементов типа `TRes`.

Модифицируем предыдущий пример таким образом, чтобы в группы включались преобразованные числа из исходной последовательности, в которых отброшена правая цифра:

```
var a := Seq(10, 29, 33, 84, 40, 59, 52, 53, 60);
var res := a.GroupBy(e -> e mod 10, e -> e div 10);
Writeln(res); // [[1,4,6],[2,5],[3,5],[8],[5]]
foreach var e in res do
    Print(e.Key); // 0 9 3 4 2
```

В этом примере свойство `Key` оказывается особенно полезным, так как по значениям самих элементов в полученных группах невозможно восстановить значение их общего ключа.

Третий и четвертый варианты запроса `GroupBy` являются наиболее гибкими, поскольку предоставляют программисту возможность самостоятель-

но определить, в каком виде будут храниться сгруппированные элементы в результирующей последовательности. Для этого используется параметр `finalSel` (*финальный селектор*) — лямбда-выражение, принимающее ключ и связанную с ним группу (последовательность) и возвращающее новое представление данной группы. В третьем варианте запроса финальный селектор принимает последовательность исходных элементов (типа `T`); в четвертом варианте дополнительно выполняется проецирование: исходные элементы преобразуются к типу `TElem` с помощью селектора элемента `elemSel`, после чего полученные последовательности сгруппированных элементов (уже имеющих тип `TElem`) обрабатываются в финальном селекторе.

Обычно варианты запроса `GroupBy` с финальным селектором используются в ситуациях, когда в результате группировки требуется получить не сами группы элементов, а некоторые их характеристики.

Например, следующий фрагмент позволяет получить для каждой группы кортеж из двух значений: ключа группы и суммы всех входящих в нее элементов:

```
var a := Seq(10, 29, 33, 84, 40, 59, 52, 53, 60);
var res := a.GroupBy(e -> e mod 10, (k, ee) -> (k, ee.Sum));
Writeln(res); // [(0,110),(9,88),(3,86),(4,84),(2,52)]
```

Обратите внимание на имя второго параметра в финальном селекторе: `ee`. Подобные «удвоенные» идентификаторы часто используются для обозначения последовательностей, являющихся параметрами лямбда-выражений.

Предположим теперь, что нам требуется получить суммы не исходных чисел, а чисел, у которых отброшена правая цифра. Для решения такой задачи можно использовать предыдущий фрагмент, добавив в запрос `Sum` параметр — лямбда-выражение вида `e -> e div 10`. Другим способом решения будет использование четвертого варианта запроса `GroupBy` с указанным лямбда-выражением в качестве второго параметра (селектора элемента):

```
var a := Seq(10, 29, 33, 84, 40, 59, 52, 53, 60);
var g := a.GroupBy(e -> e mod 10, e -> e div 10,
(k, ee) -> (k, ee.Sum));
Writeln(g); // [(0,11),(9,7),(3,8),(4,8),(2,5)]
```

В приведенном фрагменте добавленное лямбда-выражение выделено полужирным шрифтом.

Нам осталось описать специализированные запросы группировки, добавленные в библиотеку `PascalABC.NET`. По сравнению с описанием запроса `GroupBy` это более простая задача.

Запрос `Batch` разбивает исходную последовательность на группы одинакового размера (определяемого параметром `size`); при этом последняя группа может иметь меньший размер. Запрос `Batch` может иметь второй па-

раметр — лямбда-выражение, которое применяет к каждой группе некоторое преобразование перед помещением результата в итоговую последовательность. Если параметр не указан, то в итоговую последовательность помещаются сами группы в виде последовательностей:

```
var a := Seq(10, 29, 33, 84, 40, 59, 52, 53, 60);
Writeln(a.Batch(2)); // [[10,29],[33,84],[40,59],[52,53],[60]]
Writeln(a.Batch(2, ee -> ee.Sum)); // [39,117,99,105,60]
Writeln(a.Batch(3)); // [[10,29,33],[84,40,59],[52,53,60]]
Writeln(a.Batch(4)); // [[10,29,33,84],[40,59,52,53],[60]]
```

Запрос `Pairwise` разбивает исходную последовательность на пары соседних элементов, причем пары являются *перекрывающимися* — например, второй элемент последовательности входит в две пары: с первым и с третьим элементом. Запрос `Pairwise` может иметь параметр — лямбда-выражение, которое определяет, каким образом надо преобразовать пару соседних элементов перед помещением результата в итоговую последовательность. Если параметр не указан, то в итоговую последовательность помещаются пары соседних элементов в виде кортежей:

```
var a := Seq(10, 29, 33, 84, 40, 59, 52, 53);
Writeln(a.Pairwise);
// [(10,29),(29,33),(33,84),(84,40),(40,59),(59,52),(52,53)]
Writeln(a.Pairwise((e1, e2) -> e1 + e2));
// [39,62,117,124,99,111,105]
```

Заметим, что для получения *неперекрывающихся* пар элементов (в предположении, что последовательность имеет четный размер) достаточно использовать комбинацию запросов `Pairwise` и `Slice` (см. п. 3.7):

```
var a := Seq(10, 29, 33, 84, 40, 59, 52, 53);
Writeln(a.Pairwise.Slice(0, 2));
// [(10,29),(33,84),(40,59),(52,53)]
```

Завершая рассмотрение запросов группировки, рассмотрим задачу **LinqBegin57**. В ней дается целочисленная последовательность и требуется выбрать максимальный элемент среди всех ее элементов, оканчивающихся одной и той же цифрой. Полученную последовательность максимальных элементов надо упорядочить по возрастанию их последних цифр:

```
Task('LinqBegin57'); // Вариант 1
ReadSeqInteger.GroupBy(e -> Abs(e) mod 10, (k, ee) -> ee.Max)
.OrderBy(e -> Abs(e) mod 10).WriteAll;
```

Здесь мы использовали третий вариант запроса `GroupBy`, заменив в финальном селекторе полученные группы элементов на максимальные элементы этих групп. При формировании ключа необходимо использовать

функцию `Abs`, так как среди элементов исходной последовательности могут быть отрицательные.

Чтобы избежать повторного вычисления ключа в запросе сортировки `OrderBy`, можно создать при группировке кортеж из двух полей: ключа и максимального значения, однако в этом случае перед выводом результатов придется дополнительно вызвать запрос `Select`:

```
Task('LinqBegin57'); // Вариант 2
ReadSeqInteger.GroupBy(e -> Abs(e) mod 10, (k, ee) -> (k, ee.Max))
    .OrderBy(e -> e[0]).Select(e -> e[1]).WriteAll;
```

Наконец, приведем решение, в котором используется первый вариант запроса `GroupBy` (совместно с запросом `Select`):

```
Task('LinqBegin57'); // Вариант 3
ReadSeqInteger.GroupBy(e -> Abs(e) mod 10)
    .OrderBy(ee -> ee.Key).Select(ee -> ee.Max).WriteAll;
```

В этом варианте при сортировке указывается свойство `Key` полученных групп элементов (напомним, что группы в данном случае имеют тип `IGrouping`), а максимальный элемент в этих группах определяется в запросе `Select`. Вариант 3 является более наглядным, чем вариант 2, и, в отличие от варианта 1, не требует повторного вычисления ключа в запросе сортировки.

#### 4.6. Запросы экспортирования и вспомогательные запросы

В данном, завершающем пункте главы 4 мы рассмотрим *экспортирующие* запросы, связанные с преобразованием последовательностей в структуры других типов, а также некоторые запросы, для которых сложно подобрать какую-либо определенную категорию.

Ранее мы уже отмечали, что многие структуры данных могут быть преобразованы в последовательности автоматически; такие структуры данных мы называли *коллекциями*. К коллекциям можно применять любые запросы; при этом коллекция преобразуется в последовательность, а к полученной последовательности применяется указанный запрос.

Однако обратное преобразование последовательности в структуру другого типа необходимо выполнять явным образом. Таким преобразованием обеспечивает особая группа *экспортирующих* запросов. Часть *экспортирующих* запросов определена в стандартной библиотеке платформы .NET, а часть реализована в библиотеке `PascalABC.NET`.

**Экспортирование**  
**ToArray**: array of T

Методы `sequence of T`



**ToList:** List<T>

**ToDictionary**(keySel: T -> TKey): Dictionary<TKey, T>

ToDictionary(keySel: T -> TKey; valueSel: T -> TVal):

Dictionary<TKey, TVal>

**ToLookup**(keySel: T -> TKey): ILookup<TKey, T>

ToLookup(keySel: T -> TKey; valueSel: T -> TVal):

ILookup<TKey, TVal>

\* **ToLinkedList:** LinkedList<T>

\* **ToHashSet:** HashSet<T>

\* **ToSortedSet:** SortedSet<T>

С применением перечисленных запросов последовательность `sequence of T` может быть преобразована в следующие структуры данных:

- *динамический массив* `array of T`,
- *список на базе массива* `List<T>`,
- *словарь* `Dictionary<TKey, T>` или `Dictionary<TKey, TVal>`,
- *таблица просмотра* `ILookup<TKey, T>` или `ILookup<TKey, TVal>`,
- *двусвязный список* `LinkedList<T>`,
- *множество на базе хеш-таблицы* `HashSet<T>`,
- *отсортированное множество* `SortedSet<T>`.

С динамическими массивами мы уже знакомы; остальные структуры (кроме таблицы просмотра) рассматриваются в следующем выпуске настоящей серии — см. [1, гл. 2–4]. Таблица просмотра редко создается в программах в явном виде; она используется во внутренней реализации запросов объединения (см. п. 4.4) для повышения их эффективности.

Для большинства экспортирующих запросов параметры не требуются. Исключение составляют запросы для преобразования в словарь и таблицу просмотра. Это объясняется тем, что данные структуры связывают с каждым своим элементом особую характеристику — *ключ*, и поэтому при указанном преобразовании необходимо определить лямбда-выражение `keySel` (*селектор ключа*), позволяющее по элементу исходной последовательности получить связанный с ним ключ (при создании словаря необходимо, чтобы все элементы последовательности имели *уникальные* ключи; при создании таблицы просмотра ключи элементов могут совпадать). При создании словаря или таблицы просмотра можно дополнительно указать лямбда-выражение `valueSel` (*селектор значения*) которое будет преобразовывать элемент исходной последовательности перед его добавлением в словарь или таблицу просмотра в качестве *значения*, т. е. второго элемента пары «ключ–значение».

Вместо любого запроса экспортирования, не имеющего параметров, можно использовать так называемую *короткую функцию*, выполняющую

аналогичное преобразование (эпитет «короткая» объясняется краткостью *имен* этих функций):

```
function Arr(a: sequence of T): array of T
function Lst(a: sequence of T): List<T>
function LLst(a: sequence of T): LinkedList<T>
function HSet(a: sequence of T): HashSet<T>
function SSet(a: sequence of T): SortedSet<T>
```

Удобство коротких функций заключается в том, что в качестве их параметров можно указывать не только последовательность, но и любое количество параметров типа *T*. К этой категории можно отнести и функцию *Seq* для генерации последовательности, которая нами уже неоднократно использовалась. Имеется также короткая функция *Dict* для генерации словаря, но она требует использования параметров специального типа. Для генерации элементов словаря (пар «ключ–значение») предназначена короткая функция *KV*. Обе эти функции описываются в [1, п. 4.2].

Нам осталось рассмотреть немногочисленную группу вспомогательных запросов.

#### **Вспомогательные запросы**

Методы sequence of T

- \* **ForEach**(action: (T[, integer]) -> ())
- \* **Print**([delim: integer]): sequence of T
- \* **Println**([delim: integer]): sequence of T

Метод sequence of string

- \* **WriteLines**(fname: string): sequence of string

Запрос *ForEach* является аналогом цикла *foreach*; он обеспечивает перебор элементов исходной последовательности и выполнение для каждого из элементов *действия* *action*, описанного в виде лямбда-выражения — параметра запроса *ForEach*. Заметим, что это единственный из запросов, который является *процедурой* и требует указания процедуры в качестве параметра.

Использование запроса *ForEach* приводит к несколько более краткому коду, чем использование одноименного цикла, если в цикле требуется выполнить единственный оператор. Вспомним пример из п. 4.5, в котором для вывода ключей последовательности с элементами типа *IGrouping* использовался цикл *foreach*:

```
foreach var e in res do
    Print(e.Key);
```

С применением запроса *ForEach* это же действие можно записать более кратко и тоже достаточно наглядно:

```
res.ForEach(e -> Print(e.Key));
```

Кроме того, применение запроса `ForEach` оправдано, если надо выполнить некоторое простое действие для всех элементов последовательности, созданной «на лету» и поэтому не имеющей имени.

Полезной особенностью запроса `ForEach`, отсутствующей у одноименного цикла, является возможность использования в лямбда-выражении дополнительного параметра — *индекса* обрабатываемого элемента.

Запросы `Print` и `Println` не требуют дополнительных пояснений: они используются в нашей книге, начиная с п. 3.2, для вывода последовательностей на экран. Мы называли их *методами* массивов и последовательностей, чтобы подчеркнуть, что они, в отличие от одноименных процедур, требуют применения точечной нотации, однако фактически они являются запросами последовательностей, что позволяет использовать их для любых коллекций.

Обратите внимание на особенность этих запросов, которая раньше не упоминалась: они не только обеспечивают вывод элементов последовательности на экран, но и возвращают эту последовательность, поэтому после вызова запроса `Print` или `Println` цепочку запросов можно продолжить. Это превращает запросы `Print/Println` в удобное средство отладки (аналогичное отладочному запросу `Show`, реализованному в задачнике *Programming Taskbook*, — см. его описание в п. 4.2).

В качестве примера можно обратиться к фрагменту, демонстрирующему действия для получения непересекающихся пар элементов исходной последовательности (этот пример приводился в п. 4.5 при описании запроса `Pairwise`):

```
var a := Seq(10, 29, 33, 84, 40, 59, 52, 53);  
WriteLn(a.Pairwise.Slice(0, 2));
```

Заменяем процедуру `WriteLn` на несколько запросов `Println`:

```
a.Println.Pairwise.Println.Slice(0, 2).Println;
```

В результате мы получим на экране «снимки» последовательности на каждом этапе ее преобразования:

```
10 29 33 84 40 59 52 53  
(10,29) (29,33) (33,84) (84,40) (40,59) (59,52) (52,53)  
(10,29) (33,84) (40,59) (52,53)
```

Последний из вспомогательных запросов `WriteLines` тоже связан с выводом данных. Он позволяет записать все элементы *строковой* последовательности (*sequence of string*) в текстовый файл; имя файла `fname` указывается в качестве единственного параметра запроса. При использовании этого запроса все действия по созданию файла, записи в него данных и закрытию выполняются автоматически. Запрос возвращает исходную последователь-

ность в неизменном виде, что позволяет использовать его (как и запросы Print/Println) в середине цепочки запросов.

## Глава 5. Дополнительные средства обработки массивов

В главе 3 мы познакомились со средствами языка PascalABC.NET, обеспечивающими генерацию, ввод и вывод динамических массивов. Что касается их обработки и преобразования, то в нашем распоряжении к настоящему времени имеется стандартная операция индексирования (и ее расширение, позволяющее создавать срезы), метод `Length`, позволяющий определить размер массива (и методы `Low` и `High`, определяющие диапазон его индексов), процедура `Reverse`, а также весь арсенал запросов, описанных в главе 4, поскольку массив может рассматриваться как последовательность.

Однако для динамических массивов в PascalABC.NET предусмотрен еще ряд возможностей, реализованных в виде методов, операций и процедур. Большинство из этих возможностей мы рассмотрим в данной главе. Для их иллюстрации мы будем использовать, наряду с фрагментами обычных программ, задачи группы `Array` из электронного задачника `Programming Taskbook` (см. главу 2).

### 5.1. Особенности присваивания статических и динамических массивов. Копирование и объединение динамических массивов

При работе с динамическими массивами необходимо учитывать, что они являются *ссылочными* типами данных, как и большинство типов в PascalABC.NET<sup>5</sup>. Это означает, что переменная, описанная как динамический массив, фактически содержит только *адрес* того участка памяти, который выделен для хранения элементов массива и дополнительной информации о нем (в эту информацию входит, например, *размер* массива). Именно поэтому подобную переменную можно описать еще до того, как память для массива будет выделена:

```
var a: array of integer;
```

---

<sup>5</sup> Среди немногочисленных стандартных типов, не являющихся ссылочными (они называются *размерными*), следует упомянуть все числовые типы, а также `boolean` и `char`.

В результате такого описания переменная *a* будет содержать значение *nil* (пустой адрес), означающее, что с ней еще не связан никакой «реальный» массив. Только инициализация переменной *a* с помощью конструкции `new integer[10]` (или другим способом, например с применением функции-генератора) обеспечивает ее связь с существующим массивом. В противоположность этому описание переменной как статического массива (например, `var s: array[1..10] of integer`) сразу выделяет для этого массива память (размер которой определяется указанным диапазоном индексов) и связывает всю эту память с описываемой переменной.

Отмеченные различия существенным образом влияют на смысл операции присваивания, выполняемой для статических и динамических массивов.

Для статических массивов присваивание вида `a := b` возможно только в случае, если переменные *a* и *b* описаны с применением *общего описателя массива* или если для них использован один и тот же *именованный тип*. Например, можно описать массивы таким образом:

```
var a, b: array[1..10] of integer;
...
b := a; // допустимое присваивание
```

Можно описать массивы и по отдельности, но для их совместимости по присваиванию необходимо в этом случае ввести новый тип (который требуется определить в разделе описаний):

```
type intArray = array[1..10] of integer;
begin
  var a: intArray;
  var b: intArray;
  ...
  b := a; // допустимое присваивание
```

В любом случае в результате присваивания вида `a := b` произойдет *копирование значений* всех элементов массива *a* в элементы массива *b* с теми же индексами.

Если же попытаться откомпилировать следующий фрагмент, то произойдет ошибка компиляции:

```
var a: array[1..10] of integer;
var b: array[1..10] of integer;
b := a; // ошибка компиляции
```

При этом сообщение об ошибке будет иметь следующий «странный» вид: *«Нельзя преобразовать тип array [1..10] of integer к array [1..10] of integer»*. Подобное сообщение объясняется тем, что при каждом появлении в программе нового описания *статического* массива компилятор создает новый «внутренний» *именованный тип*, который связывает с описываемой

переменной. Поскольку в нашем примере *a* и *b* описаны в *разных* операторах описания, для них были сгенерированы разные внутренние типы (условно говоря, `arr1_10_integer1` и `arr1_10_integer2`), которые *не считаются эквивалентными и, в частности, не являются совместимыми по присваиванию*. Поэтому компилятор фактически сообщает, что он не может преобразовать тип `arr1_10_integer1` к `arr1_10_integer2`, но поскольку эти имена являются внутренними и отсутствуют в программе, он заменяет их на текст исходных описаний (чем, возможно, только сбивает с толку неопытного программиста).

Указанная особенность, связанная с эквивалентностью типов, приводит к аналогичным проблемам при использовании статических массивов в качестве параметров подпрограмм (или возвращаемых значений функций). Поскольку мы в дальнейшем не собираемся работать со статическими массивами, мы не будем обсуждать детали этих проблем (см. по этому поводу [2, п. 13.7.1]). Упомянутые особенности и связанные с ними проблемы являются добавочным аргументом в пользу того, чтобы отказаться от использования статических массивов в программах на языке PascalABC.NET.

С динамическими массивами ситуация совершенно иная. Для эквивалентности динамических массивов (и, следовательно, их совместимости по присваиванию) достаточно, чтобы они имели *одинаковый тип элементов*. Где и как они описаны в программе, не имеет никакого значения. Подобная эквивалентность называется *структурной* (в отличие от *именной* эквивалентности статических массивов). Заметим, что структурная эквивалентность в PascalABC.NET реализована также для *множеств set of T* [1, гл. 3] и процедурных типов (см. п. 1.3).

Поскольку размер динамического массива при описании не указывается, он не влияет на совместимость массивов. Но при таком подходе поэлементное копирование при присваивании теряет смысл. И действительно, присваивание динамических массивов означает совсем другое: оно лишь изменяет *ссылку*, хранящуюся в изменяемой переменной-массиве. Приведем пример:

```
var a := Arr(1, 2, 3);  
var b := Arr(6, 7);  
b := a;
```

Поскольку в генераторах `Arr` для переменных *a* и *b* использованы параметры одинакового типа (`integer`), эти переменные имеют одинаковый тип `array of integer` и поэтому совместимы по присваиванию. Подчеркнем, что с ними связываются массивы *разного* размера. Фактически в *a* хранится *ссылка* на массив с элементами 1, 2, 3, а в *b* хранится ссылка на массив с элементами 6 и 7. Присваивание вида `b := a` изменяет лишь ссылку, хранящуюся в переменной *b*. Это приводит к важным последствиям. Во-первых,

программа теряет связь с массивом, содержащим элементы 6, 7, поскольку эта связь обеспечивалась только за счет ссылки, хранящейся в `b`. Массив становится недоступным программе и в дальнейшем автоматически удаляется из памяти специальной подсистемой платформы .NET — *сборщиком мусора* (garbage collector) [3, п. 8.1.2]. Во-вторых, доступ к массиву с элементами 1, 2, 3 теперь оказывается возможным и с помощью переменной `a`, и с помощью переменной `b`.

Дополним предыдущий фрагмент следующими операторами

```
b[0] := 5;
Write(a); // [5, 2, 3]
```

Мы видим, что изменение элемента `b[0]` привело к тому, что одновременно изменился и массив `a` (так как фактически у нас имеется единственный массив с двумя именами).

Указанное обстоятельство необходимо учитывать при передаче динамических массивов в качестве параметров подпрограмм. В частности, следует понимать, что и передача динамического массива по значению, и передача его по ссылке (с указанием модификатора `var`) будет, во-первых, выполняться одинаково быстро (так как передача по значению не будет приводить к поэлементному копированию) и, во-вторых, позволит *изменить* элементы переданного массива (даже при передаче массива по значению, поскольку созданная в подпрограмме при такой передаче *копия ссылки* будет обращаться к тому же самому массиву).

Как при подобном способе присваивания создать настоящую копию массива, которую в дальнейшем можно было бы обрабатывать независимо? Для этого в PascalABC.NET предусмотрено несколько способов.

Простейшим является использование функции `Copy(a)` с единственным параметром — копируемым массивом. Эта функция создает и возвращает копию данного массива (точнее, возвращается *ссылка* на созданную в памяти копию). Пример:

```
var a := Arr(1, 2, 3);
var b := Arr(6, 7);
b := Copy(a);
b[0] := 5;
WriteLn(a); // [1, 2, 3]
WriteLn(b); // [5, 2, 3]
```

Мы видим, что теперь изменение значения элемента массива `b` никак не повлияло на содержимое массива `a`.

Заметим, что для получения копий массива или его частей можно использовать *срезы* (см. п. 3.7), поскольку любой срез фактически является копией некоторой части массива. Например, в предыдущем фрагменте мы



могли вместо оператора присваивания с функцией `Copy` использовать вариант со срезом:

```
b := a[:];
```

Важно понимать, что при описанных способах копирования выделяется дополнительная память для хранения копии.

Копию можно создать без выделения памяти, если скопировать данные из одного массива (массива-источника) непосредственно в другой (массив-приемник). Для этого можно использовать процедуру (метод массива) `CopyTo`, вызов которой имеет вид `src.CopyTo(dst, dstStart)`, где `src` определяет массив-источник (от англ. *source*), `dst` — массив-приемник (от англ. *destination*), а `dstStart` — индекс в массиве `dst`, начиная с которого в него будут записываться элементы массива `src`. Всегда копируются *все* элементы массива-источника; если для этого действия в массиве-приемнике недостаточно элементов, то будет возбуждено исключение.

Например, если массивы `a` и `b` инициализированы так же, как в предыдущем примере, то вызов `a.CopyTo(b, 0)` приведет к аварийному завершению программы, а вызов `b.CopyTo(a, 0)` выполнится успешно (в результате в `a` будут содержаться три элемента: 6, 7, 3). Успешно проработает и вызов `b.CopyTo(a, 1)`; в этом случае в `a` будут содержаться элементы 1, 6, 7.

С помощью метода `CopyTo` можно реализовать короткий и эффективный алгоритм *объединения* массивов, не использующий циклы. Предположим, что требуется объединить элементы массивов `a` и `b` (в указанном порядке) в новом массиве `res`. Вначале надо выделить память для массива `res` (при этом его размер должен быть равен суммарному размеру исходных массивов), а затем вызвать процедуру `CopyTo` дважды: для копирования в нужные позиции массива `res` элементов массива `a` и `b`, например:

```
var res := new integer[a.Length + b.Length];  
a.CopyTo(res, 0);  
b.CopyTo(res, a.Length);
```

В языке `PascalABC.NET` предусмотрен специальный вариант операции `+` для динамических массивов, позволяющий для формирования массива `res` использовать вместо указанных выше операторов единственный оператор:

```
var res := a + b;
```

Операция `a + b` размещает в памяти массив и заполняет его нужным образом, после чего *ссылка* на эту область памяти записывается в переменную `res`. Обратите внимание на то, что в данном случае тип массива `res` выводится из типов массивов `a` и `b`; необходимо лишь, чтобы типы массивов `a` и `b` совпадали.

Проиллюстрируем применение рассмотренных возможностей, решив задачу **Array57**. В ней требуется записать в новый целочисленный массив **b** вначале все элементы исходного массива **a** с четными порядковыми номерами (2, 4, 6, ...), а затем — с нечетными (1, 3, 5, ...).

При решении надо учесть, что индекс элемента на 1 меньше порядкового номера, поэтому вначале надо записать в массив **b** все элементы массива **a** с нечетными индексами, а затем — с четными:

```
Task('Array57'); // Вариант 1
var a := ReadArrInteger;
var b := a[1::2] + a[0::2];
b.Print;
```

Мы могли бы не связывать полученный массив с переменной **b**, а сразу вывести его:

```
Task('Array57'); // Вариант 1a
var a := ReadArrInteger;
(a[1::2] + a[0::2]).Print;
```

Это не нарушает условия задачи, так как в данном случае мы также сформировали новый массив (просто при этом мы не связали созданный массив с каким-либо именем).

Для сравнения приведем решение, использующее цикл:

```
Task('Array57'); // Вариант 2
var a := ReadArrInteger;
var b := new integer[a.Length];
for var i := 0 to a.Length div 2 - 1 do
  b[i] := a[2 * i + 1];
for var i := 0 to (a.Length + 1) div 2 - 1 do
  b[a.Length div 2 + i] := a[2 * i];
b.Print;
```

Разумеется, по части наглядности и простоты реализации этот вариант не выдерживает сравнения с предыдущим. Правда, он более эффективно использует память: кроме исходного массива **a** память выделяется только для результирующего массива **b**. В первом варианте память выделялась для массива **a**, для каждого из двух срезов, а также для суммы этих срезов (ссылка на которую записывалась в **b**). Если считать, что массив **a** содержит  $N$  элементов, то во втором варианте программы выделялась память для хранения  $2N$  элементов (массивы **a** и **b**), а в первом — для хранения  $3N$  элементов (массив **a**, два среза и их сумма). Однако в большинстве ситуаций подобный «перерасход» памяти (и связанное с ним небольшое замедление алгоритма) можно не принимать во внимание.

**Замечание.** Во втором варианте решения были использованы формулы, позволяющие определить количество элементов с нечетными и четными

ми индексами для массива размера  $N$ . Для элементов с нечетными индексами формула имеет вид  $N \div 2$ , а для элементов с четными индексами —  $(N + 1) \div 2$ . При этом число  $N$  может быть как четным, так и нечетным.

## 5.2. Изменение размеров динамического массива

В определении массива, данном в п. 3.1, особо отмечалось, что массив размещается на *непрерывном* участке оперативной памяти. Это условие является важным, так как оно обеспечивает быстрое нахождение адреса любого элемента массива по его индексу. С другой стороны, отмеченное условие, казалось бы, не позволяет *изменять размер* уже созданного массива. Действительно, после участка, на котором располагается массив, могут быть размещены другие данные программы, и поэтому «расширить» память для массива (сохраняя ее в виде непрерывного участка) окажется невозможно. Это справедливо для статических массивов, которые, будучи созданы, сохраняют свой размер на протяжении всей программы. Однако для динамических массивов ситуация иная.

Во всех современных реализациях Паскаля предусмотрены средства, позволяющие изменить размер динамического массива (эти средства были добавлены в язык одновременно с включением в него динамических массивов; из распространенных реализаций Паскаля это впервые было сделано в языке Delphi Pascal версии 4). Таким образом, слово «динамический» в отношении массивов PascalABC.NET означает не только то, что массив можно создать *в любой момент* выполнения программы, но и то, что в дальнейшем его размер *можно изменить с сохранением прежнего содержания*.

Для изменения размера массива  $a$  достаточно вызвать процедуру `SetLength(a, newLength)`, в которой второй параметр определяет новый размер. В результате переменная  $a$  будет связана с массивом нового размера (который может быть больше или меньше размера исходного массива), причем все сохранившиеся элементы исходного массива сохранят свои значения (а новые будут содержать нулевые значения соответствующего типа).

Следует, однако, иметь в виду, что операция изменения размера динамического массива является *длительной* и *ресурсоемкой* операцией. Она требует, во-первых, выделения памяти под новый массив и, во-вторых, копирования в новый массив сохраняемой части исходного массива.

Заметим, что процедуру `SetLength` можно вызвать даже для *неинициализированного* массива, т. е. сразу после его описания (когда переменная  $a$  имеет значение `nil` и, таким образом, не связана с каким-либо конкретным массивом, размещенным в памяти). Это означает, что процедуру `SetLength` можно использовать для *инициализации массива*, как и конструктор `new`. Именно с помощью процедуры `SetLength` обеспечивается инициализация

динамических массивов в языке Delphi Pascal, тогда как вариант инициализации с применением конструктора `new` заимствован языком Pascal-ABC.NET из языка C#. Вариант с конструктором является более кратким, так как не требует предварительного описания массива. Ниже приведены два варианта описания и инициализации массивов одинакового размера: для массива `a1` использован конструктор, для массива `a2` — процедура `SetLength`:

```
var a1 := new integer[10];
var a2: array of integer;
SetLength(a2, 10);
```

Чтобы проиллюстрировать особенности применения процедуры `SetLength`, рассмотрим задачу **Array54**, в которой требуется переписать в новый массив `b` все четные числа из исходного массива `a` в том же порядке и вывести размер полученного массива и его элементы. Одним из вариантов решения может быть следующий:

```
Task('Array54'); // Вариант 1
var a := ReadArrInteger;
var b := new integer[0];
foreach var e in a do
  if not Odd(e) then
    begin
      SetLength(b, b.Length + 1);
      b[b.Length - 1] := e;
    end;
b.WriteAll;
```

Вначале мы создаем пустой результирующий массив `b`, а затем в цикле, как только в исходном массиве обнаруживается четное число, увеличиваем размер массива `b` и добавляем это число в конец увеличенного массива. Для вывода как размера массива, так и его элементов мы использовали специальный метод вывода `WriteAll`, предусмотренный в задачнике. Заметим, что создать пустой массив целого типа можно было бы и с помощью функции-генератора `ArrFill(0, 0)`.

Это решение дает правильный результат и выглядит достаточно привлекательно. Однако многократные вызовы процедуры `SetLength` при обработке массивов большого размера могут существенно замедлить выполнение этого алгоритма. Для того чтобы это продемонстрировать, будем генерировать большие массивы случайных чисел (используя функцию `ArrRandom` — см. п. 3.3), применять к ним описанный алгоритм и замерять время его работы. Время генерации исходного массива не будем учитывать, не будем также выводить на печать полученный массив:

```
var size := 10000;
```

```
for var k := 1 to 6 do
begin
  size *= 2;
  var a := ArrRandom(size);
  MillisecondsDelta;
  // начало алгоритма
  var b := new integer[0];
  foreach var e in a do
    if not Odd(e) then
      begin
        SetLength(b, b.Length + 1);
        b[b.Length - 1] := e;
      end;
  // конец алгоритма
  WriteInFormat('size = {0,6} time = {1,6}', size,
    MillisecondsDelta);
end;
```

В данном случае для измерения времени использовалась функция `MillisecondsDelta`. Ее отличие от уже известной нам функции `Milliseconds` (см. п. 3.4) состоит в том, что функция `MillisecondsDelta` возвращает время (в миллисекундах), прошедшее с момента *последнего вызова* этой же функции (при первом вызове функции `MillisecondsDelta`, как и при вызове `Milliseconds`, возвращается время, прошедшее с момента начала работы программы). Обратите внимание на то, что первый из двух вызовов функции `MillisecondsDelta` имеет вид вызова *процедуры*, поскольку в данном случае нам не требуется возвращаемое значение: мы лишь «помечаем» начало того участка программы, время работы которого требуется измерить.

Полужирным шрифтом выделен фрагмент, для которого подсчитывается время работы. Приведем возможный результат выполнения программы (напомним, что программу следует запускать в *режиме релиза* — см. замечание в п. 3.4):

```
size = 20000 time = 36
size = 40000 time = 147
size = 80000 time = 1483
size = 160000 time = 8682
size = 320000 time = 39110
size = 640000 time = 147069
```

Мы видим что, время растет с большей скоростью, чем размер `size`.

Алгоритм можно записать короче, если использовать операцию `+` для массивов:

```
Task('Array54'); // Вариант 2
```

```
var a := ReadArrInteger;  
var b := new integer[0];  
foreach var e in a do  
  if not Odd(e) then  
    b := b + Arr(e);  
b.WriteAll;
```

В этом решении мы в цикле добавляем к массиву *b* одноэлементный массив, содержащий элемент *e*. Быстродействие у данного варианта решения будет таким же, как и у предыдущего.

Теперь рассмотрим вариант решения, не требующий многократного выделения памяти и копирования элементов:

```
Task('Array54'); // Вариант 3  
var a := ReadArrInteger;  
var b := new integer[a.Length];  
var m := 0;  
foreach var e in a do  
  if not Odd(e) then  
    begin  
      b[m] := e;  
      m += 1;  
    end;  
SetLength(b, m);  
b.WriteAll;
```

В этом варианте для массива *b* сразу выделяется наибольший возможный размер (совпадающий с размером исходного массива). Кроме того, вводится переменная *m*, определяющая «заполненность» массива *b* реальными элементами, полученными из *a*. В начале алгоритма заполненность равна 0. В цикле четные элементы добавляются в конец *заполненной* части массива и одновременно увеличивается *m*. После завершения цикла останется *уменьшить* размер массива *b*, оставив в нем только заполненную часть. Таким образом, в этом варианте процедура *SetLength* вызывается только один раз.

**Замечание.** Можно было бы вообще не вызывать в этой программе процедуру *SetLength*, но тогда в массиве *b* останутся «лишние» нулевые элементы, что не позволит использовать для его вывода метод *WriteAll* (поскольку этот метод выведет все элементы, в том числе и «лишние»). В этой ситуации можно было бы вывести в качестве размера число *m*, а для вывода элементов использовать цикл *for*. Если же полученный таким способом массив предполагается использовать в других частях программы, то вызов процедуры *SetLength* существенно упростит его дальнейшее использование,

поскольку в противном случае значение свойства `Length` для этого массива не будет соответствовать размеру его «истинного» содержимого.

Тестирование нового варианта алгоритма для массивов большого размера дает следующий результат:

```
size = 20000 time = 1
size = 40000 time = 1
size = 80000 time = 1
size = 160000 time = 1
size = 320000 time = 3
size = 640000 time = 6
```

Мы получили гораздо более эффективный алгоритм.

Приведем еще один вариант решения, использующий изученные нами в предыдущей главе *запросы*. В данном случае достаточно применить *запрос фильтрации* `Where` (см. п. 4.2), отбирающий нужные элементы. Чтобы полученную последовательность можно было сохранить в виде массива (как того требуют условия задачи), достаточно применить к ней *экспортирующий запрос* `ToArray`, рассмотренный в п. 4.6:

```
Task('Array54'); // Вариант 4
var a := ReadArrInteger;
var b := a.Where(e -> not Odd(e)).ToArray;
b.WriteAll;
```

Для проверки быстродействия этого варианта в программе замера времени между комментариями «начало алгоритма» и «конец алгоритма» надо поместить *единственный* оператор:

```
var b := a.Where(e -> not Odd(e)).ToArray;
```

Проверка быстродействия показывает, что оно будет почти таким же, как и у варианта 3 (использующего «счетчик заполненности» `m`):

```
size = 20000 time = 3
size = 40000 time = 1
size = 80000 time = 1
size = 160000 time = 2
size = 320000 time = 5
size = 640000 time = 10
```

Конечно, в отношении наглядности вариант со счетчиком заполненности существенно проигрывает варианту, использующему запросы.

Если немного отступить от условий задачи, то в последнем варианте решения можно обойтись и без запроса `ToArray`. В этом случае в переменной `b` будет сохраняться ссылка не на массив, а на *последовательность*, содержащую требуемый набор данных:

```
Task('Array54'); // Вариант 4a
```

```
var a := ReadArrInteger;  
var b := a.Where(e -> not Odd(e));  
b.WriteAll;
```

Данное решение также будет засчитано как правильное. Однако нам не удастся проверить его быстродействие, подставив в программу замера времени оператор

```
var b := a.Where(e -> not Odd(e));
```

При запуске этого варианта мы получим следующий результат:

```
size = 20000 time = 3  
size = 40000 time = 0  
size = 80000 time = 0  
size = 160000 time = 0  
size = 320000 time = 0  
size = 640000 time = 0
```

Разумеется, это не означает, что в данном варианте алгоритм работает менее 1 миллисекунды для исходных массивов *любого размера* (большего 20000). Вспомним, что основная особенность запросов, преобразующих последовательности, состоит в их *отложенном выполнении*. Таким образом, в варианте 4а при формировании последовательности *b* не выполняется фактическая обработка исходного массива *a*: в последовательность *b* лишь записывается «правило», по которому эта обработка будет происходить *впоследствии*, когда, например, элементы последовательности будут перебираться в цикле `foreach` или когда последовательность будет преобразована в массив запросом `ToArray` (как в варианте 4). Понятно, что для записи *правила обработки* требуется очень мало времени и, главное, это время не зависит от размера обрабатываемых массивов.

**Замечание.** Значение 2, выведенное в первой строке, связано с выполнением некоторых дополнительных действий программы при *первом* обращении к запросу `Where`. Если, например, вызвать запрос `Where` в начале программы (применив его к произвольной последовательности и указав произвольный предикат), то в выведенном списке *все* значения `time` будут равны нулю.

### 5.3. Поиск в динамических массивах

В главе 4 мы познакомились с некоторыми запросами, которые можно использовать при организации поиска в последовательностях. Это, прежде всего, запросы `First` и `Last` (и их варианты `FirstOrDefault` и `LastOrDefault`) с параметром — лямбда-выражением, а также запросы-квантификаторы (см. п. 4.1). Можно считать, что специализированный поиск выполняют и агрегирующие запросы `Max` и `Min` (и их варианты `MaxBy`, `MinBy`, `LastMaxBy`, `LastMinBy`), также описанные в п. 4.1. Для поиска в динамических массивах в



библиотеке PascalABC.NET предусмотрен более обширный набор методов. Перечислим эти методы, предполагая, что они применяются к массиву типа `array of T`:

Поиск	Методы <code>array of T</code>
<b>Find</b> (pred: T -> boolean): T	
<b>FindLast</b> (pred: T -> boolean): T	
<b>FindAll</b> (pred: T -> boolean): array of T	
<b>FindIndex</b> ([start: integer;] pred: T -> boolean): integer	
<b>FindLastIndex</b> ([start: integer;] pred: T -> boolean): integer	
<b>IndexOf</b> (x: T [; start: integer]): integer	
<b>LastIndexOf</b> (x: T [; start: integer]): integer	
<b>IndexMax</b> ([start: integer]): integer	
<b>LastIndexMax</b> ([start: integer]): integer	
<b>IndexMin</b> ([start: integer]): integer	
<b>LastIndexMin</b> ([start: integer]): integer	

Методы `Find` и `FindLast` выполняются аналогично запросам `FirstOrDefault` и `LastOrDefault`: они возвращают соответственно первый и последний элементы массива, удовлетворяющие предикату `pred`, или нулевое значение типа `T`, если массив не содержит требуемых элементов. При поиске в массиве следует использовать именно эти методы, а не аналогичные им по назначению запросы последовательностей, так как методы массивов выполняются быстрее.

Вызов метода `a.FindAll(pred)` возвращает массив, содержащий все элементы массива `a`, удовлетворяющие предикату `pred`. Он может рассматриваться как аналог цепочки из двух запросов вида `a.Where(pred).ToArray` и выполняется с такой же скоростью.

Все прочие методы поиска для массивов не имеют аналогов среди запросов, так как возвращают *индексы* найденных элементов. Ищется индекс либо первого/последнего элемента, удовлетворяющего указанному предикату `pred` (методы `FindIndex` и `FindLastIndex`), либо элемента с указанным значением `x` (методы `IndexOf` и `LastIndexOf`); если требуемые элементы отсутствуют, то все эти методы возвращают особое значение `-1`. К этой же группе можно отнести методы `IndexMax`, `LastIndexMax`, `IndexMin`, `LastIndexMin`, позволяющие найти индекс первого/последнего максимального или минимального элемента массива.

У всех методов, возвращающих индекс найденного элемента массива, предусмотрен дополнительный параметр `start`, позволяющий выполнять поиск, начиная с элемента с индексом `start`. Для методов, просматривающих массив с начала в поисках *первого* подходящего элемента, индекс `start` по умолчанию считается равным `0`, для методов, просматривающих массив

с конца в поисках *последнего* подходящего элемента, индекс `start` по умолчанию считается равным  $N - 1$ , где  $N$  обозначает размер массива.

Использование параметра `start` позволяет организовать в цикле поиск индексов *всех* требуемых элементов как в возрастающем, так и в убывающем порядке. Например, чтобы напечатать индексы всех элементов массива `a`, равных значению `x`, можно использовать следующий фрагмент:

```
var i := a.IndexOf(x);
while i <> -1 do
begin
  Print(i);
  i := a.IndexOf(x, i + 1);
end;
```

Для того чтобы подобные фрагменты всегда успешно завершались, в методах `FindIndex` и `IndexOf` разрешено в качестве `start` указывать не только допустимые индексы от 0 до  $N - 1$  (где  $N$  — размер массива), но и значение  $N$ , при указании которого *всегда* возвращается  $-1$ . К сожалению, для методов `FindLastIndex` и `LastIndexOf` *нельзя* указывать значение параметра `start`, равное  $-1$ , что не позволяет организовать поиск всех требуемых элементов в обратном порядке так же просто, как и в прямом.

Для методов, связанных с нахождением индексов максимальных и минимальных элементов, использование параметра `start` позволяет найти индекс максимального или минимального элемента *в указанной части массива* (при этом значение соответствующего элемента может отличаться от значения максимума или минимума для массива в целом). В данном случае `start` может содержать только допустимые индексы массива. Пример:

```
var a := Arr(1, 2, 3, 4, 5);
for var i := 0 to a.Length - 1 do
  Print(a.IndexMin(i)); // 0 1 2 3 4
```

Если требуется проверить сам факт наличия в массиве `a` элемента со значением `x` и не требуется определять его индекс, то соответствующую проверку можно записать максимально коротким способом, используя выражение `x in a`, которое возвращает значение `True`, если `x` содержится в массиве `a`, и `False` в противном случае.

В дополнение к рассмотренным выше средствам поиска, для массивов предусмотрены еще два специализированных метода:

#### **Специальный поиск**

Методы `array of T`

**BinarySearch**(`x: T`): `integer`

**AdjacentFind**([`pred: (T, T) -> boolean`;] [`start: integer`]): `integer`

Метод `BinarySearch` позволяет выполнять очень эффективный поиск в массиве, отсортированном по возрастанию. По сравнению с «обычными»

методами поиска он имеет две особенности. Во-первых, при наличии нескольких элементов с требуемым значением (разумеется, все такие элементы в отсортированном массиве располагаются подряд) он возвращает индекс *какого-либо* из этих элементов, не обязательно первого или последнего. Во вторых, если требуемое значение в массиве не найдено, метод `BinarySearch` возвращает не особое значение  $-1$ , а некоторое отрицательное число, определяющее позицию, в которую *можно было бы* добавить требуемое значение с сохранением упорядоченности элементов массива. Более точно, если возвращено отрицательное число  $-k$ , то это означает, что требуемое значение (при его наличии) должно было бы располагаться *перед* элементом с индексом  $k - 1$ . Примеры:

```
var a:= Arr(11, 22, 33, 33, 44, 44, 44, 44, 44, 55, 55, 55, 55, 66);
Println(a.BinarySearch(33)); // 2
Println(a.BinarySearch(44)); // 6
Println(a.BinarySearch(55)); // 10
Println(a.BinarySearch(0)); // -1
Println(a.BinarySearch(40)); // -5
Println(a.BinarySearch(50)); // -10
Println(a.BinarySearch(80)); // -15
```

Варианты метода `AdjacentFind` позволяют искать *пары* соседних элементов, удовлетворяющие предикату `pred`, и возвращают индекс левого элемента первой найденной пары. В первый параметр предиката подставляется левый элемент каждой пары, а во второй — правый элемент. Если предикат `pred` не указан, то ищутся пары *совпадающих* элементов (иными словами, в данном случае используется предикат вида  $(e1, e2) \rightarrow e1 = e2$ ). Если указан параметр `start`, то поиск начинается с пары, левый элемент которой имеет индекс `start`; если параметр `start` отсутствует, то он считается равным 0. Если в массиве отсутствуют пары, удовлетворяющие требуемому условию, то метод возвращает значение  $-1$ . Примеры:

```
var a := Arr(11, 22, 33, 33, 44, 66, 55, 15);
Println(a.AdjacentFind); // 2
Println(a.AdjacentFind(3)); // 4
Println(a.AdjacentFind((e1, e2) -> (e1 + e2) mod 10 = 0)); // 5
Println(a.AdjacentFind((e1, e2) -> e1 > e2)); // 6
Println(a.AdjacentFind((e1, e2) -> e1 = -e2)); // -1
```

В отношении скорости выполнения метод `AdjacentFind` эквивалентен соответствующему циклу по переменной `i`, в котором анализируются элементы массива с индексами `i` и `i + 1`.

Для иллюстрации применения некоторых из описанных методов обратимся к задачам из группы `Array`.

В задаче **Array47** требуется найти количество различных элементов в исходном целочисленном массиве.

Если использовать только средства традиционного Паскаля и не применять дополнительные структуры данных, то решение можно представить в следующем виде:

```
Task('Array47'); // Вариант 1
var a := ReadArrInteger;
var k := 1;
for var i := 1 to a.Length - 1 do
begin
  var p := 1;
  for var j := i - 1 downto 0 do
    if a[j] = a[i] then
      begin
        p := 0;
        break;
      end;
    k += p;
  end;
Write(k);
```

Мы сразу, до цикла, записываем в счетчик *k* значение 1, чтобы учесть первый элемент массива. Остальные элементы анализируются в цикле; счетчик увеличивается только если значение очередного элемента *a[i]* ранее в массиве не встречалось. Для поиска значения *a[i]* в предыдущей части массива приходится организовывать вложенный цикл по переменной *j*. Результатом поиска является переменная *p*, равная 1, если элемент *a[i]* ранее в массиве не встречался, и равная 0 в противном случае. Полученное значение переменной *p* можно без дополнительной проверки добавить к счетчику.

С применением одного из методов поиска алгоритм можно существенно сократить:

```
Task('Array47'); // Вариант 2
var a := ReadArrInteger;
var k := 1;
for var i := 1 to a.Length - 1 do
  if a.LastIndexOf(a[i], i - 1) = -1 then
    k += 1;
Write(k);
```

Еще более коротко можно записать условие, используя операцию *in* и выражение для среза массива:

```
Task('Array47'); // Вариант 3
```

```

var a := ReadArrInteger;
var k := 1;
for var i := 1 to a.Length - 1 do
  if not (a[i] in a[:i]) then
    k += 1;
Write(k);

```

Однако все эти способы решения проигрывают в краткости и наглядности способу, использующему запрос `Distinct` (см. п. 4.2), который возвращает последовательность, не содержащую одинаковых элементов, после чего остается определить ее размер с помощью запроса `Count` (см. п. 4.1):

```

Task('Array47'); // Вариант 4
var a := ReadArrInteger;
var k := a.Distinct.Count;
Write(k);

```

Вариант 4 можно оформить в виде *единственного* оператора:

```

Task('Array47'); // Вариант 4a
Write(ReadArrInteger.Distinct.Count);

```

Интересно проанализировать скорость выполнения полученных алгоритмов для массивов большого размера. В таблице 5.1 приводятся результаты измерения времени (в миллисекундах) для четырех алгоритмов, обрабатывающих один и тот же массив размера `size`. Для генерации массива использовался метод `ArrRandom(size, 1, size div 2)`. Полученный результат `k` (одинаковый для всех алгоритмов) выводится в последнем столбце.

**Таблица 5.1.** Быстродействие вариантов решения задачи *Array47*

Размер size	Вариант 1	Вариант 2	Вариант 3	Вариант 4	Результат k
20000	185	141	482	2	8662
40000	672	559	2863	2	17320
80000	2666	2229	12550	5	34594
160000	10803	9074	52551	9	69111

Мы видим, что последний алгоритм является не только самым коротким и наглядным, но и потрясающе быстрым, оставляя далеко позади все предыдущие варианты. Вторым по производительности оказывается алгоритм 2. Сравнивая его результаты с алгоритмом 1, мы убеждаемся, что реализация стандартного метода поиска в массиве работает быстрее, чем обычный перебор в цикле. Несколько неожиданным может показаться очень большое время работы алгоритма 3. Однако это объясняется тем, что в данном алгоритме на каждой итерации цикла создается и размещается в памяти *новый массив* (срез исходного массива), на что тратится дополнительное время (а также дополнительная память).

## 5.4. Преобразование динамических массивов

Для динамических массивов предусмотрен набор методов, выполняющих различные *преобразования* их элементов. Большинство этих методов (кроме `ConvertAll`) изменяют исходный массив. Некоторые из них (`ConvertAll` и `Shuffle`) являются функциями и возвращают преобразованный массив, другие (`Replace`, `Sort` и `Transform`) являются процедурами. Все методы оставляют неизменным размер массива; все, кроме `ConvertAll`, сохраняют неизменным тип его элементов.

### Преобразование

Методы `array of T`

**ConvertAll**(conv: T -> TRes): `array of TRes`

**Shuffle**: `array of T`

**Replace**(oldValue, newValue: T)

**Sort**([comp: (T, T) -> integer])

**Transform**(func: T -> T)

Функция `ConvertAll` применяет к каждому элементу `e` исходного массива преобразование `conv(e)` (переданное в качестве параметра — лямбда-выражения) и возвращает новый массив, содержащий преобразованные элементы. Преобразование `conv` может, в частности, изменять тип элементов. Исходный массив не изменяется.

Функция `Shuffle` изменяет случайным образом порядок элементов исходного массива. Она не только изменяет исходный массив, но и возвращает ссылку на него.

Процедура `Replace` заменяет в исходном массиве все элементы со значением `oldValue`, присваивая им новое значение `newValue`.

Процедура `Sort` сортирует исходный массив. Если она вызывается без параметров, то массив сортируется по возрастанию значений его элементов (в этом случае необходимо, чтобы элементы можно было сравнивать операциями «меньше»–«больше»). Необязательный параметр `comp` позволяет задать способ сортировки, определяя отношение сравнения для элементов массива: если значение `a` меньше значения `b`, то `comp(a, b)` должно возвращать отрицательное число, если `a` равно `b`, то `comp(a, b)` должно возвращать 0, если `a` больше `b`, то `comp(a, b)` должно возвращать положительное число. Вариант процедуры `Sort` с параметром `comp` можно вызывать для любых массивов, в том числе и тех, для элементов которых не определены стандартные операции сравнения «меньше»–«больше».

Процедура `Transform(func)` преобразует исходный массив, применяя к каждому его элементу лямбда-выражение `func`. Поскольку изменяется сам исходный массив, преобразование `func(e)` должно возвращать значение, тип которого совпадает с типом параметра `e` (в этом состоит одно из отличий процедуры `Transform` от ранее рассмотренной функции `ConvertAll`).

Кроме *методов* преобразования массивов (вызываемых с применением точечной нотации) в библиотеке PascalABC.NET имеются две *процедуры*, предназначенные для преобразования массива и принимающие этот массив в качестве параметра:

```
procedure Reverse(a: array of T; start, count: integer)
procedure Sort(a: array of T)
```

Процедура `Reverse` инвертирует массив `a` (или его часть из `count` элементов, начиная с индекса `start`). Эта процедура ранее обсуждалась и использовалась в п. 3.7. Процедура `Sort` сортирует массив `a`; ее действие аналогично применению одноименного метода.

В качестве иллюстрации одного из описанных методов преобразования рассмотрим задачу **Array88**. В этой задаче дан массив вещественных чисел, все элементы которого, кроме последнего, уже отсортированы по возрастанию. Требуется переместить последний элемент на новую позицию в массиве таким образом, чтобы весь массив стал упорядоченным по возрастанию.

Наиболее эффективное решение данной задачи требует однократного просмотра массива с одновременным сдвигом его элементов вправо, пока не будет найдена позиция `p` для размещения последнего элемента:

```
Task('Array88'); // Вариант 1
var a := ReadArrReal;
var b := a[a.Length - 1];
var p := 0;
for var i := a.Length - 2 downto 0 do
  if a[i] > b then
    a[i + 1] := a[i]
  else
    begin
      p := i + 1;
      break;
    end;
a[p] := b;
a.Print;
```

В алгоритме учитывается особая ситуация, при которой последний элемент массива (который хранится во вспомогательной переменной `b`) окажется *меньше* всех предыдущих элементов. В этом случае условие `a[i] > b`, проверяемое в цикле, никогда не станет ложным, поэтому значение переменной `p` останется равным 0, и после выхода из цикла значение `b` будет записано в начальный элемент массива.

Возникает вопрос: насколько хуже данного варианта будет вариант, просто сортирующий исходный массив? Ведь в стандартной библиотеке

реализован весьма эффективный алгоритм сортировки! Во всяком случае, с точки зрения краткости решения новый вариант будет существенно более выигрышным:

```
Task('Array88'); // Вариант 2
var a := ReadArrReal;
a.Sort;
a.Print;
```

Заодно можно сравнить эффективность метода `Sort` и аналогичного *запроса* сортировки `Order` (см. п. 4.2), применение которого тоже дает правильный вариант решения (обратите внимание на вызов запроса `ToArray`, преобразующего отсортированную последовательность «обратно» в массив):

```
Task('Array88'); // Вариант 3
var a := ReadArrReal;
a := a.Order.ToArray;
a.Print;
```

Заметим, что у этого варианта имеется один очевидный недостаток: отсортированный набор возвращается в качестве новой последовательности и при ее преобразовании в массив выделяется новый участок памяти. В варианте 2 выполнялась сортировка элементов самого исходного массива, и поэтому дополнительная память не выделялась. Таким образом, в отношении использования памяти вариант 3 уступает как варианту 1, так и варианту 2.

Для проверки быстродействия алгоритмов необходимо обрабатывать массивы большого размера `size`. Для генерации набора из `size` возрастающих вещественных чисел удобно использовать запрос `Range` (см. п. 3.5): `Range(0.0, 1.0, size - 1)`. В результате будет создан набор из `size` вещественных чисел, равномерно распределенных на отрезке `[0; 1]` (включая концы). После этого останется преобразовать полученную последовательность в массив запросом `ToArray` и изменить значение последнего элемента, положив его равным, например, `0.5` (в этом случае для решения задачи потребуется переместить этот элемент в *середину* массива).

Полученный массив обрабатывался каждым из приведенных выше алгоритмов; при этом измерялось время работы алгоритма в миллисекундах (время, использованное на генерацию массива, не учитывалось). Программа запускалась в *режиме релиза* (см. замечание в п. 3.4), как и все рассмотренные ранее программы для проведения численного эксперимента.

В таблице 5.2 приведены результаты работы программы, а также информация о том, во сколько раз варианты 2 и 3 работают медленнее варианта 1: в столбцах с заголовками «Var.2 / Var.1» и «Var.3 / Var.1» указы-



ваются значения, полученные при делении времени работы варианта 2 и варианта 3 на время работы варианта 1.

*Таблица 5.2. Быстродействие вариантов решения задачи Array88*

Размер size	Вар. 1	Вар. 2	Вар. 3	Вар.2 / Вар.1	Вар.3 / Вар.1
8000000	11	229	2013	20.8	183.0
16000000	22	448	4203	20.4	191.0
32000000	42	947	8851	22.5	210.7
64000000	85	1984	18949	23.3	222.9

Мы видим, что вариант 2, выполняющий сортировку массива, работает медленнее варианта 1 в 20 раз, а вариант 3, использующий для сортировки запросы `Sorted` и `ToArray`, медленнее примерно в 200 раз! Таким образом, в отношении эффективности первый вариант существенно превосходит два других (по крайней мере, в случае обработки массивов большого размера). Полученные результаты свидетельствуют также о том, что для сортировки массивов большого размера следует использовать специально разработанные методы, а не универсальные запросы сортировки для последовательностей.

## Литература

1. *Абрамян М. Э.* Структуры данных в PascalABC.NET. Выпуск 2. Минимумы и максимумы. Списки, множества, словари, стеки и очереди. Многомерные структуры. — Ростов н/Д : Изд-во ЮФУ, 2016. — 118 с.
2. *Абрамян М. Э.* Практикум по программированию на языке Паскаль: Массивы, строки, файлы, рекурсия, линейные динамические структуры, бинарные деревья. — 7-е изд., перераб. и доп. — Ростов н/Д : Изд-во ЮФУ, 2010. — 276 с.
3. *Абрамян М. Э.* Платформа .NET: Основные типы стандартной библиотеки. Работа с массивами, строками, файлами. Объекты, интерфейсы, обобщения. Технология LINQ. — Ростов н/Д : Изд-во ЮФУ, 2014. — 218 с.
4. *Абрамян М. Э.* Технология LINQ на примерах. Практикум с использованием электронного задачника Programming Taskbook for LINQ. — М. : ДМК Пресс, 2014. — 326 с.
5. *Абрамян М. Э., Михалкович С. С.* Основы программирования на языке Паскаль. Скалярные типы данных, управляющие операторы, процедуры и функции, работа с графикой в системе PascalABC.NET. 4-е изд., перераб. и доп. — Ростов н/Д : Изд-во «ЦВВР», 2008. — 223 с.
6. *Долинер Л. И.* Основы программирования в среде PascalABC.NET. — Екатеринбург : Изд-во Урал. ун-та, 2014. — 128 с. URL: <http://elar.urfu.ru/handle/10995/28702> (дата обращения 02.08.2016).
7. *Куклин Д. В.* Учебник по программированию. Первые шаги. Язык программирования PascalABC.NET. — Киров, 2014. URL: [http://dvkuklin.ru/fs\\_ABC/annotation.html](http://dvkuklin.ru/fs_ABC/annotation.html) (дата обращения 02.08.2016).

## Указатель

Чтобы упростить использование указателя, запросы, методы и свойства в нем размещены в двух местах: в алфавитном порядке их имен и в соответствующих группах. В частности, все запросы дополнительно указаны в группе «Запросы», все функции-генераторы — в группе «Генераторы», а все методы и свойства массивов — в группе «Динамические массивы». Сгруппированы также подпрограммы ввода, подпрограммы вывода и подпрограммы, описанные в модуле РТ4. В группе «Решения задач» перечислены все задачи из задачника Programming Taskbook, рассмотренные в данной книге, с указанием страниц, на которых приведены различные варианты их решения.

- ?, тернарная операция, 65, 76
- +
  - операция для кортежа, 15
  - операция для массивов, 96
  - особенность выполнения при наличии строкового или символьного операнда, 82
- Abs, функция, 87
- Add, метод кортежа, 15
- AdjacentFind, метод массива, 106
- Aggregate, запрос, 63
- All, запрос, 62
- Any, запрос, 62
- Arr, генератор массива (короткая функция), 35, 89
- ArrFill, генератор массива, 33
- ArrGen, генератор массива, 33
- ArrRandomInteger, ArrRandom, ArrRandomReal, генераторы массива, 35
- Average, запрос, 62
- Batch, запрос, 83
- BinarySearch, метод массива, 106
- Cartesian, запрос, 77
- Concat, запрос, 68
- Contains, запрос, 62
- ConvertAll, метод массива, 109
- Copy, функция, 95
- CopyTo, метод массива, 96
- Count, запрос, 41, 62
- Cycle, генератор бесконечной последовательности, 49
- DefaultIfEmpty, запрос, 80
- Distinct, запрос, 67
- ElementAt, запрос, 41, 61
- ElementAtOrDefault, запрос, 61
- Except, запрос, 68
- Find, метод массива, 104
- FindAll, метод массива, 104
- FindIndex, метод массива, 104
- FindLast, метод массива, 104
- FindLastIndex, метод массива, 104
- First, запрос, 61
- FirstOrDefault, запрос, 61
- ForEach, запрос, 90
- foreach, оператор цикла, 31
- GroupBy, запрос, 83
- GroupJoin, запрос, 77
- High
  - свойство массива, 29
  - функция, 30

- HSet, генератор HashSet (короткая функция), 89
- in, операция для массива, 106
- IndexMax, метод массива, 104
- IndexMin, метод массива, 104
- IndexOf, метод массива, 104
- int64, целочисленный тип, 63
- Interleave, запрос, 68
- Intersect, запрос, 68
- Iterate, генератор бесконечной последовательности, 48
- Join, запрос, 77
- JoinIntoString, запрос, 63
- Last, запрос, 61
- LastIndexMax, метод массива, 104
- LastIndexMin, метод массива, 104
- LastIndexOf, метод массива, 104
- LastMaxBy, запрос, 63
- LastMinBy, запрос, 63
- LastOrDefault, запрос, 61
- Length
- свойство массива, 29
  - функция, 30
- LLst, генератор LinkedList (короткая функция), 89
- LongCount, запрос, 63
- Low
- свойство массива, 29
  - функция, 30
- Lst, генератор List (короткая функция), 89
- Max, запрос, 62
- MaxBy, запрос, 63
- MaxValue, свойство числовых типов, 46, 65
- Milliseconds, функция, 38
- MillisecondsDelta, функция, 100
- Min, запрос, 62
- MinBy, запрос, 63
- Numerate, запрос, 73
- OrderBy, запрос, 67
- OrderByDescending, запрос, 67
- Pairwise, запрос, 84
- Partition, запрос, 68
- PositiveInfinity, свойство типа real, 46
- Print, Println, запросы, 90
- Print, Println, методы массивов, 32
- Print, Println, процедуры вывода, 9
- PrintDelimDefault, переменная, 32
- Random, функция, 36
- Range, генератор последовательности, 42
- ReadArrInteger, ReadArrReal, ReadArrString, функции ввода массивов, 52
- ReadInteger, ReadReal, ReadChar, ReadString, ReadBoolean, функции ввода, 8
- ReadLnInteger, ReadLnReal, ReadLnChar, ReadLnString, функции ввода, 9
- ReadSeqInteger, ReadSeqReal, ReadSeqString, функции ввода последовательностей, 55
- Repeat, генератор бесконечной последовательности, 48
- Replace, метод массива, 109
- Reverse
- запрос, 56, 68
  - процедура, 57, 110
- Select, запрос, 38, 73
- SelectMany, запрос, 73
- Seq, генератор последовательности (короткая функция), 37
- SeqFill, генератор последовательности, 37
- SeqGen, генератор последовательности, 37
- SeqRandomInteger, SeqRandomReal, генераторы последовательности, 37
- SequenceEqual, запрос, 62
- SeqWhile, генератор последовательности, 45
- SetLength, процедура, 98
- SetPrecision, процедура модуля PT4, 25
- Show, ShowLine, процедура модуля PT4, 24
- Shuffle, метод массива, 109
- Single, запрос, 61
- SingleOrDefault, запрос, 61
- Skip, запрос, 67
- SkipWhile, запрос, 67
- Slice, запрос, 59, 67
- Sort
- метод массива, 109
  - процедура, 110
- Sorted, запрос, 67
- SortedDescending, запрос, 68
- SplitAt, запрос, 68

- Sqr, Sqrt, функции, 19  
SSet, генератор SortedSet (короткая функция), 89  
Step, генератор бесконечной последовательности, 48  
Sum, запрос, 40, 62  
Tabulate, запрос, 73  
Take, запрос, 47, 67  
TakeLast, запрос, 67  
TakeWhile, запрос, 67  
Task, процедура модуля PT4, 18  
ThenBy, запрос, 67  
ThenByDescending, запрос, 68  
ToArray, запрос, 88  
ToDictionary, запрос, 88  
ToHashSet, запрос, 88  
ToLinkedList, запрос, 88  
ToList, запрос, 88  
ToLookup, запрос, 88  
ToSortedSet, запрос, 88  
ToString, метод, 64  
Transform, метод массива, 109  
Union, запрос, 68  
UnzipTuple, запрос, 82  
Where, запрос, 66  
WriteFormat, WriteLineFormat, процедуры вывода, 9  
WriteLines, запрос, 90  
yield sequence, оператор, 50  
yield, оператор, 49  
Zip, запрос, 77  
ZipTuple, запрос, 77  
Вывод типов, 7
- Генераторы  
  Arr (короткая функция), 35, 89  
  ArrFill, 33  
  ArrGen, 33  
  ArrRandomInteger, ArrRandom, ArrRandomReal, 35  
  Cycle, 49  
  HSet (короткая функция), 89  
  Iterate, 48  
  LLst (короткая функция), 89  
  Lst (короткая функция), 89  
  Range, 42  
  Repeat, 48  
  Seq (короткая функция), 37  
  SeqFill, 37  
  SeqGen, 37  
  SeqRandomInteger, SeqRandomReal, 37  
  SeqWhile, 45  
  SSet (короткая функция), 89  
  Step, 48
- Динамические массивы, 28  
  +, операция, 96  
  AdjacentFind, метод, 106  
  BinarySearch, метод, 106  
  ConvertAll, метод, 109  
  CopyTo, метод, 96  
  Find, метод, 104  
  FindAll, метод, 104  
  FindIndex, метод, 104  
  FindLast, метод, 104  
  FindLastIndex, метод, 104  
  High, свойство, 29  
  in, операция, 106  
  IndexMax, метод, 104  
  IndexMin, метод, 104  
  IndexOf, метод, 104  
  LastIndexMax, метод, 104  
  LastIndexMin, метод, 104  
  LastIndexOf, метод, 104  
  Length, свойство, 29  
  Low, свойство, 29  
  Print, Println, методы, 32  
  Replace, метод, 109  
  Shuffle, метод, 109  
  Sort, метод, 109  
  Transform, метод, 109  
  описание и инициализация, 29  
  срезы, 57
- Запросы  
  Aggregate, 63  
  All, 62  
  Any, 62  
  Average, 62  
  Batch, 83  
  Cartesian, 77  
  Concat, 68  
  Contains, 62  
  Count, 41, 62  
  DefaultIfEmpty, 80  
  Distinct, 67  
  ElementAt, 41, 61  
  ElementAtOrDefault, 61  
  Except, 68  
  First, 61

- FirstOrDefault, 61
- ForEach, 90
- GroupBy, 83
- GroupJoin, 77
- Interleave, 68
- Intersect, 68
- Join, 77
- JoinIntoString, 63
- Last, 61
- LastMaxBy, 63
- LastMinBy, 63
- LastOrDefault, 61
- LongCount, 63
- Max, 62
- MaxBy, 63
- Min, 62
- MinBy, 63
- Numerate, 73
- OrderBy, 67
- OrderByDescending, 67
- Pairwise, 84
- Partition, 68
- Print, Println, 90
- Reverse, 56, 68
- Select, 38, 73
- SelectMany, 73
- SequenceEqual, 62
- Single, 61
- SingleOrDefault, 61
- Skip, 67
- SkipWhile, 67
- Slice, 59, 67
- Sorted, 67
- SortedDescending, 68
- SplitAt, 68
- Sum, 40, 62
- Tabulate, 73
- Take, 47, 67
- TakeLast, 67
- TakeWhile, 67
- ThenBy, 67
- ThenByDescending, 68
- ToArray, 88
- ToDictionary, 88
- ToHashSet, 88
- ToLinkedList, 88
- ToList, 88
- ToLookup, 88
- ToSortedSet, 88
- Union, 68
- UnzipTuple, 82
- Where, 66
- WriteLines, 90
- Zip, 77
- ZipTuple, 77
- Захват переменных, 12
- Исключение, 61
- Коллекция, 31
- Короткие функции
  - Arr, 35, 89
  - HSet, 89
  - LLst, 89
  - Lst, 89
  - Seq, 37
  - SSet, 89
- Кортежи, 13
  - метод Add и операция +, 15
  - сравнение, 15
- Кортежное присваивание, 14
- Лямбда-выражения, 11
- Обобщенные функции, 33
- Подпрограммы ввода
  - ReadArrInteger, ReadArrReal, ReadArrString, 52
  - ReadInteger, ReadReal, ReadChar, ReadString, ReadBoolean, функции, 8
  - ReadlnInteger, ReadlnReal, ReadlnChar, ReadlnString, функции, 9
  - ReadSeqInteger, ReadSeqReal, ReadSeqString, 55
- Подпрограммы вывода
  - Print и Println, методы последовательностей, 42
  - Print, Println, методы массивов, 32
  - Print, Println, процедуры, 9
  - WriteFormat, WritelnFormat, процедуры, 9
- Подпрограммы модуля PT4
  - SetPrecision, процедура, 25
  - Show, ShowLine, процедуры, 24
  - Task, процедура, 18
  - Write, Writeln, WriteAll, Print, Println, PrintAll, методы для вывода коллекций, 32
- Последовательности*, 36
- запросы. См. Запросы
- описание, 37

- Режимы выполнения программы (debug и release), 39
- Решения задач
- Array1, 34, 49, 50, 51
  - Array12, 59
  - Array47, 107, 108
  - Array5, 35, 49
  - Array54, 99, 101, 102, 103
  - Array57, 97
  - Array7, 56, 57, 58, 59
  - Array88, 110, 111
  - LingBegin1, 64
  - LingBegin15, 66
  - LingBegin17, 70
  - LingBegin2, 65
  - LingBegin25, 71
  - LingBegin29, 71
  - LingBegin33, 75
  - LingBegin38, 76
  - LingBegin51, 82
  - LingBegin53, 83
  - LingBegin57, 87, 88
  - LingBegin9, 65
  - Proc20, 21
  - Proc4, 23, 24
- Срезы, 57
- Статические массивы, 27
- Структурная эквивалентность типов, 94
- Форматная строка и форматные настройки, 9

## Содержание

Предисловие.....	3
Глава 1. Некоторые расширения Паскаля в языке PascalABC.NET.....	7
1.1. Описание переменных .....	7
1.2. Ввод и вывод данных .....	8
1.3. Лямбда-выражения.....	11
1.4. Кортежи.....	13
Глава 2. Электронный задачник Programming Taskbook.....	16
2.1. Создание заготовки для выбранного задания.....	16
2.2. Окно задачника.....	18
2.3. Решение задачи.....	19
2.4. Отладочные средства задачника.....	24
Глава 3. Массивы и последовательности.....	27
3.1. Статические и динамические массивы в PascalABC.NET .....	27
3.2. Создание массивов и их вывод .....	28
3.3. Функции генерации массивов .....	33
3.4. Последовательности.....	36
3.5. Вывод последовательностей и их генерация. Бесконечные последовательности .....	42
3.6. Дополнение. Генерация последовательностей с помощью конструкции yield.....	49
3.7. Ввод массивов и последовательностей. Инвертирование. Срезы .....	51
Глава 4. Запросы.....	60
4.1. Поэлементные операции, квантификаторы и агрегирование .....	61
4.2. Фильтрация, сортировка, комбинирование и расщепление.....	66
4.3. Проецирование .....	73
4.4. Объединение. Запрос DefaultIfEmpty .....	77
4.5. Группировка.....	83
4.6. Запросы экспортирования и вспомогательные запросы .....	88
Глава 5. Дополнительные средства обработки массивов.....	93
5.1. Особенности присваивания статических и динамических массивов. Копирование и объединение динамических массивов .....	93
5.2. Изменение размеров динамического массива .....	99
5.3. Поиск в динамических массивах .....	104
5.4. Преобразование динамических массивов .....	110
Литература .....	114
Указатель.....	115