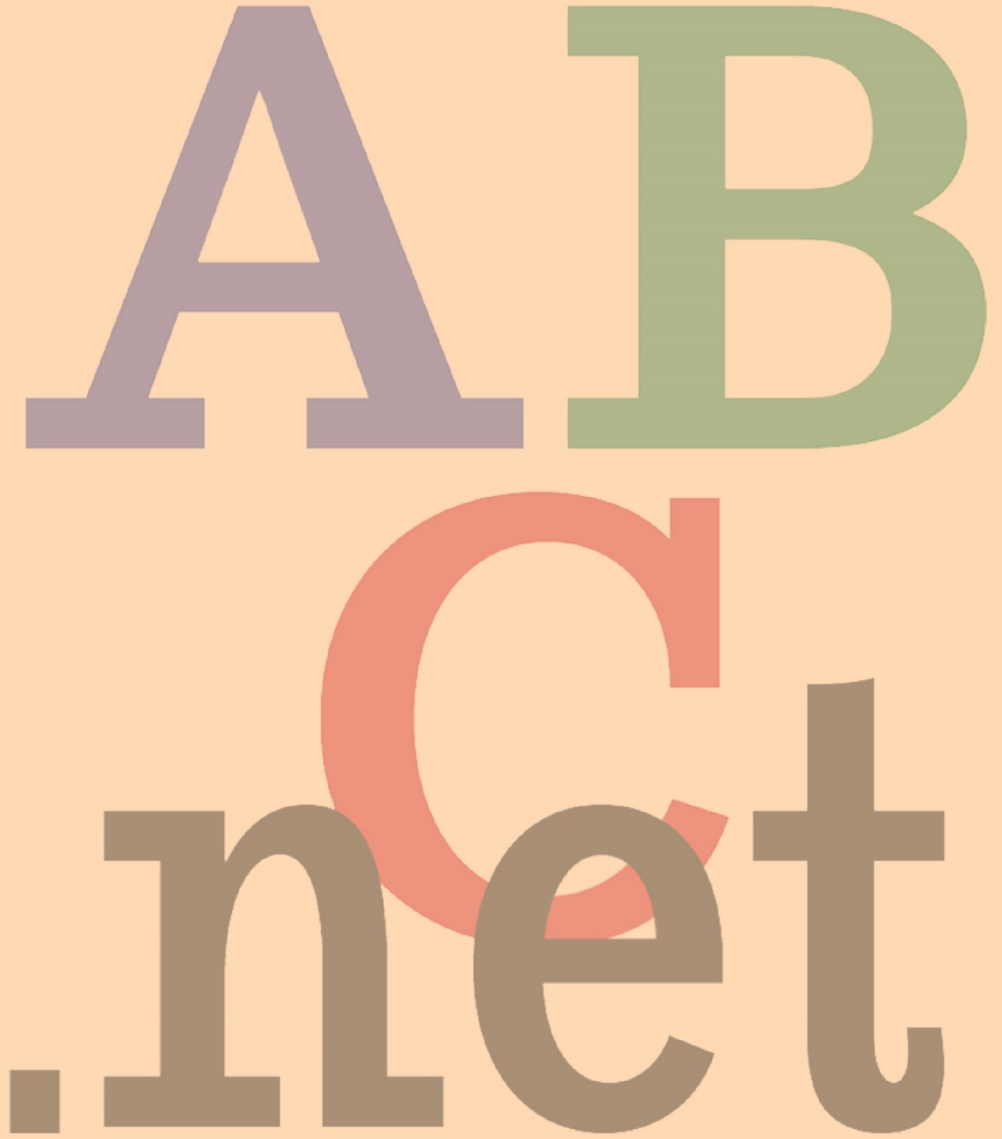
Александр Осипов

PascalABC.NET: выбор школьника
Часть 2



Александр Осипов

PascalABC.NET: выбор школьника
Часть 2

Подпрограммы
Последовательности
Массивы
Чтение программ Turbo/Free Pascal

70 типичных задач:
постановка, математическая модель,
программный код

Ростов-на-Дону
2020

УДК 004.432
ББК 32.973
О741

Рецензенты:

кандидат физико-математических наук,
доцент кафедры алгебры и дискретной математики
Южного федерального университета
С. С. Михалкович

кандидат физико-математических наук,
доцент, заведующий кафедрой информатики, физики и МПИФ
Оренбургского государственного педагогического университета
В. О. Дженжер

Осипов А. В.

О741 PascalABC.NET: выбор школьника. Часть 2. /А. В. Осипов. –
Ростов-на-Дону : – 179 с.

Целевая аудитория книги – школьники и учащиеся иных общеобразовательных учреждений среднего образования. Книга может быть также полезна студентам младших курсов, учителям и преподавателям, интересующимся решением задач в современной версии языка Pascal. Приводится теория и дается решение задач по программированию из школьного курса информатики с максимальным использованием возможностей PascalABC.NET. Подборка задач позволяет использовать книгу в качестве ныне популярного «решебника», но главная цель – научиться писать современный короткий, понятный и эффективный код.

УДК 004.432
ББК 32.973

Оглавление

Предисловие разработчика PascalABC.NET

Книга Осипова А.В. «PascalABC.NET: выбор школьника. Часть 2» продолжает серию книг по PascalABC.NET для школьников. Эта книга затрагивает такие важные темы как функции, последовательности, лямбды, одномерные и двумерные массивы.

Следует сразу отметить, что книга получилась не для начинающих. Здесь используется практически весь арсенал стандартных методов и конструкций PascalABC.NET в этих темах. Читать такой материал – сложно. От обилия стандартных методов кружится голова.

Несомненно, в этом есть цель. Читатель, пробившийся через такое чтение, поймёт, что есть небольшое количество стандартных алгоритмов, которые в сочетании с лямбдами позволяют решить практически любую задачу средней сложности коротко и понятно.

Глава о последовательностях и лямбда-выражениях – по существу новый материал для школьников. Это распространённые примитивы программирования, которые есть сейчас в подавляющем большинстве языков и пришли из функционального программирования.

Глава о массивах позволяет показать, как, используя многочисленные стандартные методы массивов, решать сложные задачи и записывать их решение в простой и понятной форме.

Глава о матрицах показывает мощь стандартной библиотеки PascalABC.NET, где с помощью небольшого числа стандартных методов и конструкций решаются опять-таки весьма сложные задачи, требующие груды кода при классическом подходе.

Охват материала значительно больше, чем требуется для сдачи ЕГЭ. Материал книги позволяет решать олимпиадные задачи – ясно и компактно. Но олимпиадные задачи – особая область, для них нужна отдельная книга.

Иногда изложение похоже на справочник – стандартных методов действительно очень много. Однако, и задач решено также много – нужен только пытливый ум разбираться в их решении.

Я бы рекомендовал книгу для старших школьников и студентов младших курсов – именно для знакомства с арсеналом современного программирования.

Как и первая, данная книга – замечательная, даёт много пищи для пытливого ума. Книга может быть использована школьными учителями и работниками дополнительного образования для обновления методик преподавания программирования.

Руководитель проекта PascalABC.NET,
директор Воскресной компьютерной школы
при мехмате Южного федерального университета,
Михалкович Станислав Станиславович

Предисловие рецензента

Вышла вторая часть книги А. В. Осипова «PascalABC.NET: выбор школьника». Она содержит две традиционных для учебников по программированию главы (о подпрограммах и двумерных массивах), а также одну нетрадиционную (о последовательностях). Использование последовательностей радикально меняет не только стиль программирования, но и стиль мышления программиста. Рассуждать в терминах последовательностей многим гораздо проще, чем в терминах циклических действий. Правда, для уверенного использования последовательностей необходимы дополнительные знания. Например, нужно владеть аппаратом лямбда-выражений, которым посвящён один из разделов второй части книги. Введение в программирование новых сущностей может показаться излишним: в Турбо Паскале такого нет, на ЕГЭ не требуют — так зачем тогда? Автор этой книги, как и авторы языка, хотят показать, что времена изменились, и современное программирование опирается на использование новых примитивов: кортежей, последовательностей, лямбд, а также удобных встроенных структур данных, без которых мы навсегда застрянем в прошлом веке.

Описание работы с массивами тоже нельзя назвать традиционным. PascalABC.NET включает мощные новые средства работы с матрицами, которые рушат старые подходы. Теперь матрицу можно рассматривать и как набор строк, и как набор столбцов и даже просто как последовательность элементов. Многие школьные задачи становятся в этом случае просто примерами на использование методов матриц.

Сегодня многие учителя высказывают опасения насчёт использования новых средств программирования. Во-первых, сомнения связаны с тем, что новый стиль может помешать при сдаче ЕГЭ по информатике и ИКТ. Действительно, в КИМ по информатике фрагменты кода приводятся на старом (базовом) Паскале. Однако, как показано в Приложении к части 2, «перевод» на современный язык не представляет никаких трудностей. А в задачах, требующих написания программ, можно, указав версию компилятора, писать на любом языке. Кстати, автор планирует выпустить своеобразный «решебник» задач

ЕГЭ на PascalABC.NET. Будет полезным сравнить эти решения с «эталонными».

Во-вторых, часто указывается на то, что при использовании новых средств ученик разучится решать задачи «по-старому». Таких переходных моментов было много, даже на моей памяти. Помнится, многие очень переживали, что языки высокого уровня напрочь отбивают желание программировать на ассемблере. А как настоящему программисту прожить без ассемблера? Сейчас это немного смешно. Новый стиль — это не прихоть разработчиков PascalABC.NET. Это реальный мейнстрим, это используется во всех современных языках. Изучение новых языковых средств необходимо для того, чтобы сегодня быть программистом.

Думаю, что читателю захочется отдельно поблагодарить автора за огромное количество содержательных задач, приведённых в тексте по каждой теме. Разбор примеров является одним из важнейших способов изучения языка. Автор демонстрирует современный стиль написания программ, короткий, ёмкий и ясный.

Мы ждём третьей части, в которой будут описаны не вполне традиционные для школы типы данных: стеки, очереди, динамические списки и многое другое!

Дженжер В. О., к. ф.-м. н., доцент, заведующий кафедрой информатики, физики и МПИФ Оренбургского государственного педагогического университета

От автора

Melius non incipient, quam desinent.
Лучше не начинать, чем останавливаться на полпути. (лат.)

В середине марта 2020 года в издательстве Южного федерального университета небольшим тиражом вышла первая часть книги. Мы планировали ознакомить с ней в первую очередь школьных учителей и раздать экземпляры школьникам, пришедшим на мехмат ЮФУ в день открытых дверей. Нам были нужны отзывы читательской аудитории, чтобы понять, надо ли менять что-то хотя бы во второй части или можно продолжать, сохранив прежний стиль. Но известные события привели к повсеместной отмене публичных мероприятий. Я не мог себе позволить ожидать, когда и как изменится ситуация, чтобы можно было ознакомить потенциальных читателей с первой частью книги. В этих условиях единственным разумным решением было писать следующую часть.

Фактически, обе части – одна книга, и это ощущается буквально во всем: даже нумерация глав второй части продолжает нумерацию первой. Продолжено рассмотрение средств языка PascalABC.NET и решений задач. Изучив вторую часть книги, читатель получит возможность эффективно решать практически любые задачи из школьной информатики, связанные с программированием на алгоритмическом языке. Все, что останется неясным, вы можете попытаться найти в книге «PascalABC.NET: Введение в современное программирование», опубликованной 7 октября 2019 года на официальном Интернет-сайте PascalABC.NET (http://pascalabc.net).

Объем книги не позволил посвятить отдельные главы множествам, символам и строкам, работе с файлами, пользовательским типам данных, стандартным контейнерным классам, модулям и библиотекам, обработке ошибок и объектно-ориентированному программированию. Часть этого материала входит в школьную программу, но в КИМ ЕГЭ (и, тем более, ОГЭ) задач на этот материал пока что нет. Вполне возможно, что через какое-то время будет принято решение

выпустить третью часть, в которой будут рассмотрены перечисленные темы.

Отдельный вопрос – компьютерная графика. О ней не сказано ничего. Графика – не компонент алгоритмического языка PascalABC.NET. В современных универсальных языках программирования графика реализуется через подключаемые библиотеки. Но я не исключаю, что рано или поздно может появиться еще одна часть этой книги, возможно другого автора, посвященная графическим библиотекам PascalABC.NET.

Как и в работе над первой частью книги, я постоянно консультировался с руководителем проекта PascalABC.NET Станиславом Станиславовичем Михалковичем, по-прежнему обеспечивающим технический надзор за точностью излагаемого материала. И снова Вадим Олегович Дженжер тщательно вычитывал тексты, находя опечатки и огрехи. Хочу еще раз, публично, высказать этим двум людям огромную благодарность за большую проделанную работу и проявленное ими терпение.

А.Осипов

Глава 5

Подпрограммы

В этой главе…

Основные понятия
Процедуры
Функции
Рекурсия
Лямбда-выражения
Понятие о классах
Примеры решения задач

Divide et impera
Разделяй и властвуй (лат.)

При решении задач вы неоднократно сталкивались с ситуацией, кода требовалось несколько раз выполнить одни и те же действия. Частично эту проблему решает использование циклов. Но как быть, если один и тот же участок кода нужно повторить в другом месте программы? Как поступить, если нужно произвести вычисления для группы изменяемых данных (например, вычислить длины сторон треугольника, заданного координатами вершин)? Вот тут на помощь и приходят подпрограммы.

Подпрограмма – именованный или идентифицированный иным образом фрагмент программного кода, к которому можно многократно обращаться. Подпрограмма содержит описание определённого набора действий.

Обращение к подпрограмме называется ее ***вызовом***. Чаще всего для вызова подпрограммы используется ее имя. Место в коде программы, откуда вызывается подпрограмма, называется ***точкой вызова***, а соответствующая программная единица – ***вызывающей***. Сама же подпрограмма при этом считается ***вызываемой***.

Функция – разновидность подпрограммы, возвращающей некоторое значение, которое затем используется в выражении.

Вы давно знакомы с функциями – Sqr, Sqrt, Abs и т.п. – все это стандартные функции; их PascalABC.NET уже «знает». Но также можно писать собственные функции.

Процедура - подпрограмма, не возвращающая значения.

Иными словами, процедуры – это подпрограммы, которые не являются функциями. Например, процедуры Write и Println.

Подпрограммы могут также вызывать сами себя. В этом случае они называются ***рекурсивными***. Наиболее популярны рекурсивные функции – они получаются на основе рекуррентных соотношений.

5.1. Параметры подпрограмм

Для того, чтобы обмениваться данными с подпрограммами, служит механизм ***параметров***. Подпрограмма должна уметь получать данные от вызывающей программной единицы, а также, при необходимости, возвращать ей измененные данные. Если обмен данными не требуется, параметры не нужны.

> ***Параметры подпрограммы*** – переменные в ее описании, служащие для обмена данными с другими программными единицами.

Для передачи данных подпрограмме в точке вызова указывается ее имя, за которым в круглых скобках перечисляются через запятую выражения, значения которых будут вычислены и переданы. Такие параметры называются ***аргументами*** или ***фактическими параметрами***. В частном случае аргумент может быть просто именем переменной.

В заголовке, описывающем подпрограмму, также указывается список параметров. Это так называемые ***формальные параметры***. Их формальность в том, что при вызове подпрограммы значения формальных параметров замещаются значением фактических и уже с этими значениями выполняется код тела подпрограммы.

Передавать параметры можно двумя способами.

Первый способ называется ***передачей по значению***. Значение фактического параметра копируется в формальный, после чего связь фактического и формального параметра разрывается. На практике это означает, что подпрограмма может что угодно делать с формальным параметром: значение фактического при этом не изменится. Я написал объявление (фактический параметр), снял с него копию (передал по значению в формальный параметр) и повесил где-то на доску объявлений. Я не знаю, что дальше будет с этой копией и не могу на это повлиять. Для запоминания: «передал – и забыл».

Второй способ называется ***передачей по ссылке***. Мы не будем сейчас разбираться, что такое ссылка, кому интересно – может почитать об этом в моей книге «PascalABC.NET: Введение в современное программирование», опубликованной на официальном Интернет-сайте PascalABC.NET. Важно понять, что никакого копирования данных из фактического параметра в формальный не происходит. Вместо этого, подпрограмма получает непосредственный доступ к фактическому параметру, когда обращается к формальному. Фактически, на время выполнения, формальный и фактический параметр становятся синонимами, «ссылаются» на одни и те же данные. Передача по ссылке позволяет подпрограмме вносить изменения в фактический параметр.

При передаче по ссылке фактический параметр не может быть выражением – только именем переменной.

Традиционно, функция не должна менять значений аргументов, она лишь возвращает вычисленный результат. Процедура не возвращает значений, поэтому изменять значения аргументов для нее совершенно нормально. Но это лишь традиция, не более того.

Формальные параметры, вызываемые по значению, имеют следующее описание:

```
имя: тип
имя, имя, …: тип
```

Формальные параметры, вызываемые по ссылке, описываются несколько иначе:

```
var имя: тип
var имя, имя, …: тип
```

Формальные параметры могут иметь значения, которые присваиваются по умолчанию:

```
имя := значение
имя: тип := значение
```

Такие формальные параметры должны указываться только после параметров, не получающих начального значения. Умолчание позво-

ляет списку фактических параметров быть короче списка формальных параметров.

Параметры вовсе не обязательно должны быть простыми переменными. Например, они могут быть последовательностями и массивами, о чем говорится в главе 6.

5.2. Процедуры

Процедура состоит из заголовка процедуры, начинающегося ключевым словом **procedure** и тела процедуры. В заголовке указывается имя процедуры и перечисляются ее формальные параметры, если они есть. В теле находится код, реализующий некоторый алгоритм.

```
procedure Имя(параметр1; параметр2; …);
begin
  тело процедуры
end;
```

Процедура всегда должна быть записана выше по тексту, чем основная программа, поскольку в языке Pascal описание всегда предшествует обращению.

Рассмотрим процедуру, находящую максимум, минимум и среднее значение для N случайных целых чисел, принадлежащих отрезку [-99;99].

```
procedure MyFirst(n: integer; var mn, mx: integer; var m: real);
begin
  (mn, mx) := (100, -100);
  var s := 0;
  loop n do
  begin
    var k := Random(-99, 99);
    s += k;
    if k < mn then
      mn := k
    else if k > mx then
      mx := k
  end;
  m := s / n
end;
```

Кроме заголовка здесь для вас нет ничего нового. Параметр *n* принимаем по значению, остальные – по ссылке. Поскольку параметры уже описаны в заголовке, в теле процедуры их описывать не надо (более того, это будет считаться ошибкой). А теперь вызовем эту процедуру для различных значений *n*.

```
begin
  var Макс, Мин: integer;
  var Сред: real;
  MyFirst(10, Мин, Макс, Сред); // вызов процедуры
  Writeln('min = ', Мин, ', max = ', Макс, ', среднее = ', Сред);
  MyFirst(100, Мин, Макс, Сред); // вызов процедуры
  Writeln('min = ', Мин, ', max = ', Макс, ', среднее = ', Сред);
  MyFirst(1000, Мин, Макс, Сред); // вызов процедуры
  Writeln('min = ', Мин, ', max = ', Макс, ', среднее = ', Сред);
  MyFirst(1000000, Мин, Макс, Сред); // вызов процедуры
  Writeln('min = ', Мин, ', max = ', Макс, ', среднее = ', Сред);
end.
```

```
min = -89, max = 98, среднее = 13.5
min = -97, max = 99, среднее = -4.08
min = -99, max = 99, среднее = 0.506
min = -99, max = 99, среднее = 0.048458
```

Как видите, с ростом *n* минимум и максимум все ближе подбираются к границам отрезка, а среднее значение – к нулю. При вызове MyFirst значение первого параметра копируется в формальный параметр n, параметр mn понимается процедурой как Мин, mx – как Макс, а m – как Сред.

Если процедура вызывается без параметров, то скобки после ее имени также не указывают.

5.3. Функции

Функция, в отличие от процедуры, возвращает некоторое значение. Эта особенность функций отражается в их описании.

```
function Имя(параметр1; параметр2; …): тип;
begin
  тело функции, обязательно с переменной Result
end;
```

При описании функции нужно указать тип возвращаемого ею значения. В теле функции всегда присутствует переменная *Result*, которая не описывается, но обязательно должна получить некоторое значение, которое и будет считаться результатом вычисления функции. Как и процедура, функция может принимать параметры по значению и по ссылке. Функция может не иметь параметров.

В качестве примера рассмотрим реализацию функции Sqrt3(x), вычисляющей значение кубического корня аргумента.

```
function Sqrt3(x: real): real;
begin
  Result := Sign(x) * Abs(x) ** (1 / 3)
end;
```

Здесь абсолютная величина аргумента возводится в степень ⅓ и к результату приписывается знак аргумента.

Вызывается функция как и процедура: путем указания ее имени и, при необходимости, фактических параметров.

```
begin
  Println(Sqrt3(2.0));
  Println(Sqrt3(8.0));
  Println(Sqrt3(-8.0));
  Println(Sqrt3(153.642));
  Println(Sqrt3(4913));
end.
```

```
1.25992104989487
2
-2
5.35595168436603
17
```

Обратите внимание на последний вызов. В качестве аргумента указано значение 4913, а не 4913.0, т.е. аргумент имеет тип **integer**, а не **real**, как в описании функции. Но еще в главе 1 мы говорили о том, что тип **integer** при необходимости автоматически приводится компилятором к типу **real** (но не наоборот). Это и произошло. А можно написать что-то типа Sqrt3(a > 0) ? Нет, потому что a > 0 – это логическое выражение, а тип **boolean** не приводится к **real** и компилятор зафиксирует ошибку.

5.4. Упрощенный синтаксис

Если тело процедуры или функции состоит из одного оператора, PascalABC.NET позволяет использовать при описании упрощенный синтаксис без **begin** и **end**.

```
procedure Имя(параметр1; параметр2; …) := оператор;
function Имя(параметр1; параметр2; …): тип := выражение;
function Имя(параметр1; параметр2; …) := выражение;
```

Упрощенный синтаксис делает программу более компактной. Мы могли бы записать функцию Sqrt3(x) и так:

```
function Sqrt3(x: real) := Sign(x) * Abs(x) ** (1 / 3);
```

Тип здесь выводится автоматически из типа выражения: операция возведения в степень всегда дает тип **real**. Но при желании вы можете указать тип для тех, кто будет читать вашу программу и незнаком с правилами автовыведения типов в PascalABC.NET.

5.5. Рекурсия

Вспомните, что факториал натурального числа n вычисляется как произведение всех натуральных чисел от 1 до n включительно. Факториал единицы равен единице и записывается это так: n! = 1. Мы можем найти факториал, используя рекуррентную формулу

$$n! = \begin{cases} 1, \text{ при } n = 1 \\ n \cdot (n-1)!, \text{ при } n > 1 \end{cases}$$

Задачу вычисления факториала мы уже решали в главе 4 (задача 4.2), а теперь попробуем использовать рекурсию.

Рекурсивная функция обращается сама к себе. Но поскольку это не может продолжаться бесконечно, требуется определить условие выхода из процесса рекурсии. В нашем случае это равенство единице аргумента. Посмотрим, как вычисляется 5!

5! = 5·4! = 5·4·3! = 5·4·3·2! = 5·4·3·2·1! = 5·4·3·2·1 = 120

Если мы напишем рекурсивную функцию Fact(n), то при вызове Fact(5) она обратится сама к себе 4 раза. Количество таких обращений называется ***глубиной рекурсии***. Если глубина рекурсии большая, ресурсов компьютера может не хватить, что необходимо учитывать при написании рекурсивных функций. Рекурсия может быть сведена к итерации, но к сожалению, этот процесс не всегда прост.

```
function Fact(n: integer): real := n = 1 ? 1.0 : n * Fact(n - 1);

begin
  Print(Fact(5))
end.
```

```
120
```

Возможна и иная запись функции

```
function Fact(n: integer): real := if n = 1 then 1.0 else n * Fact(n - 1);
```

Для нужд, в частности, комбинаторики факториал дополняется возможностью принимать аргумент, равный нулю. Принято считать по определению, что 0! = 1.

```
function Fact(n: integer): real := if n <= 1 then 1.0 else n * Fact(n - 1);
```

Программирование рекурсивных функций отличается простотой, а код – наглядностью. Плата за это – расход ресурсов компьютера. Рекурсия может быть достаточно сложной, например, параметры рекурсивной функции могут иметь, в свою очередь, рекурсивное определение. Типичный представитель – функция Аккермана A(n, m). Она растёт очень быстро, например, число A(4, 4) уже настолько велико, что количество цифр в нем во много раз превосходит количество атомов в наблюдаемой части вселенной. А вот значение A(3,4) можно попробовать вычислить и мы это сделаем в задаче 5.6.

5.6. Перегрузка имен подпрограмм

Слово «перегрузка» здесь никак не связано с увеличением ускорения свыше земного g. Правильнее, наверно, было бы использовать термин «перезагрузка», но что поделать?

Потребность в перегрузке имен возникает, когда подпрограмма может иметь различный набор параметров. Пусть нам нужна функция Snp(a, b), умеющая находить площадь прямоугольника со сторонами a, b и возвращать ее значение с типом **integer**, если оба параметра типа **integer** и с типом **real** в прочих случаях. Как быть – называть функции разными именами? Перегрузка позволяет обойтись одним.

```
function Snp(a, b: integer) := a * b;

function Snp(a, b: real) := a * b;

begin
  var a := Snp(3, 7);
  var b := Snp(2.5, 5);
  Print(a, b)
end.
```

```
21 12.5
```

Чем отличаются описания функций? Типами параметров. Эти типы и являются основой для перегрузки имен. Если типы аргументов полностью совпадают с одним из описаний, именно это описание будет выбрано. Если точного совпадения нет – компилятор попробует выполнить приведение типа аргументов. Поэтому в примере в первом случае вызывается первая функция, во втором – вторая.

Тип результата в перегрузке имен не участвует.

5.7. Примеры решения задач

По мере возможности, в этом разделе будет использована классификация задач из первой части книги.

5.7.1. Вычисления по известным формулам

Задача 5.1

Вычислить и вывести с 12 знаками после запятой значение x, найденное по формуле

$$x = \frac{5+\sqrt{5}}{\sqrt{7}+7} + \frac{12+\sqrt{12}}{\sqrt{8}+8} + \frac{31+\sqrt{31}}{\sqrt{2}+2}$$

Изучив формулу, решаем написать функцию F(n) = n + √n, что должно сделать программу короче.

```
function F(n: integer) := n + Sqrt(n);

begin
  Write(F(5) / F(7) + F(12) / F(8) + F(31) / F(2):0:12)
end.
```

```
12.888734395799
```

Можно было определить функцию с двумя параметрами, сразу получая слагаемые:

```
function F(m, n: integer) := (m + Sqrt(m)) / (n + Sqrt(n));

begin
  Write(F(5, 7) + F(12, 8) + F(31, 2):0:12)
end.
```

Тут программист сам решает, как ему удобнее. Потому что еще и так можно было написать:

```
function P(n: integer) := n + Sqrt(n);

function F(m, n: integer) := P(m) / P(n);
begin
  Write(F(5, 7) + F(12, 8) + F(31, 2):0:12)
end.
```

А можно описать сначала F, а потом P? Нет. Уже не раз было сказано: пока не описали, обращаться нельзя. А тут из F вызывается P.

Задача 5.2
Вычислить и вывести с 7 знаками после запятой значение x, найденное по формуле

$$x = \frac{1+\sin 4}{4+\sin 1} + \frac{7+\sin 5}{5+\sin 7} + \frac{3+\sin 2}{2+\sin 3}$$

Здесь получаем слагаемое, как функцию двух аргументов.

```
function F(m, n: integer) := (m + Sin(n)) / (n + Sin(m));

begin
  Write(F(1, 4) + F(7, 5) + F(3, 2):0:7)
end.
```

```
2.9439475
```

5.7.2. Вычисления по формулам из геометрии

Задача 5.3
Вершины треугольника заданы случайными целочисленными координатами (x, y), принадлежащими отрезку [-500; 500]. Найти максимальный периметр среди десяти треугольников.

Периметр треугольника определяется суммой его сторон. Длина стороны треугольника вычисляется как длина отрезка, начало и конец которого заданы координатами (формула П3.1). Имеет смысл написать функцию *L* с четырьмя аргументами – координатами точек, возвращающую длину отрезка. Для нахождения периметра ее придется вызывать трижды (стороны AB, BC, AC) и каждая пара координат будет повторена два раза. Поэтому можно ввести функцию *P* для нахождения периметра, передав ей шесть параметров – координаты вершин треугольника. Для каждого треугольника надо случайным образом получить координаты вершин, вывести их значения и определить периметр. Всю эту работу будет делать функция *Triangle*. Ей не надо передавать параметров, а возвращать она будет значение периметра.

```
function L(Ax, Ay, Bx, By: integer) :=
    Sqrt(Sqr(Bx - Ax) + Sqr(By - Ay));


function P(Ax, Ay, Bx, By, Cx, Cy: integer) :=
   L(Ax, Ay, Bx, By) + L(Bx, By, Cx, Cy) + L(Ax, Ay, Cx, Cy);
function Triangle: real; // функция без параметров
begin
   var Ax, Ay, Bx, By, Cx, Cy: integer;
   (Ax, Ay) := Random2(-500,500);
   (Bx, By) := Random2(-500,500);
   (Cx, Cy) := Random2(-500,500);
   Result := P(Ax, Ay, Bx, By, Cx, Cy); // вызов P
   Writeln('A(', Ax, ',', Ay, '), B(', Bx, ',', By,
      '), C(', Cx, ',', Cy, '), P=', Result)
end;

begin
  var Pmax := 0.0;
  loop 10 do
  begin
    var Pt := Triangle; // вызов функции без параметров
    if Pt > Pmax then
      Pmax := Pt
  end;
  Print('Максимальный периметр', Pmax)
end.
```

```
A(-381,352), B(77,343), C(330,451), P=1451.03499599004
A(-328,-317), B(-89,22), C(163,443), P=1810.24710610173
A(276,-215), B(-298,-267), C(-274,-327), P=1202.26037039865
A(-105,-354), B(-492,451), C(-468,-482), P=2211.40826097886
A(-203,-301), B(16,368), C(-426,-216), P=1674.99059803584
A(58,479), B(213,-477), C(148,-6), P=1937.2276900267
A(-354,-384), B(273,393), C(-438,-456), P=2216.45653626702
A(421,-33), B(202,191), C(70,-21), P=914.209184414368
A(-339,-49), B(-120,-386), C(418,-255), P=1740.15571343101
A(45,-331), B(319,59), C(46,131), P=1220.96582516904
Максимальный периметр 2216.45653626702
```

Если не пользоваться подпрограммами, пришлось бы поместить внутрь цикла тело функции *Triangle*, в него вставить тело функции *P*, которая потребует трех вставок тела функции *L*. Использование функций позволило структурировать программу и сделать ее более наглядной. Если понадобится вместо треугольников работать с

четырехугольниками, легко найти места, в которых потребуется скорректировать код.

Задача 5.4

Реализовать и протестировать процедуру Triangle, выполняющую следующие операции: ввод с клавиатуры трех вещественных чисел, обозначающих длины отрезков, проверку того, что из них можно составить треугольник, вывод площади полученного треугольника или нуля, если треугольник составить нельзя. Использовать функции.

```
function ТреугольникВозможен(a, b, c: real) :=
(a + b > c) and (a + c > b) and (b + c > a);

function Geron(a, b, c: real): real;
begin
  var p := (a + b + c) / 2;
  Result := Sqrt(p * (p - a) * (p - b) * (p - c))
end;

procedure Triangle;
begin
  var (a, b, c) := ReadReal3('Введите три числа:');
  Print('Площадь треугольника равна');
  if ТреугольникВозможен(a, b, c) then
    Print(Geron(a, b, c))
  else
    Print(0)
end;

begin
  Triangle
end.
```

```
Введите три числа: 27 31.8 21.63
Площадь треугольника равна 288.294238456389
```

5.7.3. Задачи на рекурсию

Задача 5.5

Написать рекурсивную функцию получения ***цифрового корня*** *натурального числа. Цифровой корень данного числа получается следующим образом. Если сложить все цифры этого числа, затем все цифры найденной суммы и повторять этот процесс, то в результате получится однозначное число (цифра), которая и называется цифровым корнем данного числа.*

Программировать рекурсивные функции – одно удовольствие! Читаем определение функции и пишем код прямо по нему.

```
function DR(n: int64): integer;
begin
  if n > 9 then
  begin
    var s := 0;
    while n > 0 do
    begin
      s += n mod 10;
      n := n div 10
    end;
    n := s
  end;
  Result := n < 10 ? n : DR(n)
end;

begin
  Print(DR(1553457985643424))
end.
```

```
3
```

Основную часть функции занимает получение цифр числа с одновременным накоплением их суммы. Рассмотрим еще один вариант решения задачи.

```
function СуммаЦифр(n: int64): integer;
begin
  Result := 0;
  while n > 0 do
  begin
    Result += n mod 10;
    n := n div 10
  end
end;

function ЦифровойКорень(n: int64): integer;
begin
  if n > 9 then
    n := СуммаЦифр(n);
  Result := if n < 10 then n else СуммаЦифр(n)
end;

begin
  Print(ЦифровойКорень(1553457985643424))
end.
```

Здесь вынесение получения суммы цифр числа в отдельную функцию в сочетании с нормальными именами на русском языке сделало реализованный алгоритм существенно нагляднее.

Задача 5.6

Функция Аккермана A(n, m) представляет пример сложной рекурсии. У нее рекурсивное описание и рекурсивное определение параметров.

$$A(n,m)=\begin{cases} m+1, & \text{если } n=0, \\ A(n-1,1), & \text{если } m>0, n\neq 0, \\ A(n-1,A(n,m-1)), & \text{если } n>0, m>0 \end{cases}$$

```
function Akk(n, m: integer): integer :=
    if n = 0 then
      m + 1
    else if m = 0 then
      Akk(n - 1, 1)
    else
      Akk(n - 1, Akk(n, m - 1));
```

```
begin
  Print(Akk(3, 4))
end.
```

125

Если вы считаете, что возможность определять и использовать рекурсивные функции является некоторым излишеством, попытайтесь запрограммировать эту функцию без рекурсии.

5.8. Область видимости

Объекты, описанные в некотором блоке, компилятор «видит» только в нем и во всех вложенных в него блоках. В частности, это означает, что все объекты определенные внутри подпрограмм, вызывающая программа не видит. Это позволяет разрабатывать подпрограммы, не задумываясь о возможном совпадении имен, которое могло бы породить проблемы.

PascalABC.NET имеет одну интересную особенность. Если внутри программной единицы описать переменную, а ниже по тексту использовать внутриблочную переменную с таким же именем, компилятор зафиксирует ошибку. Но если описать переменную после блока, это не считается ошибкой. Параметр цикла также считается внутриблочной переменной.

Ошибка	Разрешено

`var i := 0;`
`for var i := 1 to 5 do`
`  Print(i);`
	`for var i := 1 to 5 do`
	`  Print(i);`
	`var i := 0;`

Может возникнуть вопрос, как вызывающая программа видит подпрограмму, если подпрограмма описана вне вызывающей программы. Ответ очень прост: описание подпрограмм является частью кода программы, именуемом разделом подпрограмм. А сам раздел подпрограмм помещается текстуально выше кода основной программы.

5.9. Лямбда-выражения

Вот мы и подошли к материалу, который пока что в школьной информатике, как правило, не рассматривают. Мы тоже на будем подробно и научно рассматривать эти «лямбды», а обратимся к практике их использования.

Лямбда-выражения (или просто лямбды) – термин ***функционального программирования***. Они с успехом применяются вместо уже привычных вам процедур и функций. Собственно, они и есть процедуры или функции, только безымянные. Вернитесь к определению подпрограммы на первой странице главы 5. Та самая фраза «или идентифицированный иным образом фрагмент программного кода» как раз и подразумевает лямбды.

> Лямбда-выражение представляет собой некоторое безымянное выражение, отражающее функциональную зависимость. Упрощенно, на его основе компилятор строит функцию, некоторым образом идентифицирует ее и подменяет этим идентификатором лямбда-выражение.

В коде программы лямбда-выражения легко найти по характерной паре символов ->. Это операция, которую называют по-разному, но мне нравится фраза «переходит в …». Вот пример:

```
x -> 3 * x * x - 6.3 * x + Sqrt(x)
```

Данное лямбда-выражение читается как «x переходит в $3x^2+6.3x+\sqrt{x}$». По сути, это запись функции с параметром x, которая возвращает значение выражения $3x^2+6.3x+\sqrt{x}$. Помните, в комедии «Бриллиантовая рука»: – Брюки превращаются … в элегантные шорты. Если бы там знали о лямбдах, могли сказать «переходят в …» вместо «превращаются».

```
(x, y) -> if x > y then x else y;
```

А это лямбда-функция с двумя параметрами и возвращает она максимальный из них.

```
x -> begin Write('Значение равно ', x:0:5) end;
```

Эта лямбда не возвращает значения. Возможно вы уже поняли, что так записываются лямбда-процедуры.

Чем интересы лямбды? Во-первых, их не надо заранее описывать вне вызывающей программы как отдельные функции и процедуры, а можно записывать прямо в коде программы по мере надобности. Во-вторых, как будет показано в следующих главах, лямбды очень сильно упрощают программирование многих задач, являясь параметрами (да-да, именно параметрами) подпрограмм. А еще, лямбду можно присвоить переменной и с этого момента обращаться к ней, как к именованной.

Если вы внимательно знакомились с материалом, должен был возникнуть вопрос: а как же типы? В лямбде x -> x * x какой тип имеет x? Целый, вещественный, логический? В PascalABC.NET для описания типов в лямбда выражениях существует свой синтаксис.

```
begin
  var x: integer-> integer := t -> t * t;
  Print(x(7)); // 49
  x := t -> t * t * t;
  Print(x(7)); // 343
  x := t -> 3 * t - 1;
  Print(x(7)) // 20
end.
```

Посмотрите на описание. Какой тип имеет переменная *x*? Может быть, вам это покажется странным, но ее тип **integer** -> **integer** ! Т.е. это лямбда-функция с параметром типа **integer**, возвращающая результат типа **integer**. В остальной части программы функция x переопределяется, поэтому один и тот же вызов x(7) порождает разные результаты.

Можно ли было в программе написать x := t -> t ** 3 ? Нет! Операция ** возвращает тип **real**, а в соответствии с описанием должен быть тип **integer**. Поэтому при компиляции будет выдана ошибка «Нельзя преобразовать тип real к integer».

Но если сделать описание

```
var x: integer -> real := t -> t * t;
```

в правой части лямбда-выражений можно указывать любые арифметические выражения, приводимые к **real**.

Задача 5.7
Написать процедуру, выводящую таблицу значений произвольной функции одного аргумента на отрезке [a;b] с шагом h.

```
procedure Tab(a, b, h: real; f: real -> real);
begin
  var x := a;
  while x <= b + h / 2 do
  begin
    Writeln(x:10:3, f(x):20:12);
    x += h
  end
end;

begin
  Tab(-5, 3, 1, x -> x * x);
  Writeln;
  Tab(-0.9 * Pi, Pi, Pi / 8, x -> Sqr(Sin(x)) + Sqrt(Abs(x)))
end.
```

В процедуре *Tab* последним параметром определена лямбда-функция. В основной программе указаны два вызова *Tab*, где конкретизируются табулируемые функции. Вот такие они – лямбды!

PascalABC.NET позволяет «для красоты» вместо -> использовать символ → (получается при зажатой клавише Alt набором числа 26 на малой цифровой клавиатуре).

5.10. Понятие о классах

Классы – основа объектно-ориентированного программирования (ООП), которое не рассматривается в этой книге. Но, поскольку язык PascalABC.NET насквозь «пропитан» ООП, базовую информацию дать необходимо.

В ООП объекты программист видит именованными элементами, созданными на основе классов. Класс – абстрактное описание наподобие чертежа, по которому конструируется конкретный объект. Класс содержит в себе описания данных и средств управления этими данными. Данные в классе называются ***полями***, а средства для манипуляции полями называются ***методами***. Поле – это некоторый аналог переменной, а метод можно считать подпрограммой. Объект создается путем вызова метода, называемого ***конструктором***.

При обращении к полям и методам объекта используется ***точечная нотация***. После имени объекта (или класса) ставится разделяющая точка, вслед за которой указывается имя поля или метода. Все рассмотренные нами типы переменных являются классами, поэтому переменные этих типов – объекты, имеющие некоторые поля и методы. Обращение к полям мы уже встречали, когда знакомились с константами вида T.V, например **real**.MaxValue означает обращение к полю MaxValue класса **real**.

Полезно помнить, что каждый базовый (следовательно, заранее известный в программе) класс имеет методы Print и Println, позволяющие выводить значение объекта с последующим пробелом или сменой строки, как это делают стандартные процедуры Print и Println. Например, мы можем вывести значение **real**.MaxValue, записав **real**.MaxValue.Print. Если воспринимать точку, как разделитель в перечислении, такая запись читается вполне логично: «Класс **real**. Поле MaxValue. Вывести». Точки проявляют тенденцию множиться, но это нормальное явление и в дальнейшем станут понятны все преимущества такой записи, позволяющей в одной строке кода реализовать весьма большой алгоритм.

Метод Print работает как функция и возвращает в качестве результата переданные ему данные. Посмотрите пример, но не пишите подобного кода, пока не научитесь понимать этот механизм.

```
begin
   integer.MaxValue.Print.Println.Sqrt.Println.Sqr.Print
end.
```

```
2147483647 2147483647
46340.950001052
2147483647
```

Рассмотрим, как получились такие результаты. Значение **integer**.MaxValue передается методу Print, который выводит его, а далее передает в виде объекта класса **integer** с некоторым внутренним именем, например, X. Затем формируется выражение вида X.Println. Оно обращается к методу Println класса **integer** и выводит значение X, по-прежнему равное **integer**.MaxValue. И снова это значение передается дальше в виде объекта X. Получается конструкция вида X.Sqrt. Объект X автоматически приводится к классу **real**, как того требует аргумент метода Sqrt. Вычисляется значение квадратного корня и возвращается, как объект X класса **real**. Вновь получаем X.Println, значение X выводится, а объект X передается дальше. Остается выполнить X.Sqr.Print. Получаем с некоторой точностью исходное значение, но уже в типе **real**, и выводим его.

Каждый тип, реализованный в виде класса, можно расширить путем добавления новых методов, которые в этом случае называются ***методами расширения*** или просто ***расширениями***. При этом само расширение не затрагивает код класса, а остается в пользовательском коде. В PascalABC.NET имеется большое количество расширений классов Microsoft .NET Framework, а также собственных классов. Как и любой метод, расширение записывается через точку. В работе программисту обычно непринципиально знать, использует ли он метод класса или расширение.

> Не переходите к чтению следующей главы, пока не усвоите понятия класса и объекта, а также сопутствующие термины!

Глава 6

Последовательности

В этой главе…

Основные понятия
Точечная нотация
Последовательности
Массивы
Кортежи
Список List
Генераторы
Ввод и вывод
Фильтрация и проецирование
Выборка по условию
Сортировка и поиск
Разбиение и слияние последовательностей
Примеры решения задач

Последовательность означает, что из одной ошибки выводится целая цепь ошибок.

Юзеф Бестер,
Польский афорист

С последовательностями вы уже встречались в главе 4, когда рассматривали задачи, связанные с рядами чисел. Ряд – частный случай последовательности, члены которой связаны друг с другом некоторой математической зависимостью. Но в общем случае понятие последовательности шире: ее элементы могут быть быть связаны всего лишь каким-нибудь общим признаком. Например, очередь на кассе магазина. Это последовательность людей, связанных лишь тем, что им нужно пройти через кассу для оплаты товаров. Более того, если мы условно заменим какого-нибудь человека на робота, очередь все равно останется очередью, так что тут даже признак «человек» не является существенным.

Рассмотрим простую задачу. Даны радиусы нескольких окружностей (r = 3.5, 4.11, 2.6, 5.835, 7.52, 4.6). Считая, что S – среднее значение площадей окружностей радиусов r, перечислить те из r, которые образуют окружности с площадями, отличающимися от S не более чем на 40%. Математически решение можно записать так:

$$S = \frac{1}{n}\sum_{i=1}^{n} \pi r_i^2; \quad 0,6S \le \pi r_i^2 \le 1,4S$$

И обнаруживается самое интересное: требуется перебрать значения r в двух разных местах программы. Первый раз при определении средней площади, второй – при отборе значений r для вывода. Конечно, для каждого радиуса можно завести переменные r_1, r_2, и т.д. Но... а если радиусов будет двадцать? Или пятьдесят? В подобных случаях в языках программирования используются массивы. А в некоторых, например в PascalABC.NET – еще и последовательности, как обобщение массивов.

Последовательность – объект в памяти компьютера, состоящий из однотипных элементов, которые можно перебирать в порядке от первого элемента к последнему. Элементы последовательности **в памяти не хранятся**, хранится только алгоритм их получения.

Массив – хранимая нумерованная **последовательность** однотипных элементов с непосредственным доступом к любому элементу по его индексам, являющимся своеобразным аналогом номера.

Номер элемента массива однозначно связан с его индексами и эта связь устанавливается на основе описания массива. При этом сами индексы не хранятся.

Последовательностям и массивам в PascalABC.NET отведена важная роль и описание работы с ними занимает большой объем. Например, в книге «PascalABC.NET: Введение в современное программирование» часть 5, посвященная последовательностям, занимает более 70 страниц. Поэтому здесь описывается лишь самое основное.

До сих пор мы работали с переменными, за которыми скрывалось их единственное значение. Последовательности – другой тип объектов программы, за ними скрывается набор значений, но обращаться к последовательности мы точно так же можем по ее **имени**.

Посмотрите, как приведенная задача решается при помощи массива. Скорее всего, многое в этом коде будет непонятно, но он дается лишь в демонстрационных целях.

```
##
var ar := |3.5, 4.11, 2.6, 5.835, 7.52, 4.6|; // массив ar
var S := ar.Average(r -> Pi * r * r); // среднее
var (minS, maxS) := (0.6 * S, 1.4 * S); // границы отбора
ar.Where(r -> (Pi * r * r >= minS) and (Pi * r * r <= maxS)).Print
```

```
4.11 5.835 4.6
```

6.1. Последовательность или массив?

Прочитав условие задачи, не следует терзаться выбором между последовательностью и массивом. Достаточно понимать, что в процессе решения считается вполне нормальным переход от последовательности к массиву и наоборот. Это сводит проблему выбора к вопросу, с чего начать – с последовательности или с массива. Будьте немножко авантюристами и вспоминайте слова, приписываемые Наполеону Бонапарту: «Главное ввязаться в бой, а там посмотрим».

Последовательность – объект, элементы которого не хранятся в памяти, вместо этого хранится алгоритм, описывающий способ получения элементов (например, формула). В связи с этим доступ к элементам предоставляется поочередно от начала последовательности к ее концу. Так, нельзя получить доступ к пятому от начала элементу, не получив предварительно четырех предшествующих. Если последовательность создается на основе математической зависимости (формулы), либо заполняется при описании, значения ее элементов не будут зависеть от числа просмотров последовательности. Но если последовательность формируется на основе датчика случайных чисел, при каждом просмотре любой ее член будет изменяться.

Последовательности, элементы которых формируются на основе случайных чисел, либо вводятся с клавиатуры, должны обрабатываться за один просмотр.

В созданной последовательности нельзя изменить ни количество элементов, ни их значения. В связи с этим при необходимости получения измененной последовательности на основе исходной последовательности создается другая. Расширение .Count возвращает количество элементов в последовательности, но при этом выполняется ее **полный просмотр**. И это понятно: для того, чтобы пересчитать элементы, нужно получить каждый их них.

Массив хранит в памяти все свои элементы независимо от способа их получения и доступен для внесения изменений как по количеству элементов, так и по их содержимому. С массивом можно работать, как с последовательностью, но также можно просматривать его в обратном порядке и напрямую обращаться к нужным элементам. Можно также изменять значения элементов массива. Количество элементов массива хранится в поле .Length, так что никаких его просмотров при запросе длины не происходит. Что плохо – длинный массив может занимать в памяти много места.

Не существует жестких правил, регламентирующих в каком случае **для исходных данных** использовать массив, а в каком – последовательность. Последовательностей вообще нет во многих языках про-

граммирования и ничего, обходятся массивами. Но можно попытаться дать некоторые рекомендации новичкам.

Массив обязательно используется, если:
- в условии задачи присутствует фраза «Дан массив…», «В массиве …» или равнозначная им по смыслу;
- исходные данные формируются на основе датчика случайных чисел;
- заранее заданное количество исходных данных вводится с клавиатуры (либо из файла, на который переназначен ввод) и при этом данные не могут быть обработаны за один просмотр.

Последовательность обязательно используется, если исходные данные вводятся с клавиатуры (либо из файла, на который переназначен ввод) и при этом количество данных заранее неизвестно, но задано условие прекращения ввода. Ведь невозможно заранее создать массив, не зная его размера. Поэтому данные вводятся, как последовательность, которая при необходимости преобразуется в массив для хранения.

Во всех прочих случаях можно использовать как последовательность, так и массив. Что и когда окажется эффективнее, подскажет практика. Пока лишь отметьте, что если хотите писать короткие программы в современном стиле – используйте последовательности. Применение массивов делает программы более традиционными, приближенными по стилю написания к старым версиям языка.

Задачи с использованием последовательностей (массивов) решаются по единой схеме. Вначале последовательность создается и инициализируется, т.е. заполняется некоторыми начальными значениями. Далее следуют обработка последовательности и вывод полученных результатов, либо иное их использование.

Последовательность можно преобразовать в динамический массив, указав расширение .ToArray. Динамический массив всегда можно указывать вместо последовательности.

6.1.1. Последовательности

Знание о том, что последовательность не хранится, дает немало. Становится понятно, что можно работать с бесконечной последовательностью, т.е. такой, у которой количество членов не ограничено. Перебирать их до тех пор, пока не будет выполнено некоторое условие окончания перебора. А еще, если использовать последовательность, программа удовлетворяет требованию ЕГЭ быть «эффективной по памяти».

В PascalABC.NET последовательности именуются **sequence** (от английского слова «последовательность»). Тип членов последовательности можно указать в описании, но чаще последовательности создаются с автовыведением типа.

Проще всего создать последовательность при помощи функции Seq. Ее аргументы – единого типа перечисленные через запятую члены создаваемой последовательности.

```
var a := Seq(1, 9, -4, 12, 40, 39, 54);
var b: sequence of integer := Seq(3.5, 2.0, 6.417, -12.0);
var c := Seq(True, 3.5 > 1.63 ** 2.95, Sin(x) > 1, False, False);
```

При добавлении числа к последовательности или последовательности к числу, получается **новая** последовательность. Обратите внимание, что нельзя писать *a – 2*, поскольку операция вычитания с последовательностью не определена.

```
var a := Seq(4, 5, 6); // последовательность 4,5,6
a := 8 + a + (-2); // НОВАЯ последовательность 8,4,5,6,-2
```

Последовательности можно складывать друг с другом, при этом в результате также получается **новая** последовательность.

```
var a := Seq(1, 2, 3); // 1,2,3
var b := Seq(5, 10); // 5,10
a := a + (-3) + b; // 1,2,3,-3,5,10 – НОВАЯ последовательность
```

Последовательность обладает ***ленивостью***. Нужный ее элемент создается лишь тогда, когда он требуется, что позволяет создавать последовательности бесконечной длины. А почему нет, если элементы не хранятся? Мы ведь можем писать в математике формулы вида $S = 1 – 1/2 + 1/3 – 1/4 + … + (-1)^{k+1}/k$, где $k = 1, 2, 3, …$

6.1.2. Массивы

В отличие от последовательности, элементы массива хранят присвоенные им значения, а также позволяют их изменять. Доступ к элементу массива осуществляется путем указания его имени, за которым в квадратных скобках следует индекс или перечень индексов.

Массивы могут иметь различное число измерений. Пока будем рассматривать одномерные массивы, а позднее – двумерные, которые в PascalABC.NET называются ***матрицами***. Для массивов термин «***длина***» подразумевает общее количество элементов в массиве.

§1. Статические и динамические массивы

Память под **статический** массив распределяется на этапе компиляции программы. Одновременно может быть выполнена инициализация элементов массива. Границы индексов статического массива неизменны и должны быть указаны в программе константами или выражениями, содержащими только константы. Первый элемент массива имеет индекс, указанный в качестве нижней границы, последний – индекс, указанный в качестве верхней границы.

Статические массивы – дань совместимости с более ранними версиями языка Паскаль. Работа с ними в задачах, за исключением самых примитивных, громоздка и неудобна. Единственная область, в которой нельзя обойтись без статических массивов – обмен с типизированными файлами.

Длину статического массива нельзя менять. Значение нижней и верхней границы индекса массива *a* можно получить при помощи функций Low(a) и High(a) соответственно. Принципиально важно, что со статическими массивами **нельзя работать в точечной нотации**.

Динамический массив нужного размера может быть создан в том месте программы, где он впервые потребуется. Как и у последовательности, размер массива – это количество элементов, которое в нем в данный момент содержится. В динамических массивах **индексы начинаются от нуля**. Количество элементов в динамическом

массиве может меняться, но никогда не может стать отрицательным. Текущее количество элементов в массиве хранится в поле .Length, а максимальное текущее значение индекса – в поле .High. Первый элемент массива всегда имеет индекс ноль.

Поскольку динамические массивы является разновидностью последовательности, к ним применимы **все методы и расширения**, предназначенные для работы с последовательностями. Обратите внимание, что в результате обработки массива как последовательности, будет возвращена **последовательность.** Для массивов имеются также дополнительные методы и расширения, связанные с наличием у массива индексов и возможностью изменять значения элементов.

Динамические массивы являются основным и рекомендуемым для применения типом массивов в PascalABC.NET. Их очень удобно передавать в качестве аргументов в процедуры и функции. Возможность хранения данных позволяет использовать динамические массивы вместо последовательностей там, где требуется более одного просмотра элементов.

§2. Создание и инициализация массива

В PascalABC.NET массивы именуются **array**, что в английском языке означает «массив». Все члены массива должны иметь одинаковый тип, который указывается в описании.

Статический массив обычно описывается в виде

```
var ИмяМассива: array[m..n] of ТипЭлементов;
```

Конструкция вида [m..n] задает минимальное и максимальное значение, которое может принимать индекс массива, причем допускаются и отрицательные значения. Количество элементов в массиве можно вычислить по формуле n – m + 1. На этапе компиляции под элементы массива резервируется необходимое место в памяти в соответствии с типом элементов.

```
var a: array[0..12] of byte; // 13 элементов byte
var b, c: array[-5..8] of real; // два массива по 14 элементов real
```

Описание статического массива можно совместить с инициализацией его элементов, которая также проводится на этапе компиляции. Поскольку тип массива задан, можно при инициализации массива вещественного типа указывать в списке значения целочисленных типов.

```
var a: array[3..6] of integer := (1, 2, 3, 4);
var b: array[0..2] of real := (1.2, 5, -3.05); // 5 приведется к 5.0
var c: array [1..4] of real := (10, 11, 12, 13);
```

Описание и инициализация **динамического** массива отличаются лишь тем, что границы индексов не указываются. Вследствие этого компилятор не может отвести место под такой массив и необходимая память выделяется во время выполнения программы.

```
var a1: array of integer; // массив целых
var p: array of real; // массив вещественных
var q: array of boolean; // массив логических
```

Здесь описаны три массива, а их длина и значения элементов будут определены позднее – ведь массивы динамические. Массив принадлежит к определенному классу, который задается описанием массива. Поскольку в программе мы оперируем не классами, а объектами, созданными на основе классов, массив мало описать, его еще нужно создать.

Проще всего создать динамический массив при помощи функции Arr, возвращающей такой массив. Ее аргументы – единого типа перечисленные через запятую члены будущего массива. Вместо Arr( ) удобно использовать более короткую конструкцию | |.

```
var a := Arr(1, 9, -4, 12, 40, 39, 54);
var d := |1, 9, -4, 12, 40, 39, 54|; // альтернативный вариант
var b := |3.5, 2.0, 6.417, -12.0|;
var c := Arr(True, 3.5 > 1.63 ** 2.95, Sin(x) > 1, False, False);
```

Можно совместить описание динамического массива с его созданием, для чего потребуется указать необходимую длину массива.

```
var a := new integer[10]; // массив из 10 целых элементов
var b := new real[7]; // массив из 7 вещественных элементов
var c := new boolean[4]; // массив из 4 логических элементов
```

Здесь новое для вас ключевое слово **new** – вызов конструктора массива (см. 5.10).

Далее везде динамический массив будет именоваться просто «массив», если явно не указано иное. Статические масивы будут использоваться со словом «статический», если это не очевидно.

Стандартный конструктор массива запрашивает у операционной системы необходимую память и размещает там элементы массива, после чего **инициализирует их** некоторым значением по умолчанию. Для числовых типов это ноль, для логического – False. Если вы будете полагаться на эту инициализацию **при сдаче ЕГЭ**, эксперты снимут один балл с оценки решения в соответствии с чудаковатыми требованиями **всегда** выполнять инициализацию явно.

«Специально для ЕГЭ» можно создать массив и обнулить его элементы следующим образом:

```
var a := |0| * 15; // создан массив из 15 нулевых элементов
```

В случае, когда массив был описан заранее, его длину можно установить (либо поменять) вызовом процедуры SetLength.

```
var a1: array of integer; // массив целых
// тут какой-то участок программы
SetLength(a1, 13); // установлена длина массива a1, равная 13
```

Удобство использования SetLength состоит в том, что оставшиеся после изменения длины элементы сохранят свои значения, а новые, если они появятся, будут инициализированы значениями по умолчанию. Процедура SetLength умеет работать и с массивом, память под который она ранее не выделяла.

Тип динамического массива всегда «**array of** Тип». Это делает массивы одного и того же типа совместимыми между собой, что важно при передаче их в качестве параметров.

Для обращения к конкретному элементу массива нужно указать его имя и за ним в квадратных скобках индекс элемента, например, a[5], mas[0], Директор[Школа[Номер]]. Значение элемента массива можно изменить обыкновенным присваиванием.

В PascalABC.NET имеется возможность вести отсчет индексов от конца массива. На такой отсчет указывает символ ^, который записывается перед значением индекса. Последний элемент имеет индекс [^1], предпоследний – [^2] и т.д. Отсчет от конца преимущественно используется в ***срезах*** массивов.

Задача 6.1
Для приведенных описаний массивов указать их организацию (статический, динамический), тип и общее количество элементов.
var *a :=* ***new real****[5];*
var *b := |-10, 182, 0, 45, 0, 13|;*
var *c:* ***array****[-6..3]* ***of integer****;*
var *d:* ***array of integer*** *:= (3, -7, 9, 153);*

a – динамический, 5 элементов типа **real**;
b – динамический, 6 элементов типа **integer**;
c – статический, 10 элементов типа **integer**;
d – динамический, 4 элемента типа **integer**.

a

Порядковый номер	1	2	3	4	5
Индекс	0	1	2	3	4
Значение	0	0	0	0	0

b

Порядковый номер	1	2	3	4	5	6
Индекс	0	1	2	3	4	5
Значение	-10	182	0	45	0	13

c

Порядковый номер	1	2	3	4	5	6	7	8	9	10
Индекс	-6	-5	-4	-3	-2	-1	0	1	2	3
Значение										

d

Порядковый номер	1	2	3	4
Индекс	0	1	2	3
Значение	3	-7	9	153

Задача 6.2
Для приведенных описаний массивов указать сумму элементов с индексами 2 и 5, а также сумму элементов с номерами 2 и 5.
```
var a: array[-3..7] of integer := (3, 0, -2, 4, 9, 7, 8, -9, 11, -5, 6);
var b := |3, 0, -2, 4, 9, 6, 8, -9, 11, -5, 6|;
var c := new integer[11] (6, -5, 11, -9, 8, 6, 9, 4, -2, 0, 3);
var d: array[1..11] of integer := (6, -5, 11, -9, 8, 6, 9, 4, -2, 0, 3);
```

Первый статический массив *a* содержит 7 - (-3) + 1 = 11 элементов с индексами от -3 до 7. Номера элементов отличаются от их индексов на 1 - (-3) = 4. Следовательно, *a[2]* – это элемент с номером 2 + 4 = 6, *a[5]* – элемент с номером 9. Сумма *a[2] + a[5]* равна 7 + 11, т.е. 18. Сумма элементов с номерами 2 и 5 равна 0 + 9, т.е. 9.

Второй динамический массив *b* содержит 11 элементов (по числу элементов в списке инициализации) и имеет индексы от нуля, поэтому номера элементов на единицу больше их индексов (это всегда так для динамических массивов!). Следовательно, *b[2]* – это элемент с номером 3, *b[5]* – элемент с номером 6. Сумма *b[2] + b[5]* равна -2 + 6, т.е. 4. Сумма элементов с номерами 2 и 5, как и для массива *a* равна 9.

Третий динамический массив с содержит 11 элементов. Как и для массива *b, c[2]* – это элемент с номером 3, *c[5]* – элемент с номером 6. Сумма *c[2] + c[5]* равна 11 + 6, т.е. 17. Сумма элементов с номерами 2 и 5 равна -5 + 8, т.е. 3

Четвертый статический массив *d* содержит 11 - 1 + 1 = 11 элементов с индексами от 1 до 11 и в нем номера элементов совпадают с индексами. Сумма *d[2] + d[5]* равна -5 + 8, т.е. 3. Сумма элементов с номерами 2 и 5 также равна 3.

В задачах, связанных с индексами массивов, очень важно определить, о чем идет речь – о порядковом номере элемента или о его индексе. Лишь для статического массива с нижней границей, равной 1, этой проблемы не существует. Но если массив динамический, а в задаче предлагается определить, какой из элементов массива имеет максимальное значение, что имеется в виду? Какой элемент по порядку или с каким индексом? Решая задачи, мы вернемся к этой проблеме.

6.2. Вывод последовательности (массива)

Отличная новость, что можно всего лишь указать имя, чтобы манипулировать с последовательностью или массивом, как с единым целым. В жизни мы именно так и поступаем. Коробка с конфетами, сумка с продуктами, полка с книгами, шкаф с одеждой… можно бесконечно перечислять. Но иногда нужно и внутрь заглянуть.

PascalABC.NET позволяет выводить значения **любых** объектов программы одним оператором. Например, посредством уже привычных Write (Writeln) и Print (Println). В качестве напоминания о том, что мы имеем дело с объединением элементов, вывод окружается квадратными скобками. Как Write, так и Print, перечисляют элементы через запятую. Print по окончании вывода дополнительно делает пробел.

```
##
var a := Seq(1, 2, 3); // последовательность
var b := Seq(5, 10); // последовательность
var c := (a + (-3) + b).ToArray; // массив
Print(c)
```

```
[1,2,3,-3,5,10]
```

В большинстве школьных задач требуется вывести элементы в ином виде, например, через запятую. И, конечно, без скобок. Используйте метод .Print (или .Println). По умолчанию вывод идет через один пробел, но можно указать необходимый разделитель.

```
##
var c := Seq(1, 2, 3) + (-3) + Seq(5, 10);
c.Print(', ')  // указан вид разделителя запятая и пробел
```

```
1, 2, 3, -3, 5, 10
```

Для статических массивов использование точечной нотации не предусмотрено, поэтому метод .Print для них официально не поддерживается. Если вдруг Вы обнаружите, что какие-то методы работают и для статических массивов, можете пользоваться ими для отладки. Но не следует возмущаться, если в очередной версии языка это перестанет работать. Выводите элементы статических массивов в обычном цикле.

6.3. Перебор элементов в цикле

В определении последовательности было сказано, что ее элементы можно перебирать в порядке от первого к последнему. Элементы массива можно также перебирать, причем и в направлении от последнего элемента к первому. Перебор – всегда цикл. Вы знакомы с четырьмя разновидностями циклов; давайте познакомимся с пятой.

Для перебора всех элементов (*a*) последовательности или массива (*s*) служит цикл **foreach** (англ. for each – для каждого). В таком цикле индексы массива не используются.

```
foreach var a in s do
  оператор или блок;
```

В теле цикла переменная *a* будет поочередно принимать значения каждого элемента из *s*. Конечно, англоязычным пользователям несколько проще, ведь они просто читают заголовок цикла: «Для каждого *a* в *s* выполнить…». Ну а вам остается утешительная мысль о том, что заодно приобщаетесь к английскому языку.

Задача 6.3
Найти и вывести с 5 знаками после запятой сумму кубических корней ряда чисел 37, 194, -18, 46, 59, 13, 2183, -42, 7, 66, -111

Массив здесь ни к чему, алгоритм однопроходный, выражений, связанных с индексами, нет. Воспользуемся последовательностью, поскольку надо где-то разместить данные. В PascalABC.NET функция нахождения кубического корня отсутствует, поэтому придется использовать формулу $\mathrm{Sign}(x)\cdot|x|^{1/3}$. Вы же помните, что отрицательные числа нельзя возводить в нецелую степень?

```
##
var a := Seq(37, 194, -18, 46, 59, 13, 2183, -42, 7, 66, -111);
var Sum := 0.0; // 0.0, потому что нужен тип real
foreach var x in a do
  Sum += Sign(x) * Abs(x) ** (1 / 3);
Write(Sum:0:5)
```

```
26.97240
```

Задача 6.4
Сгенерировать 15 случайных целых чисел на отрезке [-20; 50]. Вывести те из чисел, которые по абсолютной величине меньше среднего арифметического нечетных чисел.

В этой задаче без массива не обойтись. Требуется два цикла. В первом (**for**, а не **foreach**, потому что нужен индекс для записи в элемент массива) генерируем случайные числа, выводим их и находим количество нечетных чисел, а также их сумму. Вычисляем среднее значение. Во втором цикле отбираем данные по условию и выводим их.

```
begin
  var a := new integer[15]; // динамический массив
  var (k, s) := (0, 0); // для количества и суммы
  for var i := 0 to a.High do // требуется запись в i-й элемент
  begin
    a[i] := Random(-20, 50); // очередное случайное число
    Print(a[i]);
    if a[i].IsOdd then // если нечетное
    begin
      k += 1; // +1 в количество
      s += a[i] // накапливаем сумму
    end
  end;
  Writeln; // переходим на новую строку вывода
  var m := s / k; // среднее арифметическое нечетных
  foreach var p in a do // перебор всех элементов
    if Abs(p) <= m then Print(p)
end.
```

```
35 36 24 24 7 1 36 36 27 29 0 -1 11 37 -1
7 1 0 -1 11 -1
```

6.4. Ввод элементов с клавиатуры

Последовательность можно сформировать, вводя данные с клавиатуры. При этом нужно либо указывать число элементов, подлежащих вводу, либо осуществлять ввод до тех пор, пока выполняется какое-либо условие. Например, вводить числа, пока не встретится ноль.

В PascalABC.NET включены следующие функции для формирования последовательности типа **integer**:

ReadSeqInteger(n) – последовательность из n целых чисел;

ReadSeqInteger('приглашение', n) – то же, с предварительным выводом приглашения ко вводу;.

ReadSeqIntegerWhile(**integer** -> **boolean**) – ввод целых чисел, формирующих последовательность, до тех пока лямбда-выражение истинно;

ReadSeqIntegerWhile('приглашение', **integer** -> **boolean**) – то же, с предварительным выводом приглашения ко вводу.

Для ввода вещественных значений имеются функции ReadSeqReal и ReadSeqRealWhile.

Массив формируется путем ввода с клавиатуры значений указанного количества элементов.

ReadArrInteger(n) – ввод массива из n целых чисел;

ReadArrInteger('приглашение', n) – то же, с предварительным выводом приглашения ко вводу.

Для ввода вещественных значений имеется функция ReadArrReal.

Задача 6.5

С клавиатуры вводится последовательность целых чисел, ограниченная нулем (ноль не входит в последовательность). Получить и вывести квадраты нечетных чисел, не превышающих 15.

Здесь мы вынуждены использовать последовательность, поскольку размер массива заранее определить невозможно.

```
##
var s := ReadSeqIntegerWhile(t -> t <> 0);
foreach var n in s do
  if n.IsOdd and (n <= 15) then Print(n * n)
```

```
9 35 12 -5 20 17 -20 13 -9 -8 3 0
81 25 169 81 9
```

Решим приведенную задачу без использования последовательностей.

```
##
var n := ReadInteger;
while n <> 0 do
begin
  if n.IsOdd and (n <= 15) then Print(n * n);
  n := ReadInteger
end
```

6.5. Фильтрация и проецирование

Это термины, пришедшие из функционального программирования и они превосходные помощники в решении многих задач.

Фильтр. Обычно его образ ассоциируется с некоторым устройством, которое призвано что-то задерживать. Противомоскитная сетка на окне – фильтр, задерживающий мух и комаров. Антивирусная программа в компьютере – фильтр, который должен задерживать вирусы. Контролер на входе в кинозал – фильтр, задерживающий безбилетников. Процесс пропускания через фильтр называется **фильтрацией**.

В программировании фильтруют обрабатываемые данные. Лишь прошедшие через фильтр данные примут участие в дальнейшей обработке. Пусть дана некоторая последовательность чисел и требуется найти сумму тех из них, которые кратны трем. Представим, что у нас есть готовая функция, принимающая в качестве аргумента числовую последовательность и возвращающая сумму ее членов. Тогда задачу можно решить, передав этой функции отфильтрованные по условию кратности трем элементы исходной последовательности. Фильтрация реализуется с помощью лямбда-выражений. Условие фильтрации считается выполненным для тех элементов, для которых лямбда-выражение истинно. А сам фильтр – это метод последовательности:

```
s.Where(T -> boolean)
```

Здесь *s* – последовательность или объект, приводящийся к ней (например, динамический массив), *Where* – ключевое слово, определяющее операцию фильтрации (англ. «где»), а лямбда-выражение понимается так, что для каждого элемента типа T из *s* должно возвращается логическое значение, истинность которого позволит пройти элементу через фильтр. Для случая, приведенного в качестве примера, можно записать s.**Where**(t -> t **mod** 3 = 0).

В результате фильтрации последовательность может сохранить все элементы, либо потерять их часть и даже все. В последнем случае последовательность окажется пустой!

Проецирование сохраняет исходное количество элементов последовательности, но преобразовывает каждый элемент по некоторому правилу. Аналог – мясорубка. На входе мясо – на выходе фарш, причем именно из этого мяса. Проецирование осуществляет функция, преобразующая входные элементы по правилу, заданному лямбда-выражением.

```
s.Select(T -> T1)
```

Здесь *s* – последовательность элементов или объект, приводящийся к ней (например, динамический массив), *Select* – ключевое слово, определяющее операцию фильтрации (не особо удачное англ. «выбрать»), а лямбда-выражение понимается так, что для каждого элемента типа T из *s* будет возвращается некоторое значение типа T1 (конечно, возможно совпадение типов T и T1), которое будет передано «на выход». Например, получить последовательность квадратов элементов исходной последовательности *s* можно, если выполнить проецирование *s.Select(t -> t * t)*.

Использование фильтрации и проецирование существенно меняет стиль программирования, позволяя в одной строке записывать достаточно сложные алгоритмы. Здесь надо отчетливо понимать, что реализация содержит неявный цикл просмотра всей последовательности.

Фильтрация и проецирование всегда возвращают последовательность.

В качестве иллюстрации решим задачу 6.5 с использованием фильтрации и проецирования.

Задача 6.5 (повторно)
С клавиатуры вводится последовательность целых чисел, ограниченная нулем (ноль не входит в последовательность). Получить и вывести квадраты нечетных чисел, не превышающих 15.

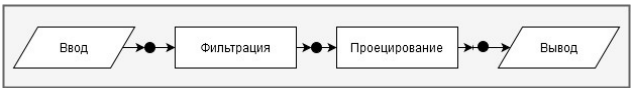
```
##
var s := ReadSeqIntegerWhile(t -> t <> 0); // ввод данных
s := s.Where(n -> n.IsOdd and (n <= 15)); // фильтрация
s := s.Select(n -> n * n); // проецирование
s.Print // вывод
```

На основе введенных данных создается последовательность *s*. Далее, к ней применяется фильтрация, реализующая критерий отбора. Результат фильтрации также является последовательностью и ее можно запомнить вместо исходной *s*. За фильтрацией следует проецирование, преобразующее каждый элемент последовательности в его квадрат. И снова результатом будет последовательность, и снова ее можно запомнить вместо исходной *s*. Остается вывести полученную последовательность методом *Print*.

Но ведь элементы последовательности на самом не хранятся. И последовательности ленивы. Запрос на ввод исходных значений будет выдан только лишь при выполнении метода *Print*. А все эти промежуточные присваивания переменной s – лишь способ довести до компилятора алгоритм получения результирующей последовательности. Фактически, все операторы приведенной программы, кроме *s.Print* – невыполняемые, они лишь описывают алгоритм обработки. Чтобы убедиться в этом, закомментируйте строку с *s.Print* и запустите программу на выполнение. Запроса на ввод данных не будет! Программа запустится и сразу же завершится. Эти рассуждения и опыты подводят к мысли о том, что неплохо иметь возможность все манипуляции с последовательностью записывать как единый оператор PascalABC.NET. И такая возможность существует. Она реализуется при помощи **точечной нотации**.

6.6. Точечная нотация

PascalABC.NET поддерживает механизм передачи данных между объектами программы, делающий программный код компактным и одновременно наглядным. Он назван ***точечной нотацией***. Точка – это место, где происходит стыковка объектов по данным. Объект, расположенный левее точки, служит источником данных, а объект справа от точки служит ее приемником. Соединение многих объектов в цепочку позволяет программировать в одной строке длинные алгоритмы. Вот так выглядит цепочка в приведенной выше задаче 6.5:



Запишем решение этой задачи в точечной нотации.

```
## ReadSeqIntegerWhile(t -> t <> 0)
    .Where(n -> n.IsOdd and (n <= 15))
    .Select(n -> n * n)
    .Print
```

Вам хорошо видны точки в коде? Конечно, можно код вытянуть в одну строку или записать в пару строк – это не Python, так что форматирование, обеспечивающее наилучшее восприятие кода, определяется вашими предпочтениями.

В точечной нотации мы последовательно читаем слева направо все, что будет выполнено. Это естественная и наглядная форма записи алгоритма. Во многом благодаря точечной нотации рисование блок-схем теряет смысл. Какова великая сакральная цель рисования стрелок вместо точек и помещения каждого фрагмента кода, находящегося между точками, в рамочку?

Давайте на приведенном примере научимся произносить код, записанный в точечной нотации. Ставим точки после каждой «переведенной на русский язык» фразы на тех же местах, что и в коде.
«Читать с клавиатуры последовательность целых чисел до тех пор, пока не будет введен ноль. Отобрать те из чисел, которые нечетны и не превышают 15. Каждое отобранное число возвести в квадрат. Вывести полученные значения»

Сравним наш «перевод» с формулировкой исходной задачи 6.5:
«С клавиатуры вводится последовательность целых чисел, ограниченная нулем (ноль не входит в последовательность). Получить и вывести квадраты нечетных чисел, не превышающих 15.»

Для большинства школьных задач точечная нотация позволяет читать условие задачи и сразу писать код.

Вы все еще видите необходимость в рисовании блок-схем?

6.7. Экстремумы, сумма, среднее, ...

PascalABC.NET предоставляет методы расширения для последовательностей и массивов, позволяющие получать значения минимума, максимума, среднего значения, суммы и произведения их элементов.

6.7.1. Минимум и максимум

s.Min возвращает значение первого минимального элемента последовательности или массива s;

s.Max возвращает значение первого максимального элемента последовательности или массива s;

s.Min(T -> T1) – преобразует каждый элемент последовательности или массива в соответствии с правилом, заданным лямбда-выражением и возвращает значение первого минимального среди полученных значений;

s.Max(T -> T1) – преобразует каждый элемент последовательности или массива в соответствии с правилом, заданным лямбда-выражением и возвращает значение первого максимального среди полученных значений;

s.MinBy(T -> T1) – преобразует каждый элемент последовательности или массива в соответствии с правилом, заданным лямбда-выражением, находит первый минимальный элемент среди полученных значений и возвращает соответствующий исходный элемент;

s.LastMinBy(T -> T1) – то же, последний минимальный элемент;

s.MaxBy(T -> T1) – преобразует каждый элемент последовательности или массива в соответствии с правилом, заданным лямбда-выражением, находит первый максимальный элемент среди полученных значений и возвращает соответствующий исходный элемент;

s.LastMaxBy(T -> T1) – то же, последний максимальный элемент.

Задача 6.6

С клавиатуры вводятся целые числа, ограниченные нулем. Сам ноль в набор чисел не входит. Найти и вывести значения максимального и минимального из введенных чисел.

Количество вводимых чисел заранее неизвестно, поэтому используем последовательность. Нахождение максимума и минимума потребуют

двух просмотров вводимых чисел, что требует запомнить последовательность в массиве.

```
##
var a := ReadSeqIntegerWhile(t -> t <> 0).ToArray;
Write('Min = ', a.Min, ', max = ', a.Max)
```

```
35 -17 12 21 22 7 54 59 -2 -27 56 0
Min = -27, max = 59
```

Задача 6.7
Даны n целых чисел, принадлежащих отрезку [-99; 99]. Найти и вывести значение последнего четного числа, максимального по абсолютной величине.

Принципиальное отличие от задачи 6.6 состоит в том, что здесь находится максимум не значения элемента, а некоторой функции от этого значения, в данном случае – абсолютной величины. При этом требуется вывести значение самого элемента. Алгоритм однопроходный, что позволяет использовать последовательность. Условие четности напоминает о фильтрации.

```
##
var n := ReadInteger('Укажите количество чисел:');
ReadSeqInteger('Вводите:', n)
    .Where(t -> t.IsEven)
    .LastMaxBy(t -> Abs(t))
    .Print
```

```
Укажите количество чисел: 15
Вводите: 71 38 -61 -80 -14 9 -63 -33 89 -29 -85 -60 -35 -67 2
-80
```

На примере нахождения минимума еще раз отметим:
.Min – просто первый минимум среди значений;
.Min(T -> T1) – первый минимум от преобразованных значений;
.MinBy(T -> T1) – первое исходное значение, минимальное среди преобразованных значений.

6.7.2. Сумма и произведение

s.Sum возвращает значение суммы элементов последовательности или массива *s*;

s.Product возвращает значение произведения последовательности или массива *s*;

s.Sum(T -> T1) – преобразует каждый элемент последовательности или массива *s* в соответствии с правилом, заданным лямбда-выражением и возвращает значение суммы полученных значений;

s.Product(T -> T1) – то же, но возвращает произведение.

При вычислении суммы целочисленных значений проблемы, связанные с переполнением разрядной сетки, возникают довольно редко. Но при вычислении произведения целых чисел шанс получить ошибку велик – достаточно вспомнить проблемы с вычислением факториалов. Универсальное решение состоит в том, чтобы накапливать произведение в переменной типа **real** и примириться с тем, что его точность не превысит 15 знаков. Это лучше, чем 8 (иногда 9) знаков в числах типа **integer**, но хуже, чем 19 знаков в числа типа **int64**. Не случайно функция Sqr(k) для k типа **integer** возвращает его квадрат типа **int64**. Расширение .Product поступает точно также, возвращая для элементов типа **integer** значение типа **int64**. Если заранее понятно, что в некоторых случаях результат может превысить **int64**.MaxValue, можно воспользоваться лямбдой: .Product(t -> **real**(t)).

Задача 6.8

Даны n случайных чисел, находящихся в диапазоне от -100 до 100. Найти и вывести значения суммы и произведения его элементов, не кратных трем. Количество чисел не превышает 50.

Оценим максимальное значение для суммы и произведения. В самом худшем и нереальном случае значение каждого элемента равно 100, а количество элементов равно 50. Тогда сумма достигнет величины $100 \times 50 = 5000$, а произведение $100^{50} = 10^{100}$. Следовательно, тип переменной для суммы можно оставить **integer**, а для произведения придется брать тип **real**.

Наличие случайных чисел вынуждает использовать для их хранения массив. Отбор элементов выполним при при помощи фильтрации. Но PascalABC.NET не имеет средств, позволяющих за один проход найти и сумму, и произведение элементов последовательности. Поэтому результат фильтрации придется связать с некоторым промежуточным именем, а затем отдельно найти сумму и произведение.

```
##
var n := ReadInteger('Укажите длину массива:');
var a := ArrRandom(n, -100, 100);
var s := a.Println.Where(t -> t mod 3 <> 0);
Write('Сумма ', s.Sum, ', произведение ', s.Product(t -> real(t)))
```

```
Укажите длину массива: 17
40 -20 59 -13 39 97 93 -25 -45 24 -86 66 42 -32 -31 -73 20
Сумма -64, произведение -1.853361226496E+17
```

6.7.3. Среднее значение

s.Average – возвращает среднее арифметическое значение элементов последовательности или массива *s*;

s.Average(T -> T1) – преобразует каждый элемент последовательности или массива *s* в соответствии с правилом, заданным лямбда-выражением и возвращает среднее арифметическое полученных значений.

Среднее значение элементов последовательности (массива) вычисляется как отношение суммы значений элементов к их количеству. Если количество элементов нулевое, произойдет попытка деления нуля на ноль с результатом NaN (Not a Number - «не число»). Что в таком случае делать, определяется условиями задания.

Задача 6.9

С клавиатуры вводится последовательность натуральных чисел, не превышающих 10000 и ограниченная неположительным значением, которое в последовательность не входит. Найти среднее значение для последовательности квадратных корней введенных значений и вывести его с пятью знаками в дробной части. Если не введено ни одного натурального числа, в качестве результата вывести ноль.

```
##
var r := ReadSeqIntegerWhile(t -> t > 0)
    .Average(t -> Sqrt(t));
Write(r:0:5)
```

```
16 39 178 3 0
6.32968
```

Все будет превосходно… если только первое введенное значение будет натуральным числом. Поскольку в противном случае мы получим даже не NaN, нет. Получим вот такое сообщение: «Ошибка времени выполнения: Последовательность не содержит элементов». Вроде и понятно все в нем. Но задание требует вывести в этом случае ноль, а не сообщение. Расширение .Average не настолько тупое, чтобы искать среднее значение пустой последовательности. Как же быть?

В подобных случаях на помощь приходит расширение .DefaultIfEmpty («по умолчанию, если пусто»). Если последовательность (массив) не пустая, будет возвращена ровно она же. Если пустая – последовательность из одного элемента со значением по умолчанию. Для числовых типов умалчиваемое значение – ноль.

```
##
var r := ReadSeqIntegerWhile(t -> t > 0)
    .DefaultIfEmpty
    .Average(t -> Sqrt(t));
Write(r:0:5)
```

Теперь все будет работать как задумано.

6.8. Кортежи

Кортеж – это последовательность элементов, доступная только на чтение и содержащая от двух до семи элементов. Каждый элемент кортежа нумеруется; нумерация ведется от нуля. Для доступа к элементу кортежа указывается его имя, за которым в квадратных скобках следует индекс элемента, например, четвертый по порядку элемент кортежа *t* записывается в виде *t[3]*. Внешне работа с кортежами похожа на работу с динамическими массивами.

Для кортежей в PascalABC.NET имеется общий тип **tuple**. Элементы кортежа могут иметь различный тип, в том числе, элемент может в

свою очередь быть кортежем. Тип конкретного кортежа может оказаться достаточно сложным, поэтому лучше пользоваться автовыведением типа.

```
var K := (1, 2.5, Seq(-2, 6, 0));
```

В приведенном примере создается кортеж *K*. Его первый элемент *K[0]* имеет тип **integer**, второй *K[1]* – тип **real**, третий *K[2]* – тип **sequence of integer**. А какой же тогда тип у переменной *K*? Давайте это выясним.

```
##
var K := (1, 2.5, Seq(-2, 6, 0));
K.Println; // вывод кортежа
Println(K.GetType) // вывод типа этого кортежа
```

```
(1,2.5,[-2,6,0])
System.Tuple`3[System.Int32,System.Double,System.Collections.Generic.IEnumerable`1[System.Int32]]
```

Конечно, если вы можете точно указать подобный тип для любого кортежа, дальше читать не нужно.

Последовательности **sequence**, как уже не раз было сказано, не хранятся. А кортеж хранится. Противоречия тут нет, поскольку в кортеже будет храниться не сама последовательность, а ссылка на нее. Отправка в кортеж последовательности, элементы которой задаются случайными числами или вводятся с клавиатуры – отличный способ породить в недалеком будущем поток не лучших в родном языке слов… в собственный адрес.

При необходимости кортеж можно распаковать в отдельные переменные с помощью кортежного присваивания. Мы делали аналогичную операцию не раз, описывая несколько переменных с их одновременной инициализацией. Если каждый элемент кортежа окажется последовательностью, кортежное присваивание выполнит распаковку в отдельные последовательности. Ведь поместив в пакет несколько коробок конфет, мы намереваемся впоследствии извлечь из него эти же коробки, а вовсе не отдельные конфеты.

В качестве примера использования кортежей рассмотрим обмен местами значений двух переменных. Традиционное решение состоит в цепочке присваиваний через промежуточную переменную.

```
##
var (a, b) := (5, 18);
Println(a, b); // 5 18
var t := a;
a := b;
b := t;
Println(a, b) // 18 5
```

Кортежи дают более короткое и наглядное решение задачи.

```
##
var (a, b) := (5, 18);
Println(a, b); // 5 18
(a, b) := (b, a); // кортежное присваивание
Println(a, b) // 18 5
```

Существует процедура Swap(a, b), совершающая обмен значениями двух переменных и работающая несколько быстрее обмена посредством кортежного присваивания.

Рассмотрим один из примеров, в котором использование кортежей делает код более элегантным.

Задача 6.10

Заполнить массив из n элементов целочисленными значениями, вводимыми с клавиатуры. Вывести элементы массива, принадлежащие отрезку [min+10; max-12], где min и max – значения минимального и максимального элементов массива соответственно.

Напишем функцию *НайтиГраницы*, возвращающую кортеж, содержащий нужные для решения границы отрезка. Использование такой функции позволит убрать второстепенный алгоритм (вычисление границ отрезка) из основной программы, что повысит наглядность кода. К тому же, если условие для нахождения границ изменится, с одного взгляда можно будет понять, где потребуется внести изменения в код. Почему кортеж? Потому что функция не может вернуть два отдельных значения, только одно.

```
function НайтиГраницы(Массив: array of integer): (integer, integer);
begin
  var (min, max) := (integer.MaxValue, integer.MinValue);
  foreach var Элемент in Массив do
    if Элемент < min then
      min := Элемент
    else if Элемент > max then
      max := Элемент;
  Result := (min + 10, max - 12) // кортеж
end;

begin
  var ЧислоЭлементов := ReadInteger;
  var Массив := ReadArrInteger(ЧислоЭлементов);
  var (ЛеваяГраница, ПраваяГраница) := НайтиГраницы(Массив);
  Массив.Where(Элемент -> Элемент in ЛеваяГраница..ПраваяГраница).Print
end.
```

```
15
87 -66 17 -12 75 25 -14 -66 68 -23 77 37 55 2 62
17 -12 25 -14 -23 37 55 2 62
```

С помощью пользовательской функции определяются границы *min* и *max*, которые затем используются при фильтрации.

6.9. Об операции присваивания

Скорее всего, вы уже обратили внимание, что в основе создания последовательности, массива и кортежа лежит присваивание. В правой части оператора присваивания вызов некоторой функции создает нужный объект. Далее этот объект присваивается переменной, имя которой указано в левой части.

В разделе 5.1 был описан механизм обмена данными между фактическими и формальными параметрами подпрограмм. При передаче по значению данные копировались, а при передаче по ссылке копировалась только сама ссылка. С последовательностями, массивами и кортежами происходит примерно такой же процесс – передается ссылка. К примеру, после того как массив создан, оператор присваивания связывает его с заданным именем. Копирование элементов массива не происходит. Исключение составляют статические массивы: для них выполняется физическое копирование элементов и это

еще один повод говорить о неэффективности работы со статическими массивами. Рассмотрим пример.

```
##
var A := |0, 1, 2, 3, 4|;
A.Println;
var B := A; // копирования нет, B - еще одно имя массива A
B[2] := 7;
Println(A[2], B[2]);
var C: array[0..4] of integer := (0, 1, 2, 3, 4);
var D := C; // копирование есть, D и C – два разных массива
D[2] := 7;
Print(C[2], D[2]);
```

```
0 1 2 3 4
7 7
2 7
```

Элемент *A[2]* изменил свое значение после изменения значения *B[2]*. И это было ожидаемо, если помнить, что присваивание *B := A* всего лишь позволяет переменным (именам) *B* и *A* ссылаться на один и тот же массив. Также понятно, почему изменение значения *D[2]* не повлекло изменения *C[2]* – массив *C* объявлен статическим.

А если все же нужно сделать именно копию динамического массива? Для этого существует функция Copy и достаточно вместо оператора ***var*** *B := A* написать ***var*** *B := Copy(A)*.

Еще одна интересная возможность PascalABC.NET – распаковка последовательности в переменные. Вы уже знакомы с кортежным присваиванием вида ***var*** *(a,b,c) := (15, 3.8, x > y)*. В правой части подобного оператора может находиться также последовательность или массив. При этом в присваивании будут участвовать столько элементов, считая от начала, сколько переменных указано в левой части.

```
##
var a := ReadArrInteger('Введите 7 целых чисел:', 7);
var (max1, max2, max3) := a.OrderDescending;
Print('Три первых максимальных:', max1, max2, max3)
```

```
Введите 7 целых чисел: 35 126 -7 0 14 82 21
Три первых максимальных: 126 82 35
```

6.10. Генераторы (заполнение данными)

Под генераторами понимаются функции, которые используются для генерации последовательностей и массивов, т.е. их создания и инициализации некоторыми значениями. В результате вызова генератора возвращается последовательность или массив.

Еще раз вспомним, что последовательности ленивы, т.е. в некоторый момент времени в памяти компьютера существует только один элемент последовательности, к которому происходит обращение. Рассмотрим пример.

```
##
Println('Начало работы');
var a := ReadSeqInteger(10);
Print('Конец работы')
```

```
Начало работы
Конец работы
```

ReadSeqInteger – генератор, создающий последовательность из целочисленных значений, введенных с клавиатуры. Результатом будет присваивание переменной *a* последовательности типа **sequence of integer**, содержащей 10 элементов. Поскольку в программе нет обращения к переменной *a*, запроса на ввод данных также не будет. Зато если обратиться к *a* в двух разных местах программы, придется дважды вводить данные. Это логично: если данные не хранятся, откуда их брать при повторном обращении к последовательности? Конечно, откуда и при первом обращении – с клавиатуры.

6.10.1. Генераторы арифметической прогрессии

Используются, когда значения элементов отстоят друг от друга на фиксированную величину (шаг), которая может быть и отрицательной. Все генераторы этого типа, кроме PartitionPoints, возвращают **целочисленные последовательности**.

n.Times – элементы 0, 1, 2, … n-1;
n.Range – элементы 1, 2, 3, … n;
a.To(b) – элементы a, a+1, a+2, … b;
a..b – то же, альтернативная запись;
Range(a, b) – то же, альтернативная запись;
a.DownTo(b) – элементы a, a-1, a-2, … b;

Range(a, b, h) – элементы a, a+h, a+2h, … пока a+kh ⩽ b, h>0;
Range(a, b, h) – элементы a, a-h, a-2h, … пока a-kh ⩾ b, h<0;
(a..b).Step(h) – альтернативная запись для Range;
PartitionPoints(a, b, n) – последовательность элементов типа **real**, разбивающих отрезок [a; b] на n частей (для табуляции функций и т.п.).

Такие генераторы – удобная замена циклам с вычислением значений, зависящих от порядкового номера элемента. Теперь можно писать программы с вычислением сумм, количеств и экстремумов в точечной нотации, существенно сократив код.

Задача 6.11
Среди первой тысячи натуральных чисел найти сумму тех из них, которые оканчиваются цифрой 7.

```
## 1000.Range.Where(n -> n mod 10 = 7).Sum.Print  // 50200
```

Вся задача – в одну строку. Привет, Python! Кажется, что это сложно? Отнюдь. Пройдет совсем немного времени, и вы будете писать так же, соболезнуя тем, кто все еще использует «древние паскали».

Конечно, можно говорить, что это решение неоптимальное, что можно начать от 7 и идти с шагом 10. Зато оно написано в точности по заданию. А более оптимальное написать тоже просто:

```
## (7..1000).Step(10).Sum.Print  // 50200
```

6.10.2. Заполнение константой

Можно заполнить последовательность (массив) одним и тем же значением.

SeqFill(n, a) – n элементов со значением a. Тип a определяет тип последовательности.
ArrFill(n, a) – n элементов со значением a. Тип a определяет тип массива.

```
##
var n := ReadInteger('Число элементов:');
var k := ReadInteger('Каким значением заполнить:');
var A := ArrFill(n, k);  // есть альтернатива: |k| * n
A.Print
```

```
Число элементов: 13
Каким значением заполнить: 7
7 7 7 7 7 7 7 7 7 7 7 7 7
```

Альтернативный способ – использовать умножение. Для последовательностей можно писать Seq(a) * n, для массивов |a| * n.

6.10.3. Генераторы случайных значений

Последовательности и массивы могут содержать случайные значения. Для получения таких значений используется датчик псевдослучайных чисел, поэтому последовательности должны обрабатываться за один проход. Для массива число проходов неважно, поскольку данные в нем хранятся.

SeqRandom(n, a, b) – последовательность n целочисленных элементов, случайно распределённых на отрезке [a; b];

SeqRandomInteger(n, a, b) – другое имя для SeqRandom;

SeqRandomReal(n, a, b) – последовательность n вещественных элементов, случайно распределенных в промежутке [a; b);

ArrRandom(n, a, b) – массив из n целочисленных элементов, случайно распределенных на отрезке [a; b];

ArrRandomInteger(n, a, b) – другое имя для ArrRandom;

ArrRandomReal(n, a, b) – массив из n вещественных элементов, случайно распределённых в промежутке [a; b).

Задача 6.12

Для 15 случайных целых чисел на отрезке [-55; 92] найти минимальное, максимальное и среднее значение.

В данном случае решать задачу можно как при помощи последовательности, так и при помощи массива. Сначала напишем программу на основе последовательности с обработкой в цикле **foreach**.

```
##
var n := 15; // количество чисел
var (Мин, Макс, s) := (integer.MaxValue, integer.MinValue, 0);
foreach var m in SeqRandom(n, -55, 92) do
begin
  Print(m);
  Мин := Min(Мин, m);
  Макс := Max(Макс, m);
  s += m  // сумма для среднего
end;
Writeln;
Write('Мин = ', Мин, ', макс = ', Макс, ', среднее = ', s / n:0:2)
```

```
-30 1 12 60 -9 61 63 2 -41 75 66 62 -55 40 -49
Мин = -55, макс = 75, среднее = 17.20
```

Здесь использовано традиционное в PascalABC.NET присваивание переменной, предназначенной для хранения минимума, максимально возможного значения ***integer****.MaxValue* (можно также использовать *MaxInt*), а также переменной, предназначенной для хранения максимума, минимально возможного значения ***integer****.MinValue*. Можно присваивать этим переменным любые значения за пределами указанного в задании диапазона, например, 100 и -100 соответственно.

Использование точечной нотации существенно сократит код:

```
##
var a := ArrRandom(15, -55, 92); // массив случайных чисел
a.Println;
Write('Мин = ', a.Min, ', макс = ', a.Max, ', среднее = ', a.Average:0:2)
```

Здесь обращение к данным происходит трижды, поэтому нужно использовать массив. Конечно, массив занимает память, но простота и скорость написания кода важнее, чем сэкономленные 60 байт.

6.10.4. Бесконечные последовательности

Если последовательность бесконечна, то как остановить перебор ее членов? Перебор в цикле **foreach** можно прервать вызовом **break**. А вот вывод последовательности посредством Write или Print можно прервать, лишь прервав выполнение всей программы. Неужели с бесконечными последовательностями можно работать только перебирая их элементы в **foreach**? К счастью, нет. Существуют методы,

позволяющие ограничивать такие последовательности. А пока познакомьтесь с генераторами бесконечных последовательностей.

m.Step – получение бесконечной последовательности элементов m, m+1, m+2, ... Тип m должен быть целочисленным;

x.Step(h) – получение бесконечной последовательности элементов x, x+h, x+2h, ... Тип x должен быть числовым, тип h – совпадать с типом x или приводиться нему.

Любую последовательность можно зациклить, используя расширение .Cycle и это также даст бесконечную последовательность.

Одним из способов ограничения бесконечной последовательности является использование расширения .Take(n), позволяющего ограничиться первыми n членами последовательности.

```
1.Step.Take(30).Println; // вывод первых 30 натуральных чисел
100.Step(2).Take(12).Println; // вывод первых 12 трехзначных четных
// первые 9 значений последовательности, начиная от -2π с шагом π/4
(-2*Pi).Step(Pi/4).Take(9).Println;
// 11 элементов зацикленной последовательности 5 0 4 7
Seq(5, 0, 4, 7).Cycle.Take(11).Println; // 5 0 4 7 5 0 4 7 5 0 4
```

6.10.5. Генераторы на основе лямбда-выражений

Можно создать любую последовательность или массив, члены которой описываются математическим выражением.

§1. Аргумент пробегает значения 0, 1, ... n-1

SeqGen(n, **integer** -> T) – последовательность из n элементов; лямбда-выражение задает преобразование аргумента типа **integer** к некоторой функции типа T;

ArrGen(n, **integer** -> T) – то же, но создается массив.

```
SeqGen(5, i -> i); // 0,1,2, … 4 (5 целых)
ArrGen(7, i -> 2 * i + 1); // 1,3, … 13 (7 нечетных целых)
SeqGen(8, i -> 2 * i); // 0,2, … 14 (8 четных целых)
// следующий вызов даст 5 первых нечетных и 5 следующих четных
// чисел натурального ряда: 1,3,5,7,9,10,12,14,16,18
SeqGen(10, i -> i < 5 ? 2 * i + 1 : 2 * i );
```

Оператор **var** b := ArrGen(7, i -> 2 * i + 1) эквивалентен фрагменту

```
var F: integer -> integer := i -> 2 * i + 1;
var b := new integer[7];
for var i := 0 to 6 do
   b[i] := F(i); // сохраняем значение в элементе массива
```

Этот код можно немного упростить:

```
var b := new integer[7];
for var i := 0 to 6 do
   b[i] := 2 * i + 1;
```

§2. Аргумент пробегает значения m, m+1, ... m+n-1

SeqGen(n, **integer** -> T, m) – последовательность из n элементов; лямбда-выражение задает преобразование аргумента типа **integer** к некоторой функции типа T. Параметр m типа **integer** задает начальное значение аргумента;

ArrGen(n, **integer** -> T, m) – то же, но создается массив.

```
SeqGen(5, i -> i, 12); // 12,13,14,15,16
SeqGen(7, i -> 2 * i + 1, -5); // -9,-7,-5,-3,-1,1,3
ArrGen(8, i -> 2 * i, 1); // 2,4,6,8,10,12,14,16
```

§3. Рекуррентные зависимости при длине n

SeqGen(n, m, T -> T) – последовательность типа T из n элементов, начиная от значения m типа T; лямбда-выражение задает преобразование от предыдущего элемента к следующему;

ArrGen(n, m, T -> T) – то же, но создается массив;

SeqGen(n, m, k, (T, T) -> T) – последовательность типа T из n элементов, в которой заданы значения первого элемента m и второго k (оба типа T); лямбда-выражение задает преобразование от пары предыдущих элементов к следующему;

ArrGen(n, m, k, (T, T) -> T) – то же, но создается массив.

Рассмотрим программу получения и вывода значений первых десяти членов последовательности, в которой первый член равен 7.1, а каждый последующий получается путем увеличения предыдущего на 0.3. Рекуррентная формула: $a_i = a_{i-1} + 0.3$

```
## SeqGen(10, 7.1, x -> x + 0.3).Print // однострочный код
```

7.1 7.4 7.7 8 8.3 8.6 8.9 9.2 9.5 9.8

Примером последовательности, в которой текущий элемент связан с двумя предыдущими, может служить ряд чисел Фибоначчи: 1, 1, 2, 3, 5, 8, ... Здесь текущий элемент равен сумме двух предыдущих.

```
## SeqGen(13, 1, 1, (i, j) -> i + j).Print
```

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

Покажите этот код тем, кто считает, что в паскале писать длинно.

§4. Рекуррентные зависимости и условие окончания

SeqWhile(m, T -> T, T -> **boolean**) – последовательность элементов типа T, начиная от элемента со значением m. Первое лямбда-выражение задает преобразование от предыдущего элемента к следующему, ложное значение второго приводит к завершению генерации последовательности;

SeqWhile(m, k, (T, T) -> T, T -> **boolean**) – последовательность элементов типа T, в которой заданы значения первого элемента m и второго k. Первое лямбда-выражение задает преобразование от пары предыдущих элементов к текущему, ложное значение второго приводит к завершению генерации последовательности.

Эти генераторы используются только для последовательностей, поскольку невозможно описать массив с неизвестной текущей длиной. Они являются своеобразной альтернативой цикла **while**: логическое выражение во второй «лямбде» содержит такое же условие выхода, как и упомянутый цикл.

Ниже приведена программа вывода последовательности степеней двойки, не превышающих значение 628.

```
## SeqWhile(1, i -> 2 * i, i -> i <= 628).Println
```

1 2 4 8 16 32 64 128 256 512

Создадим массив, который будет содержать числа Фибоначчи, не превышающие 999. Решение простое и изящное.

```
##
var a := SeqWhile(1, 1, (i, j) -> i + j, i -> i <= 999).ToArray;
a.Print
```

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

Мы не описывали массив и не ломали голову над его размером. Сгенерировали последовательность и отправили ее в массив.

6.11. Задачи на перебор элементов

В этом разделе будут рассмотрены задачи с последовательностями и массивами, в которых производится перебор всех элементов в направлении от первого к последнему. Нужно понимать, что перебор всегда основан на цикле, даже если этот цикл скрыт в функции или методе.

Перед выбором пути решения внимательно изучите условие задачи. Практически все типовые школьные задачи для решения в Паскале до сих пор берутся из старых задачников, ориентированных на Turbo Pascal и, как следствие, на работу со статическими массивами, нумерованными от единицы. Если в условии говорится о четности (нечетности), тщательно выясните о чем именно речь: о четности значений элементов, четности порядковых номеров элементов или четности индексов элементов. При использовании динамического массива (в нем индексы следуют от нуля), элемент с первым (нечетным) номером имеет четный индекс! В статическом массиве номера элементов обычно совпадают с индексами.

Выбирая между последовательностью и массивом, будьте готовы при необходимости обосновать свой выбор. Аргументами могут быть: простота, экономия памяти, эффективность реализации, более наглядный код и т.д. Не забывайте, что исходные данные, получаемые при помощи датчика случайных чисел, а также вводимые с клавиатуры, следует сохранять только в массиве.

При работе с кортежами новичкам лучше постараться ограничиться кортежным присваиванием, потому что упаковка и распаковка последовательностей кортежей требует знаний, которых у вас пока нет.

6.11.1. Заполнение массива и вывод его элементов

Задача 6.13
Массив предназначен для хранения значений ростов двенадцати человек. С помощью датчика случайных чисел заполнить массив целыми значениями, лежащими на отрезке [163;190].

Здесь легко справляется генератор случайных чисел.

```
##
var a := ArrRandom(12, 163, 190);
a.Print
```

```
177 167 175 169 163 164 189 182 188 189 176 173
```

Можно ли написать **var** a := ArrRandom(12, 163, 190).Print ? Строго говоря, нельзя: расширение .Print возвратит последовательность, а не массив и формально будет нарушено условие задачи.

Задача 6.14
Заполнить массив из восьми элементов следующими значениями: первый элемент массива равен 37, второй – 0, третий – 50, четвертый – 46, пятый – 34, шестой – 46, седьмой – 0, восьмой – 13.

Тут решение совсем простое.

```
##
var a := |37, 0, 50, 46, 34, 46, 0, 13|;
a.Print
```

```
37 0 50 46 34 46 0 13
```

Задача 6.15
Используя датчик случайных чисел, заполнить массив из двадцати элементов неповторяющимися значениями, принадлежащими отрезку [10; 99]. Элементы массива вывести в строку, разделяя их запятой с последующим пробелом.

Эта задача сложнее предыдущих. Можно обеспечить уникальность элементов при помощи расширения .Distinct (это фильтр, пропускающий только неповторяющиеся значения) и контролировать размер получившейся последовательности, но нельзя заранее предсказать, сколько понадобится случайных значений для получения двадцати уникальных из них. Решать эту задачу можно разными способами.

Самый простой и быстрый в написании вариант состоит в том, чтобы создать последовательность из заведомо большего числа элементов (например, 40, 50 или 60), применить к ней .Distinct, удалив дубликаты и сохранить результат в массиве. А затем уменьшить размер до 20 при помощи процедуры SetLength. Создавать массив смысла нет: .Distinct все равно вернет последовательность. Кажется, что этот путь нерационален. Но давайте решим задачу несколькими способами и сравним затраты времени процессора. Заодно, вы познакомитесь с одним из вариантов проведения измерений производительности.

```
##
var a := SeqRandom(50, 10, 99).Distinct.ToArray;
SetLength(a, 20);
a.Print(', ');
```

```
12, 71, 67, 56, 22, 37, 42, 75, 73, 84, 54, 29, 28, 39, 59, 74, 40, 79, 23, 49
```

Оценить время выполнения программы позволит функция Milliseconds, возвращающая количество миллисекунд, прошедшее от запуска программы, включая вызова этой функции. Вторая функция, MillisecondsDelta возвращает временной интервал между ее вызовом и предыдущим вызовом Milliseconds или MillisecondsDelta. К сожалению, точность измерения времени невысока и составляет примерно 16 мс – время, за которое современный процессор успевает выполнить десятки тысяч команд. Поэтому «измеряемый» участок программы помещают в цикл, а полученный результат делят на количество проходов по циклу.

На моем компьютере было получено время 0.0067 мс = 6.7 мкс. Ничтожно малая величина. Но мы еще вернемся к этой задаче, чтобы показать ее решение, не требующее лишних вычислений. И поможет нам в этом список List.

```
##
var a: array of integer;
var k := 100000;
Milliseconds;
loop k do
begin
  a := SeqRandom(50, 10, 99).Distinct.ToArray;
  SetLength(a, 20);
end;
var t := MillisecondsDelta / k;
Print(t) // 0.0067
```

Теперь посмотрим, что покажет другой вариант решения, более близкий к классическому школьному. Создадим массив из 20 элементов, а затем в цикле будем вызывать датчик случайных чисел, получать очередное значение и проверять, есть ли оно в массиве. Если нет, будем заносить это значение в очередной элемент массива.

```
begin
  var a := new integer[20];
  var count := 0;
  while count < a.Length do
  begin
    var x := Random(10, 99);
    if not (x in a) then
    begin
      a[count] := x;
      count += 1;
    end
  end;
  a.Println(', ')
end.
```

Здесь код и длиннее, и запутаннее. Посмотрим, что покажет измерение производительности.

```
begin
  var a := new integer[20];
  var m := 100000;
  Milliseconds;
  loop m do
  begin
    var count := 0;
    while count < a.Length do
    begin
      var x := Random(10, 99);
      if not (x in a) then
      begin
        a[count] := x;
        count += 1;
      end
    end
  end;
  var t := MillisecondsDelta / m;
  Print(t);  // 0.0035
end.
```

Полученное время 0.0035 мс = 3.5 мкс. Можно праздновать победу – второй вариант быстрее вдвое. Быстрее на целые 3 микросекунды! Но на написание этого варианта ушло примерно минут пять, а на первый – около минуты. Стоит ли это сэкономленных 3 мкс работы процессора?

Задача 6.16

Заполнить массив двадцатью первыми натуральными числами, делящимися нацело на 13 или на 17 и принадлежащими отрезку, левая граница которого равна 300.

Простейшее решение без точечной нотации. Начав с 300, переберем все натуральные числа, отбирая лишь кратные 13 и 17 и заполним ими массив из 20 элементов. Обычный цикл.

```
##
var a := new integer[20];
var k := 0; // количество заполненных элементов
var n := 300; // начало отрезка
while k < 20 do
begin
  if (n mod 13 = 0) or (n mod 17 = 0) then
  begin
    a[k] := n;
    k += 1
  end;
  n += 1
end;
a.Print
```

306 312 323 325 338 340 351 357 364 374 377 390 391 403 408 416 425 429 442 455

Это же решение в точечной нотации займет всего лишь одну строку.

```
## 300.Step.Where(n ⟶(n mod 13 = 0) or (n mod 17 = 0)).Take(20).Print
```

6.11.2. Просмотр всех элементов массива

Задача 6.17
Дан массив целых чисел. Найти квадратный корень из абсолютной величины его минимального значения.

В подобных задачах удобно вводить количество элементов с клавиатуры и далее формировать массив, заполненный случайными числами из некоторого интервала. В условии ничего не сказано о границах интервала, но упоминание абсолютной величины подсказывает, что значения элементов могут быть и отрицательными. Проведем решение для значений, принадлежащих отрезку [-99; 99]. Не забывайте выводить значения сгенерированного массива, чтобы можно было проверить полученный результат.

```
##
var n := ReadInteger('n =');
var a := ArrRandom(n, -99, 99);
a.Println;
Sqrt(Abs(a.Min)).Print
```

```
n = 14
42 -91 23 39 16 35 38 26 -63 -88 -43 3 8 21
9.53939201416946
```

Хорошей программистской привычкой является оценивать полученные результаты. В данном случае минимум был равен -91, программа вывела значение, несколько превышающее 9.5. Квадрат 9.5 равен 90.25 и это значение достаточно близко к 91. Такая быстрая проверка позволяет убедиться, что программа отработала правильно.

Задача 6.18
Дан массив натуральных трехзначных чисел. Вывести те его элементы, значение которых больше средней величины.

Натуральные трехзначные числа принадлежат отрезку [100; 999]. Найдем среднее значение в массиве из n случайных чисел, при помощи фильтра выделим нужные элементы и отправим их на вывод.

```
##
var n := ReadInteger('n =');
var a := ArrRandom(n, 100, 999);
a.Println;
var m := a.Average;
a.Where(t -> t > m).Print
```

```
n = 10
393 212 508 543 861 999 222 791 816 253
861 999 791 816
```

Задача 6.19
С клавиатуры вводятся целые числа. Их общее количество неизвестно, но известно, что в конце вводится ноль, который не подлежит обработке. Найти отдельно суммы четных и нечетных чисел.

Неизвестное количество чисел невозможно сохранить в массиве, поскольку нельзя создать массив неизвестного размера. Но можно организовать ввод последовательности с условием завершения, а затем сохранить ее в массиве. Массив необходим лишь потому, что у нас нет средств, позволяющих одновременно вычислять две суммы по разным условиям. Получается короткое, но не самое эффективное решение из-за хранения данных в массиве.

```
##
var a := ReadSeqIntegerWhile(t -> t <> 0).ToArray;
Println('Сумма нечетных', a.Where(t -> t.IsOdd).Sum);
Println('Сумма четных', a.Where(t -> t.IsEven).Sum)
```

```
36 -35 17 19 -28 16 0
Сумма нечетных 1
Сумма четных 24
```

Хотите более эффективное решение без хранения данных в массиве? Тогда добро пожаловать в цикл!

```
##
var (СуммаНечетных, СуммаЧетных) := (0, 0);
var s := ReadSeqIntegerWhile(t -> t <> 0);
foreach var n in s do
  if n.IsOdd then
    СуммаНечетных += n
  else
    СуммаЧетных += n;
Println('Сумма нечетных', СуммаНечетных);
Println('Сумма четных', СуммаЧетных)
```

Последовательность *s* ленива, поэтому ввод физически начинается лишь в теле цикла.

Задача 6.20

В массиве хранится информация о массе каждого из 30 предметов, загружаемых в грузовой автомобиль, грузоподъемность которого известна. Определить, не превышает ли общая масса всех предметов грузоподъемность автомобиля.

Считая массу предмета целым числом, приходим к необходимости создать массив типа **integer** с длиной 30. Последовательность тут не подходит, поскольку мы будем использовать датчик случайных чисел. Грузоподъемность введем с клавиатуры, а массу каждого предмета сформируем в виде случайных значений, принадлежащих отрезку [5, 50]. За единицу массы примем килограммы.

```
##
var p := ReadInteger('Грузоподъемность автомобиля:');
var a := ArrRandom(30, 5, 50);
var s := a.Println.Sum; // суммарная масса
if s > p then
  Print('Автомобиль перегружен')
else
  Print('Допустимая загрузка')
```

```
Грузоподъемность автомобиля: 1000
41 15 31 14 40 5 35 23 38 42 24 29 27 24 47 47 17 46 24 31 38 6 19
38 37 17 29 10 30 44
Допустимая загрузка
```

Можно было указать вместо оператора ***var*** *s := a.Println.Sum;* два отдельных оператора, отделив вывод от суммирования.

```
a.Println;
var s := a.Sum;
```

Задача 6.21

Дан массив из 10 вещественных чисел. Верно ли, что количество его положительных элементов не превышает 5?

Выберем промежуток значений элементов [-10; 10) – для демонстрации работы программы этого вполне достаточно.

```
##
var a := ArrRandomReal(10, -10, 10);
a.Println;
if a.Count(t -> t > 0) <= 5 then
  Print('Верно')
else
  Print('Неверно')
```

```
7.91976752594103 -5.78276711319702 6.20851167301112 -0.63610973331896
-3.06842526098174 3.03923066381329 -3.9867647243602 -8.0016488851987
3.41978718220246 8.76144986542009
Верно
```

Задача 6.22

Известен рост каждого ученика класса в см. Рост мальчиков условно задан отрицательными числами. Верно ли, что средний рост мальчиков превышает средний рост девочек более чем на 5см?

Количество учеников в классе введем с клавиатуры. Рост сформируем с помощью датчика случайных чисел, расположенных на отрезке [152; 180], а знак будем приписывать, используя случайные числа 0 и 1, где 0 означает минус, а 1 – плюс.

```
##
var n := ReadInteger('Количество учеников:');
var a := SeqRandom(n, 152, 180)
    .Select(t -> Random(0, 1) = 0 ? -t : t).ToArray;
a.Println;
var m := a.Where(t -> t < 0).Average; // отрицательное
var f := a.Where(t -> t > 0).Average;
if -m - f > 5 then
  Print('Верно')
else
  Print('Неверно')
```

```
Количество учеников: 21
178 -178 -171 156 -152 153 -173 -178 164 -156 -165 -159 153 -178 -
178 -156 172 174 153 -164 -176
Верно
```

Самым сложным здесь было грамотно составить лямбда-выражение для проецирования .Select. Забавный случай, когда исходные данные сформировать сложнее, чем их обработать.

Задача 6.23
Найти сумму квадратов трехзначных чисел, сумма цифр которых равна тринадцати.

Задачу можно решить с циклом, но можно и с последовательностью. Сумма цифр трехзначного числа n равна n **div** 100 + n **div** 10 **mod** 10 + n **mod** 10. Сгенерируем последовательность трехзначных чисел и при помощи фильтра отберем нужные. Решение в один оператор:

```
## (100..999)
  .Where(n -> n div 100 + n div 10 mod 10 + n mod 10 = 13)
  .Sum(n -> n * n)
  .Print
```

```
21965334
```

6.11.3. Количество, удовлетворяющее условию

Расширение .Count(T -> boolean) возвращает количество элементов в последовательности, для которых истинно указанное лямбда-выражение.

Задача 6.24
Дан массив из n случайных целых чисел, принадлежащих отрезку [-99;99]. Найти и вывести количество элементов с нечетными значениями, абсолютная величина которых не превышает 50.

Принимаем с клавиатуры количество элементов *n*, формируем массив и используем расширение .Count. Почему массив, а не последовательность? В условии сказано, что дан массив.

```
##
var n := ReadInteger('Укажите длину массива:');
var a := ArrRandom(n, -99, 99);
a.Println.Count(t -> t.IsOdd and (Abs(t) <= 50)).Print
```

```
Укажите длину массива: 15
-37 37 -74 -61 4 97 -60 -8 -49 92 -7 15 -45 -62 89
6
```

6.11.4. Преобразование элементов

Задача 6.25
Среди первых 30 натуральных нечетных чисел найти отношение среднего значения суммы их кубов к квадрату среднего значения их квадратов.

Если условие задачи выглядит непонятным, его следует разобрать по частям. Даны 30 первых нечетных чисел: 1,3,5,... В этой последовательности k-й по порядку член может быть получен по формуле $a_k = 2k - 1$, где k = 1,2,3, ... 30. Нужно найти s_1 – среднее значение суммы кубов этих чисел и s_2 – среднее значение суммы их квадратов, а затем вывести отношение s_1 / s_2^2. Задачу можно решить при помощи одного цикла. Использование последовательности избавит нас от необходимости писать цикл. Не потому, что цикл – это плохо, а лишь с целью сократить программный код и сделать его нагляднее.

```
##
var s := SeqGen(30, k -> 2 * k - 1, 1); // генератор
Print(s.Select(t -> t * t * t).Average /
    Sqr(s.Select(t -> t * t).Average))
```

```
0.0374999971048733
```

Попробуем получить точное значение результата в виде простой дроби. Для этого нужно немного преобразовать формулу, чтобы получить целочисленные числитель и знаменатель.

$$s_1 = \frac{\sum_{k=1}^{n} a_k^3}{n};\ \ s_2 = \left(\frac{\sum_{k=1}^{n} a_k^2}{n}\right)^2;\ \ \frac{s_1}{s_2} = \frac{n \cdot \sum_{k=1}^{n} a_k^3}{\left(\sum_{k=1}^{n} a_k^2\right)^2};$$

```
##
var s := SeqGen(30, k -> 2 * k - 1, 1); // генератор
var s1 := 30 * s.Select(t -> t * t * t).Sum;
var s2 := Sqr(s.Select(t -> t * t).Sum);
Write(s1, '/', s2, ' = ', s1 / s2)
```

```
48573000/1295280100 = 0.0374999971048733
```

Точное значение равно 48573000/1295280100. Эту дробь можно при желании сократить, разделив числитель и знаменатель на их НОД.

```
function НОД(a, b: integer): integer;
begin
  while b <> 0 do
    (a, b) := (b, a mod b);
  Result := a
end;

begin
  var s := SeqGen(30, k -> 2 * k - 1, 1); // генератор
  var s1 := 30 * s.Select(t -> t * t * t).Sum;
  var s2 := Sqr(s.Select(t -> t * t).Sum);
  var t := НОД(s1, s2);
  Write(s1 div t, '/', s2 div t, ' = ', s1 / s2)
end.
```

```
485730/12952801 = 0.0374999971048733
```

Обратите внимание на имя переменной *t*, объявленной при вызове функции НОД. Оно уже использовалась в двух предыдущих строках в качестве формального параметра лямбда-выражений. Но поскольку вне этих выражений *t* не существует, сделанное объявление является корректным. Но если объявить переменную *t* в коде, расположенном выше лямбда-выражений, компилятор зафиксирует ошибку. Еще один довод в пользу объявления переменных там, где они нужны.

6.12. Удаление дубликатов (.Distinct)

Расширение .Distinct получает последовательность или массив и возвращает **последовательность**, состоящую только из неповторяющихся (уникальных) элементов. Это фильтр, не пропускающий дубликаты.

Приведенная ниже программа считывает вещественные числа, вводимые с клавиатуры, отбрасывает повторы и находит сумму этих чисел. Ввод нуля завершает чтение данных.

```
## ReadSeqRealWhile(t -> t <> 0).Distinct.Sum.Print
```

Точечная нотация очень наглядна и пояснений здесь не требуется.

6.13. Сортировка

В результате сортировки элементы последовательности или массива упорядочиваются в соответствии с некоторым правилом. При этом методы расширения всегда возвращают упорядоченную **последовательность**. Для массивов дополнительно имеется имеется процедура Sort(a), упорядочивающая элементы массива по неубыванию.

Алгоритмы сортировки оперируют понятием ***ключа сортировки***. Ключ сортировки – это некоторое выражение, которое задается или может быть найдено для каждого сортируемого элемента. В простейшем случае ключом сортировки служит само значение элемента, в более сложном – некоторая функция от значения и (или) местоположения элемента. Цель сортировки состоит в получении такой последовательности элементов, в которой их ключи будут расположены по возрастанию (неубыванию при наличии одинаковых клю-

чей), либо по убыванию (невозрастанию при наличии одинаковых ключей).

Для случая, когда ключами сортировки являются сами значения элементов последовательности (массива) имеются расширения:
- .Sorted и .Order – сортировка по неубыванию значений;
- .SortedDescending и .OrderDescending – сортировка по невозрастанию.

В случаях, когда требуется формировать ключи сортировки, расширения позволяют использовать лямбда-выражения:
- .OrderBy(T -> TKey) – сортировка по неубыванию ключа TKey;
- .OrderByDescending(T -> Tkey) – сортировка по невозрастанию TKey.

Заметьте: a.OrderBy(t -> -t) выполнит сортировку по невозрастанию.

Сортировка по ключу, к примеру, позволяет упорядочить последовательность кортежей, используя отдельные элементы кортежа для формирования ключа.

Задача 6.26
С клавиатуры вводятся десять натуральных чисел. Сформировать массив, в котором введенные числа будут расположены по неубыванию суммы своих цифр.

Сортировать надо значения элементов, а ключ сортировки – сумма цифр элемента. Возможны два варианта решения. Первый – создать кортеж из элемента и суммы его цифр. Второй – описать функцию нахождения суммы цифр числа и указать ее непосредственно в лямбда-выражении для формирования ключа сортировки. Второй способ короче и нагляднее.

```
function СуммаЦифр(Число: integer): integer;
begin
  Result := 0;
  while Число > 0 do
  begin
    Result += Число mod 10;
    Число := Число div 10
  end
end;
```

```
begin
  var s := ReadSeqInteger('Введите 10 натуральных чисел:', 10);
  var a := s.OrderBy(t -> СуммаЦифр(t)).ToArray;
  a.Print
end.
```

```
Введите 10 натуральных чисел: 456 3894 93 12 92345 234 8345 346 943 43
12 43 234 93 346 456 943 8345 92345 3894
```

6.14. Реверс (.Reverse)

Расширение для последовательности или массива .Reverse возвращает последовательность, в которой исходные элементы следуют в обратном порядке. Реверс массива также можно получить при помощи ***среза*** [::-1] (см. далее).

```
## ReadSeqInteger(5).Reverse.Print
```

```
1 2 3 4 5
5 4 3 2 1
```

6.15. Часть последовательности (массива)

В PascalABC.NET имеется множество расширений, позволяющих пропускать часть элементов последовательности или массива, а также выбирать некоторую часть элементов, ведя отсчет от начала или конца. В результате выборки получается одиночное значение, либо новая последовательность: .First, .First(T -> **boolean**), .FirstOrDefault, .FirstOrDefault(T -> **boolean**), .Take(n), .TakeWhile(T -> **boolean**), .TakeWhile(T -> **boolean**), .Skip(n), .SkipWhile(T -> **boolean**), .Last, .Last(T -> **boolean**), .LastOrDefault, .LastOrDefault(T -> **boolean**), .TakeLast(n), .SkipLast, .SkipLast(n).

Кроме перечисленных расширений, с массивом можно использовать так называемые ***срезы***.

Срез массива – динамический подмассив с элементами того же типа, что исходный массив. В общем случае срез конструируется при помощи указания в квадратных скобках трех выражений, приводящихся к типу **integer** и разделенных двоеточиями.

Срез массива a[m:n:h] возвращает подмассив, где индексы находятся в промежутке [m;n), т.е. элемент с индексом n **не рассматривается**. Значение h указывает шаг, с которым выбираются элементы и по умолчанию h = 1. Если m отсутствует, то полагается, что m = 0 (выборка от начала массива). Если n отсутствует, то полагается, что выборка производится до конца массива. Шаг может быть и отрицательным, тогда m > n и выборка производится в направлении к началу массива.

a[:5] – пять первых элементов массива (индексы 0..4);
a[3:] – элементы массива, начиная с a[3] и до конца;
a[2:7] – элементы массива, начиная с a[2] и заканчивая a[6];
a[::] – весь массив (эквивалентно просто a);
a[::2] – элементы массива с индексами 0, 2, 4, … (четными);
a[1::2] – элементы массива с индексами 1, 3, 5, … (нечетными);
a[4:13:3] – элементы массива с индексами 4, 7, 10;
a[13:4:-3] – элементы массива с индексами 13, 10, 7;
a[::-1] – элементы массива в обратном порядке.

Попытка указать в срезе несуществующий индекс вызовет ошибку. Чтобы этого не произошло, используют так называемые «безопасные срезы», для указания которых после имени массива ставится вопросительный знак. При этом, если указанный индекс не существует, выбирается ближайший существующий. Например, для массива *a* из 10 элементов срез *a[6:12]* вызовет ошибку, но срез *a?[6:12]* отработает, как *a[6:]*.

Можно указывать срезы, в которых отсчет ведется от конца массива, для чего перед значением индекса ставится знак ^. При этом *a[^1]* – это последний элемент массива *a*, *a[^2]* – предпоследний элемент и т.д. Элемента с индексом [^0] не существует!

a[^5:] – пять последних элементов массива a;
a[^1:^6:-1] – то же, но взятые в обратном порядке;
a[:^5] – все элементы массива a, кроме пяти последних.

Срезы могут находиться и в левой части части оператора присваивания, но количество элементов в срезе должно совпадать с их количеством в правой части этого оператора.

```
##
var a := ArrGen(10, i -> i + 1);
a[3:7] := a[6:2:-1];
a.Print // 1 2 3 7 6 5 4 8 9 10
```

Существуют также **срезы последовательностей**. Они реализуются посредством расширения .Slice:

Имя.Slice(НачальныйНомер, КонечныйНомер);
Имя.Slice(НачальныйНомер, КонечныйНомер, Шаг).

Нумерация элементов идет от нуля и значения параметров трактуется, так же, как в срезах массивов. Например, *s.Slice(3, 10, 2)* сформирует последовательность из элементов последовательности s с номерами (от нуля), равными 3, 5, 7, 9. По порядку же это будут элементы 4, 6, 8, 10.

Задача 6.27
Вывести трехзначные числа, являющиеся полными квадратами.

Можно решать эту задачу, составляя эффективный алгоритм. Первое трехзначное число – это 100, последнее – 999. Извлечем из этих чисел квадратные корни и отбросим дробную часть. Получим значения 10 и 31. Остается вывести квадраты натуральных чисел, принадлежащих отрезку [10; 31].

```
## (10..31).Select(t -> t * t).Print
```

```
100 121 144 169 196 225 256 289 324 361 400 441 484 529 576 625
676 729 784 841 900 961
```

Другой путь – сгенерировать бесконечную последовательность квадратов чисел натурального ряда, и отобрать из нее нужные элементы. Сначала пропустить элементы, меньшие 100, а затем отобрать элементы, меньшие 1000. Это не требует предварительных вычислений и вообще каких-либо размышлений.

```
##
1.Step // берем бесконечный ряд натуральных чисел
    .Select(t -> t * t) // получаем их квадраты
    .SkipWhile(t -> t < 100) // пропускаем пока меньше 100
    .TakeWhile(t -> t < 1000) // берем пока меньше 1000
    .Print
```

Первое решение эффективнее и короче, но в нем не видно первоначального условия задачи. Второе длиннее, но отражает это условие.

Задача 6.28

Дан массив, заполненный n случайными вещественными числами из промежутка [-25.93; 31.152). Найти и вывести среднее арифметическое кубических корней элементов массива, имеющих четные номера.

Задача, несмотря на свою внешнюю простоту, содержит сразу несколько подводных камней. Первая проблема связана с получением вещественных случайных чисел. Случайное вещественное число, как и все вещественные числа имеет от 15 до 16 цифр в мантиссе. С этим неудобно работать и это неудобно выводить в качестве отображения содержимого массива. Вот фрагмент такого вывода:

```
20.6056176525967 29.5592047828013 -10.9303085852565 -6.78768335227176E-05
```

А что, если данные в массиве округлить, например до трех знаков после запятой? Сформируем последовательность вещественных чисел, при помощи .Select выполним округление и полученную последовательность сохраним в массив. Получим что-то наподобие такого ряда значений:

```
-21.081 -10.814 15.925 26.71 -5.578 -14.184 -12.952 -0.777
```

Вторая проблема – извлечение кубического корня. Эта проблема уже обсуждалась в задаче 6.3.

```
##
var a := SeqRandomReal(10, -25.93, 31.152)
    .Select(t -> Round(t, 3)).ToArray;
a.Println;
a[1::2]
    .Select(t -> t = 0 ? 0.0 : Sign(t) * Abs(t) ** (1 / 3))
    .Average
    .Print
```

```
13.953 9.313 -22.498 -9.427 2.307 15.827 22.691 25.765 16.442 -18.645
0.560808572782222
```

Срез *a[1::2]* состоит из элементов с нечетными индексами, но поскольку динамический массив индексируется от нуля, нечетные индексы дают четные по порядку элементы.

Задача 6.29

Известны данные о количестве осадков, выпавших за каждый день февраля. Верно ли, что по четным числам выпало больше осадков, чем по нечетным?

Предположим, что в феврале было 28 дней и сгенерируем целочисленный массив из 28 элементов, заполненных случайными числами, принадлежащими отрезку [0, 100]. А затем используем срезы.

```
##
var Осадки := ArrRandom(28, 0, 100);
Осадки.Println;
if Осадки[1::2].Sum > Осадки[0::2].Sum then
  Print('Верно')
else
  Print('Неверно')
```

```
32 50 40 33 36 25 31 92 37 59 9 43 67 61 64 100 33 51 99 74 68 80 62 100 82 13 45 82
Верно
```

Как только вы привыкнете, что нечетные элементы получаются при отсчете от четного нуля через один, а четные – при отсчете от нечетной единицы через один, подобные решения будут составляться буквально за минуту.

Задача 6.30

В массиве хранятся сведения о количестве осадков, выпавших за каждый день июня. Определить общее количество осадков, выпавших за каждую декаду этого месяца.

Сгенерируем целочисленный массив из 30 элементов, заполненных случайными числами, принадлежащими отрезку [0, 100]. Декада – группа из 10 дней, т.е. интервалы дней 1...10, 11...20 и 21...30. Срезы снова придут к нам на помощь.

```
##
var Осадки := ArrRandom(30, 0, 100);
Осадки.Println;
Print(Осадки[:10].Sum, Осадки[10:20].Sum, Осадки[20:].Sum)
```

```
73 33 37 67 83 97 14 39 31 85 39 35 94 89 94 4 44 31 33 76 1 89 47 45 71 13 27 47 50 7
559 539 397
```

При задании среза в границах [a:b], выбираются индексы от a до b-1, т.е. само b не включается и математически это можно записать как [a;b). Поэтому [:10] – это индексы от начала и включая 9, т.е. элементы с первого по десятый. Аналогично [10:20] – индексы от 10 и включая 19 (элементы с 11 по 20) и [20:] – индексы от 20 до конца (элементы с 21 по последний).

Казалось бы, почему не сделать индексы динамического массива от единицы, а не от нуля? Математики возражают, что неудобно будет программировать формулы наподобие $a_0 = 1$; $a_1 = 3$; $a_n = 2a_{n-2} - 3a_{n-1}$.

Задача 6.15 (повтор)
Используя датчик случайных чисел, заполнить массив из двадцати элементов неповторяющимися числами, принадлежащими отрезку [10; 99]. Элементы массива вывести в строку, разделяя их запятой с последующим пробелом.

Мы возвращаемся к этой задаче, чтобы рассмотреть более эффективное решение в точечной нотации. Прежний код был таким:

```
##
var a := SeqRandom(50, 10, 99).Distinct.ToArray;
SetLength(a, 20);
a.Print(', ')
```

Мы генерировали массив вдвое большей длины, а затем переопределяли его размер. Но теперь мы умеем получать часть последовательности и можно сделать решение короче, нагляднее и эффективнее.

```
##
var a := SeqRandom(50, 10, 99).Distinct.Take(20).ToArray;
a.Print(', ')
```

Читаем строку кода: сгенерировать последовательность из 40 целых чисел из отрезка [10;99], удалить из нее дубликаты, взять 20 первых элементов, поместить в массив и назвать его *a*.

6.16. Разбиение

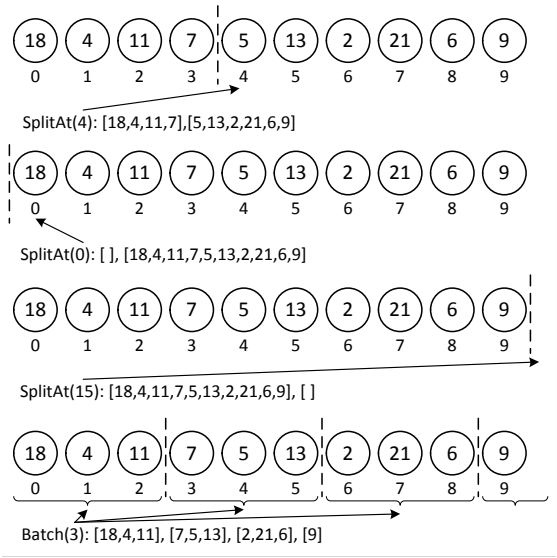
Последовательность и массив можно разбивать на две и более подпоследовательности. При этом формируется новая последовательность, каждый элемент которой является подпоследовательностью.

Разбиение можно производить **в кортеж** из двух подпоследовательностей так, что элемент с указанным номером (индексом в случае массива) попадет в начало второй подпоследовательности. При указании несуществующего номера одна из подпоследовательностей будет пустой. Другой вариант разбиения – указать логическое выражение, истинность которого определит условие попадание элемента в первую подпоследовательность, а ложность – условие попадания во вторую. Последовательность также можно превратить **в последовательность** из нескольких подпоследовательностей указанной длины. Если длина последовательности не кратна указанной длине, последняя подпоследовательность будет короче остальных.

.SplitAt(n) – разбиение на две подпоследовательности так, что n-й по порядку элемент останется в первой из них;

.Batch(k) – разбиение на подпоследовательности по k элементов в каждой;

.Partition(T -> **boolean**) – разбиение на две подпоследовательности так, что первая из них будет содержать элементы, для которых лямбда-выражение истинно, а вторая – остальные элементы. Расширение использует двухпроходный алгоритм, поэтому оно не годится для последовательностей, получаемых путем ввода с клавиатуры или от генератора случайных чисел.



Задача 6.30 (повтор)
В массиве хранятся сведения о количестве осадков, выпавших за каждый день июня. Определить общее количество осадков, выпавших за каждую декаду этого месяца.

Решим эту задачу, осуществив разбивку на декады. Если использовать расширение .Batch(10), мы получим последовательность, состоящую из трех подпоследовательностей. Как ее обработать? Можно использовать проецирование .Select, которое предоставит доступ к каждой подпоследовательности и применить расширение .Sum.

```
##
var Осадки := ArrRandom(30, 0, 100);
Осадки.Println;
Осадки.Batch(10).Select(Декада -> Декада.Sum).Print
```

Мне такое решение нравится больше уже хотя бы тем, что тут явно фигурирует понятие декады и не нужно беспокоиться по поводу возможности запутаться в значениях индексов срезов.

Здесь *Декада* – имя, которое при проецировании будет поочередно присваиваться каждой подпоследовательности и мы преобразовы-

ваем эту подпоследовательность к сумме ее членов. В результате получается также последовательность, но уже из трех элементов, представляющих собой суммы.

Задача 6.31
На вход программы поступает последовательность вещественных чисел, ограниченная нулем (ноль в последовательность не входит). Вывести эти числа так, чтобы вначале шли все положительные числа в порядке невозрастания, а затем все отрицательные в порядке неубывания.

Введем последовательность, ограниченную условием, сохраним ее в массив, а затем разобьем его на две последовательности по знаку при помощи расширения .Partition. Необходимость сохранения в массив следует из описания этого расширения, где говорится о двухпроходном алгоритме. Полученные последовательности упорядочим и соединим в одну, которую затем выведем.

```
##
var a := ReadSeqRealWhile(t -> t <> 0).ToArray;
var (s1, s2) := a.Partition(t -> t > 0);
(s1.OrderDescending + s2.Order).Print
```

```
2.5 -0.13 4.9 3.6123 -5 9.1 18 0
18 9.1 4.9 3.6123 2.5 -5 -0.13
```

Кортежное присваивание позволило дать каждой подпоследовательности собственное имя. При этом сохраняется ленивость, т.е. фактическое обращение к массиву *a* произойдет только при сортировках. В коде программы s1 – подпоследовательность, содержащая положительные элементы, s2 – подпоследовательность, содержащая отрицательные элементы.

Рассмотрим, как приведенное выше решение реализуется при помощи классической тройки базовых алгоритмических структур – следования, ветвления и цикла. Сортировки используем простые – методом пузырька.

```
begin
  var limit := 1000; // предельный размер подпоследовательности
  var s1 := new real[limit]; // положительные
  var s2 := new real[limit]; // отрицательные
  var (p1, p2) := (0, 0); // индексы первых свободных элементов
  // ввод и формирование двух массивов
  repeat
    var k := ReadReal;
    case Sign(k) of
      -1: begin
            s2[p2] := k;
            p2 += 1
          end;
      0: break;
      1: begin
           s1[p1] := k;
           p1 += 1
         end
    end
  until False;
  // сортировки и вывод
  for var i := 0 to p2 - 2 do
    for var j := i + 1 to p2 - 1 do
      if s2[i] > s2[j] then
        Swap(s2[i], s2[j]);
  for var i := 0 to p1 - 2 do
    for var j := i + 1 to p1 - 1 do
      if s1[i] < s1[j] then
        Swap(s1[i], s1[j]);
  for var i := 0 to p1 - 1 do
    s1[i].Print;
  for var i := 0 to p2 - 1 do
    s2[i].Print;
end.
```

После сравнений этих двух решений кажется наивно-смешной просьба все еще изучающих в школах «турбопаскали»: - А нельзя ли написать попроще? Конечно, речь о первом решении, потому что ожидалось увидеть что-то в стиле второго. Тридцать две строки кода против четырех – это «попроще»? Так мы еще использовали ReadReal и Swap.

Один раз учитель может развернуть компактный современный код PascalABC.NET в целях демонстрации алгоритмов. Но расписывать вот так каждую задачу? Можно один раз продемонстрировать, как

наши далекие предки добывали огонь трением, но каждый раз так получать пламя? Представьте, что на космодроме целая бригада сидит кружком вокруг ракеты и вертит палочки в кучках сухого мха, разжигая ее двигатели. Смешно? Не смешнее, чем десятки раз переписывать один и тот же детальный код пузырьковой сортировки и называть это обучением.

Задача 6.32

Оценки, полученные спортсменом в соревнованиях по фигурному катанию (в баллах), хранятся в массиве из 21 элемента. В первых семи элементах записаны оценки по обязательной программе; в следующих семи – по короткой программе и в остальных – по произвольной. По какому виду программы спортсмен показал лучший результат?

Здесь нужно получить суммы баллов по трем подпоследовательностям и сравнить их. Баллы в современном фигурном катании выставляются в интервале от 0 до 10 с шагом 0.25.

Придется использовать вещественный тип данных. Чтоб не вводить 21 значение вручную, используем датчик случайных чисел. А вот как округлить эти числа с точностью до 0.25? Воспользуемся целыми числами из отрезка [0; 40] и разделим их на 4.

```
##
var a := SeqRandom(21, 0, 40).Select(t -> t / 4).ToArray;
var r := a.Println.Batch(7).Select(t -> t.Sum).ToArray;
if (r[0] > r[1]) and (r[0] > r[2]) then
  Print('Обязательная программа')
else if (r[1] > r[0]) and (r[1] > r[2]) then
  Print('Короткая программа')
else
  Print('Произвольная программа')
```

```
7.5 0.75 5 9.5 3.75 1.25 2.25 9.75 8.25 3.25 9 6.75 6 9.5 9 4 8.25 3 5.25 4.5 1.5
Короткая программа
```

Здесь *a* – исходный массив, значений которого выведены. Массив *r* содержит суммы баллов по отдельным видам катания. Работа расширения .Batch упоминается выше при разборе повторного решения задачи 6.30.

6.17. Пары последовательностей (массивов)

Мы рассмотрели методы расширения, разбивающие последовательности и массивы на две и более частей. А теперь научимся объединять пару последовательностей (массивов) или последовательность и массив в одну последовательность. Ниже именами p и q обозначаются последовательности (массивы), участвующие в объединении.

6.17.1. Соединение и объединение

Соединение p + q дает последовательность (или массив, если p и q массивы), в которой элементы из q располагаются после элементов из p. Элементы полученной последовательности следуют в том же порядке, в котором они находились в исходных последовательностях.

Если рассмотреть совокупность элементов последовательности или массива как математическое множество, для двух последовательностей можно получить их объединение как множеств. Фактически множество – разновидность последовательности, в которой нет элементов с одинаковым значением.

Расширение p.Union(q) **объединяет** p и q, создавая новую **последовательность**, содержащую все элементы p и q **без повторения** их значений. В отличие от множества, в котором порядок следования элементов несущественен, элементы полученной последовательности следуют в том же порядке, в котором они находились в исходных последовательностях.

6.17.2. Пересечение и разность

Расширение p.Intersect(q) возвращает последовательность, являющуюся **пересечением** p и q. Пересечение содержит только те элементы, которые имеются и в p, и в q. Полученная последовательность не будет содержать повторяющихся элементов, а сами элементы следуют в том же порядке, в котором они находились в p.

Расширение p.Except(q) возвращает **разность**, состоящую из элементов p, из которых исключены элементы, присутствующие в q. Полученная последовательность не будет содержать повторяющихся

элементов, а сами элементы следуют в том же порядке, в котором они находились в p и q.

6.17.3. Решение задачи

Задача 6.33
В массивах a, b и c хранятся оценки, полученные учеником за четверть по разным предметам. Выяснить: а) верно ли, что ученик получил двойку по каждому из предметов и при этом есть предмет, по которому не было получено ни одной пятерки? б) верно ли, что ученик получил весь набор оценок – от двойки до пятерки?

Длину массивов выберем произвольно, значения будем задавать вводом с клавиатуры. Ответ на вопрос «а» даст пересечение массивов, на вопрос «в» - объединение.

```
begin
  var a := ReadArrInteger('Математика (8 оценок):', 8);
  var b := ReadArrInteger('Русский язык (9 оценок):', 9);
  var c := ReadArrInteger('Информатика (6 оценок):', 6);
  var Пересечение := a.Intersect(b).Intersect(c);
  if (2 in Пересечение) and (not (5 in Пересечение)) then
    Println('Верно')
  else
    Println('Неверно');
  if Seq(2,3,4,5).Except(a.Union(b).Union(c)).Count = 0 then
    Print('Верно')
  else
    Print('Неверно')
end.
```

```
Математика (8 оценок): 3 4 3 5 4 3 4 4
Русский язык (9 оценок): 4 4 3 5 4 3 4 2 4
Информатика (6 оценок): 5 3 4 4 2 3
Неверно
Неверно
```

Выражение *2 **in** Пересечение* вернет True, если среди элементов *Пересечение* найдется двойка. Выражение ***not** (5 **in** Пересечение)* вернет True, если среди элементов *Пересечение* не найдется пятерка. Функция *Seq(2, 3, 4, 5)* создает последовательность всех возможных оценок, а Except исключит из нее оценки, полученные объединением трех исходных массивов. Если в результате получится пустая после-

довательность (расширение .Count вернет ноль), можно утверждать, что в объединении присутствовали все четыре оценки.

Посмотрим, как решить эту задачу при без объединений, пересечений и разности.

```
function Все2иНет5(a: array of integer): (boolean, boolean);
begin
  var (Есть2, Есть5) := (False, False);
  foreach var n in a do
  begin
    if n = 2 then
      Есть2 := True;
    if n = 5 then
      Есть5 := True;
  end;
  Result := (Есть2, not Есть5)
end;

function НашлиВсе(a: array of integer): boolean;
begin
  Result := 2 in a;
  if Result then
    Result := 3 in a;
  if Result then
    Result := 4 in a;
  if Result then
    Result := 5 in a
end;
```

```
begin
  var a := ReadArrInteger('Математика (8 оценок):', 8);
  var b := ReadArrInteger('Русский язык (9 оценок):', 9);
  var c := ReadArrInteger('Информатика (6 оценок):', 6);
  var (Есть2, Нет5) := Все2иНет5(a);
  var (Есть2Доп, Нет5Доп) := Все2иНет5(b);
  (Есть2, Нет5) := (Есть2 and Есть2Доп, Нет5 and Нет5Доп);
  (Есть2Доп, Нет5Доп) := Все2иНет5(c);
  (Есть2, Нет5) := (Есть2 and Есть2Доп, Нет5 and Нет5Доп);
  if Есть2 and Нет5 then
    Println('Верно')
  else
    Println('Неверно');
  var ЕстьВсе := НашлиВсе(a);
  if ЕстьВсе then
    ЕстьВсе := ЕстьВсе and НашлиВсе(b);
  if ЕстьВсе then
    ЕстьВсе := ЕстьВсе and НашлиВсе(c);
  if ЕстьВсе then
    Print('Верно')
  else
    Print('Неверно')
end.
```

Правильно написать такой код с первого раза под силу не каждому. Хорошо, что массивов только три. И хорошо, что подобные задачи не включены в ЕГЭ для написания кода без использования компьютера.

6.17.4. Чередование элементов

Расширение *.Interleave* позволяет чередовать элементы двух, трех и четырех последовательностей или массивов таким образом, что сначала идут все первые элементы, затем – все вторые и т.д.

```
##
var s1 := SeqGen(6, p -> 2 * p * 2 + 3).Println;
var a2 := ArrGen(5, p -> 3 * p - 2).Println;
var s3 := SeqGen(4, p -> p - 8).Println;
s1.Interleave(a2, s3).Print
```

```
3 7 11 15 19 23
-2 1 4 7 10
-8 -7 -6 -5
3 -2 -8 7 1 -7 11 4 -6 15 7 -5
```

Если последовательности имеют различную длину, то из каждой последовательности выбирается столько элементов, сколько их в самой короткой последовательности.

6.18. Соединение элементов (.Zip)

Расширение соединяет элементы двух последовательностей или массивов с одинаковыми порядковыми номерами, выполняя с ними операцию, заданную лямбда-выражением. В операции участвует количество элементов, определяемое по более короткой последовательности. В результате получается последовательность, каждый элемент которой представляет собой произведение элементов исходных последовательностей.

```
##
var s1 := SeqGen(6, p -> 2 * p * 2 + 3).Println;
var s2 := SeqGen(5, p -> 3 * p - 2).Println;
s1.Zip(s2, (p, q) -> p * q).Print
```

```
3 7 11 15 19 23
-2 1 4 7 10
-6 7 44 105 190
```

Задача 6.34

В массиве x хранятся целочисленные абсциссы n точек; в массиве y хранятся целочисленные ординаты этих точек. Найти величину расстояния от центра координат до самой удаленной точки и вывести ее с тремя знаками в дробной части.

«Классический» вариант решения – цикл, в котором обрабатываются элементы обоих массивов. Можно искать максимальный квадрат расстояния $x_i^2 + y_i^2$, а затем извлечь из него квадратный корень. Посмотрим, как выполнить решение при помощи расширения .Zip.

```
##
var n := ReadInteger('n =');
var x := ArrRandom(n, -50, 50);
x.Println;
var y := ArrRandom(n, -50, 50);
y.Println;
Write(Sqrt(x.Zip(y, (p, q) -> p * p + q * q).Max):0:3)
```

```
n = 15
21 -8 23 -32 -34 -42 15 -17 4 43 -14 40 -2 46 -7
44 -4 5 5 15 33 -25 -23 9 8 19 -24 46 9 -16
53.413
```

6.19. Равенство последовательностей

Две последовательности можно поэлементно сравнить между собой. Последовательности p и q считаются равными, если для всех p_i и q_i выполняется условие $p_i=q_i$.

Расширение p.SequenceEqual(q) возвращает True, если последовательности равны и False в противном случае. Для массивов имеется расширение p.ArrEqual(q) и функция ArrEqual(p, q), также возвращающие True при равенстве массивов p и q.

А нельзя ли написать просто p = q ? Можно. Но при этом будет проверено, указывают ли ссылки p и q на один и тот же объект в памяти, но никак не поэлементное равенство.

6.20. Поиск в последовательности

Последовательность не хранится и не имеет индексов, поэтому любой поиск в ней связан с последовательным получением каждого элемента. Поиск ведется от начала последовательности и выполняется либо до полного удовлетворения условий поиска, либо до конца последовательности. Элементы последовательности нумеруются от нуля.

Если последовательность *s* пуста, расширение s.DefaultIfEmpty вернет одноэлементную последовательность со значением элемента по умолчанию, Имеется вариант, позволяющий задать значение по умолчанию: s.DefaultIfEmpty(значение). При этом тип значения должен быть точно таким же, как тип последовательности.

6.20.1. Элемент с указанным номером

Расширение s.ElementAt(n) выбирает из последовательности *s* элемент с номером *n*. Если элемента с таким номером нет, то выдается ошибка вида «Ошибка времени выполнения: Индекс за пределами

диапазона...» с последующим аварийным завершением работы программы. Путей решения проблемы несколько:
– предварительно проверить, чтобы номер лежал в пределах от 0 до s.Count-1;
– воспользоваться расширением s.ElementAtOrDefault(n).

Расширение s.ElementAtOrDefault(n) выбирает из последовательности *s* элемент с номером *n*, а если такого элемента нет, возвращает ***значение по умолчанию***. Значение по умолчанию – это некоторое значение, которое назначается каждому объекту в программе. Для последовательности, элементы которой имеют тип T, значением по умолчанию будет элемент типа T, который в свою очередь будет иметь значение по умолчанию, принятое для этого типа. Значениями по умолчанию, в частности, инициализируются переменные, если для них не задано начального значения. Для числовых последовательностей метод s.ElementAtOrDefault при неуспешном поиске элемента вернет элемент со значением 0. Если элемент с таким значением в последовательности уже имеется, это может внести путаницу.

```
##
var s := Seq(2.3, 5.951, 9.4, -3.001).Println(' | ');
s.ElementAtOrDefault(2).Println;
s.ElementAtOrDefault(-2).Println;
s.ElementAtOrDefault(10).Println;
```

```
2.3 | 5.951 | 9.4 | -3.001
9.4
0
0
```

6.20.2. Элементы, удовлетворяющие условию

Никаких отдельных средств для поиска таких элементов нет. Пользуемся фильтрацией **Where**.

```
##
var s := Seq(3, 18, 6, -8, 14, 41, 5, 0, 13, 6, 1);
s.Where(t -> Abs(t) > 6).Print   // 18 -8 14 41 13
```

6.21. Наличие элементов в последовательности

6.21.1. Есть ли такой элемент?

Расширение s.Contains(a) возвращает True, если в последовательности *s* имеется хотя бы один элемент со значением *a* и False в противном случае.

```
## (-5..8).Println.Contains(4).Print
```

```
-5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8
True
```

Аналогичный результат дает использование логического выражения

```
Элемент in Последовательность
```

Возвращается True, если в последовательности имеется хотя бы один элемент с указанным значением и False в противном случае.

```
##
var s := -5..8;
s.Println; // -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8
(4 in s).Println // True
```

6.21.2. Есть ли элемент, удовлетворяющий условию?

Расширение s.Any(T -> **boolean**) возвращает True, если хотя бы для одного элемента последовательности указанное лямбда-выражение будет истинным. В варианте s.Any возвращается True, если послежовательность непустая и False в противном случае.

```
##
var s := -5..8;
s.Println.Any(r -> r > 10).Println;
s.Any(r -> r < -3).Println;
```

```
-5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8
False
True
```

6.21.3. Все ли элементы удовлетворяют условию?

Расширение s.All(T -> **boolean**) возвращает True, если для каждого элемента последовательности *s* лямбда-выражение истинно. Напри-

мер, так можно проверить, все ли элементы последовательности положительны, перед тем как вычислять выражение с логарифмом.

```
##
loop 9 do
  ArrRandom(10, -5, 50).All(p -> p > 0).Print
Println
```

```
False False True False True False False False False
```

Здесь девять раз генерируется последовательность из десяти случайных целых чисел, принадлежащих отрезку [-5; 50]. Для каждой последовательности выводится True, если все значения в ней положительны и False в противном случае:

6.22. Поиск в массиве

В отличие от последовательности, элементы массива хранятся и имеют индексы, поэтому поиск может начинаться от любого элемента и вестись в направлении конца, либо начала массива. Индексы в массиве, как вы помните, нумеруются от нуля.

Конечно, для массива приемлемо все то, что рассмотрено для последовательностей в предыдущем разделе. Но, как правило, поиск с использованием индексов выполняется намного эффективнее.

Задача 6.35

Известен рост каждого ученика класса в см. Рост мальчиков условно задан отрицательными числами. Верно ли, что рост мальчиков отличается от их среднего роста не более, чем на 9%, а среди девочек есть хотя бы одна, чей рост отличается от среднего роста девочек более чем на 8%?

Это несколько измененное условие задачи 6.22. Количество учеников в классе введем с клавиатуры. Рост сформируем с помощью датчика случайных чисел, расположенных на отрезке [152; 180], а знак будем приписывать, используя случайные числа 0 и 1, где 0 означает минус, а 1 – плюс.

```
##
var n := ReadInteger('Количество учеников:');
var a := SeqRandom(n, 152, 180)
    .Select(t → Random(0, 1) = 0 ? -t : t).ToArray;
a.Println;
var (sm, sf) := a.Partition(t → t < 0); // мальчики, девочки
sm := sm.Select(t → -t);
var am := sm.Average; // мальчики, средний рост
var af := sf.Average;  // девочки, средний рост
if sm.All(t → Abs(t - am) <= 0.09 * am) and
    sf.Any(t → Abs(t - af) >= 0.08 * af) then
  Print('Верно')
else
  Print('Неверно')
```

```
Количество учеников: 21
156 -176 169 -159 153 180 174 -159 155 -175 -156 -179 164 -166
-170 -166 -171 -163 156 153 -162
Верно
```

6.22.1. Поиск значений элементов

Расширение a.Find(T -> **boolean**) возвращает значение первого из элементов массива *a*, для которого истинно указанное лямбда-выражение. Если такого элемента нет, возвращается нулевое значение для типа, который имеют элементы массива.

Расширение a.FindAll(T -> **boolean**) возвращает **массив** тех значений элементов *a*, для которых истинно указанное лямбда-выражение. Если ни одного элемента не найдено, возвращается массив нулевой длины.

```
##
var a := ArrRandom(20, 0, 9);
a.Println;
a.Find(p -> p > 6).Println;
a.FindAll(p -> p > 6).Println
```

```
2 9 1 4 6 0 6 8 5 2 2 9 4 6 1 7 8 9 5 5
9
9 8 9 7 8 9
```

6.22.2. Поиск индексов элементов

В PascalABC.NET имеется набор операций, позволяющий выполнять ряд действий, в которых участвуют индексы. Можно определить индексы элементов, удовлетворяющих тем или иным условиям с тем, чтобы затем выбирать эти элементы или изменять их значения.

§1. Индекс максимального элемента в массиве

a.IndexMax – индекс первого максимального элемента в массиве *a*;
a.IndexMax(k) – индекс первого максимального элемента, начиная его поиск с элемента, имеющего индекс *k*;
a.LastIndexMax – индекс последнего максимального элемента;
a.LastIndexMax(k) – индекс последнего максимального элемента, рассматривая элементы от начала массива и ограничивая поиск элементом с индексом k.

```
##
var a := ArrRandom(10, 0, 9);
a.Println;
a.IndexMax.Println; // 1й max
a.IndexMax(5).Println; // 1й max от элемента с индексом 5
a.LastIndexMax.Println; // последний max
a.LastIndexMax(7).Println; // последний max с просмотром 8
```

```
7 0 1 8 7 8 2 3 7 8
3
5
9
5
```

Задача 6.36
Дан массив, заполненный 20 случайными целыми числами, принадлежащими отрезку [-50;75). Составить новый массив, содержащий элементы с нечетными значениями, расположенными перед первым максимальным элементом и элементы с четными значениями, расположенными за ним.

Для решения задачи вначале найдем индекс максимального элемента. Далее разобьем массив на две подпоследовательности по позиции, предшествующей этому элементу. Из первой подпоследовательности возьмем элементы с нечетными значениями. Во второй подпоследовательности пропустим первый элемент (найденный ранее макси-

мум), а из оставшихся выберем элементы с четными значениями. Полученные последовательности соединим и превратим в массив.

```
##
var a := ArrRandom(20, -50, 75);
a.Println;
var (s1, s2) := a.SplitAt(a.IndexMax);
var b :=
    (s1.Where(t -> t.IsOdd) + s2.Skip(1).Where(t -> t.IsEven))
    .ToArray;
b.Println
```

```
4 -31 -38 -18 1 54 16 -28 58 0 70 13 74 30 11 59 -25 50 -40 70
-31 1 13 30 50 -40 70
```

Возьмите на заметку: индексы в динамических массивах начинаются с нуля, а нумерация элементов последовательности (и массива) при разбиении ведется от единицы. Оператор ***var*** *(s1, s2) := ...* – ленивый. Элементы последовательностей *s1* и *s2* будут вычисляться только в следующем операторе, т.е. там, где они понадобятся.

Можно ли вместо последовательностей использовать срезы? В принципе – конечно. Сначала найти ***var*** *i := a.IndexMax*, а затем записать ***var*** *(s1, s2) := (a[:i], a[i+1:])*. Но это менее наглядно. С первого взгляда не очевидно, что разбиение ведется по максимальному элементу. К тому же, *s1* и *s2* станут массивами, которые будут храниться.

§2. Индекс минимального элемента в массиве

a.IndexMin – индекс первого минимального элемента в массиве *a*;

a.IndexMin(k) – индекс первого минимального элемента, начиная его поиск с элемента, имеющего индекс *k*;

a.LastIndexMin – индекс последнего минимального элемента;

a.LastIndexMin(k) – индекс последнего минимального элемента, рассматривая элементы от начала массива и ограничивая поиск элементом с индексом k.

§3. Индекс элемента, удовлетворяющего условию

a.FindIndex(T -> boolean) – индекс первого из элементов массива *a*, для которого будет истинным указанное в качестве аргумента лямбда-выражение. Если такой элемент не найден, возвращается -1.

a.FindIndex(k, T -> boolean) – индекс первого из элементов, для которого будет истинным указанное в качестве аргумента лямбда-выражение, просматривая элементы начиная с имеющего индекс *k*. Если такой элемент не будет найден, возвращается -1.

a.FindLastIndex(T -> boolean) – индекс последнего из элементов, для которого будет истинным указанное в качестве аргумента лямбда-выражение. Если такой элемент не будет найден, возвращается -1.

a.FindLastIndex(k, T -> boolean) – индекс последнего из элементов, для которого будет истинным указанное в качестве аргумента лямбда-выражение, просматривая элементы от начала и не дальше элемента, имеющего индекс *k*. Если такой элемент не будет найден, возвращается -1.

```
##
var a := ArrRandom(20, 0, 9);
a.Println;
a.FindIndex(5, p -> p >= 7).Println;
a.FindLastIndex(8, p -> p = 3).Println
```

```
9 2 8 1 2 6 1 0 2 1 2 2 8 9 0 8 0 3 2 0
12
-1
```

§4. Индексы элементов, удовлетворяющих условию

Расширение a.Indices(T -> **boolean**) возвращает последовательность индексов тех элементов массива a, для которых будет истинным указанное в качестве аргумента лямбда-выражение. Если таких элементов не найдено, возвращается пустая последовательность.

При необходимости получить индексы всех элементов массива можно указать просто a.Indices. Например, вот так можно записать заголовок цикла для поочередного доступа ко всем элементам массива по их индексу:

```
foreach var i in a.Indices do
```

Расширение a.Indices((T, i) -> **boolean**) возвращает последовательность индексов тех элементов массива a, для которых будет истинным указанное в качестве аргумента лямбда-выражение, учитываю-

щее эти индексы i. Если таких элементов не найдено, возвращается пустая последовательность.

```
##
var a := ArrRandom(10, 0, 9);
a.Println;
a.Indices(q -> q.InRange(4, 7)).Println;
a.Indices((p, i) -> (p >= 5) and i.IsOdd).Println
```

```
1 4 6 5 1 3 7 7 8 6
1 2 3 6 7 9
3 7 9
```

В приведенном примере сначала отыскиваются индексы элементов со значениями в интервале [4;7], а затем нечетные индексы элементов со значением, не меньшим пяти.

Задача 6.37

Дан целочисленный массив. Если в нем есть элементы, большие заданного числа m, то напечатать все элементы, следующие за третьим из таких элементов. В противном случае вывести соответствующее сообщение.

Положим, что числа принадлежат отрезку [-99;99] – этого вполне достаточно, чтобы протестировать написанную программу. Получим при помощи Indices набор индексов элементов, удовлетворяющих условию и если в нем имеется третий элемент, он укажет значение индекса, которое требуется для организации вывода.

```
##
var n := ReadInteger('n =');
var a := ArrRandom(n, -99, 99);
a.Println;
var m := ReadInteger('m =');
var b := a.Indices(t -> t > m).ToArray;
if b.Length >= 3 then
  a?[b[2] + 1:].Print
else
  Print('Нет элементов для вывода')
```

```
n = 20
-94 -78 88 10 -42 -3 -10 82 -35 -89 -82 4 53 -69 -59 54 29 77 -59 94
m = 50
-69 -59 54 29 77 -59 94
```

Безопасный срез страхует от случая, если третий элемент, удовлетворяющий условию, окажется в массиве последним. В этом случае вывод будет пустым.

§5. Индекс элемента по значению

Расширение a.IndexOf(x) возвращает индекс первого из элементов массива a, равного x или -1, если такой элемент не найден.

Расширение a.IndexOf(x, k) возвращает индекс первого из элементов массива a, равного x, просматривая элементы начиная с имеющего индекс k. Если такой элемент не будет найден, возвращается -1.

Расширение a.LastIndexOf(x) возвращает индекс последнего из элементов массива a, равного x или -1, если такой элемент не найден.

Расширение a.LastIndexOf(x, k) возвращает индекс последнего из элементов массива a, равного x, просматривая элементы от начала и не дальше элемента, имеющего индекс k. Если такой элемент не будет найден, возвращается -1.

```
##
var a := ArrRandom(20, 0, 9);
a.Println;
a.IndexOf(5, 7).Println;
a.LastIndexOf(8, 3).Println
```

```
4 9 1 1 2 9 4 4 0 3 2 1 6 5 8 0 7 4 2 0
13
-1
```

§6. Бинарный поиск индекса элемента по значению

Расширение a.BinarySearch(x) производит бинарный поиск индекса элемента со значением x в массиве a, **предварительно упорядоченном по неубыванию**. Если элемент не найден, возвращается **произвольное** отрицательное значение. Если элементов с искомым значением несколько, будет найден индекс любого (т.е. не обязательно первого от начала) из них.

```
##
var a := ArrRandom(10, 0, 9);
a.Sort;
a.Println;
a.BinarySearch(5).Println
```

```
0 3 3 5 5 5 6 9 9 9
4
```

6.23. Перестановки и сочетания

Перестановками n элементов называются их упорядоченные совокупности длины n, отличающиеся порядком следования элементов. Количество возможных перестановок определяется по формуле $P_n = n!$

Если *a* – массив длиной *n*, то расширение *a.Permutations* вернет последовательность длиной *n!* всех перестановок, каждый элемент которой будет являться массивом.

```
## |1, 2, 3, 4|.Permutations.Print
```

```
[1,2,3,4] [1,2,4,3] [1,3,2,4] [1,3,4,2] [1,4,2,3] [1,4,3,2]
[2,1,3,4] [2,1,4,3] [2,3,1,4] [2,3,4,1] [2,4,1,3] [2,4,3,1]
[3,1,2,4] [3,1,4,2] [3,2,1,4] [3,2,4,1] [3,4,1,2] [3,4,2,1]
[4,1,2,3] [4,1,3,2] [4,2,1,3] [4,2,3,1] [4,3,1,2] [4,3,2,1]
```

К перестановкам приводит задача о вариантах размещения за столом группы из n человек.

Сочетаниями из n элементов по k называются их неупорядоченные совокупности длины k, отличающиеся друг от друга хотя бы одним элементом. Количество возможных сочетаний определяется по формуле $C_n^k = \frac{n!}{k!(n-k)!}$

Если *a* – массив длиной *n*, то расширение *a.Combinations(k)* вернет последовательность всех сочетаний C_n^k, каждый элемент которой будет являться массивом.

```
## |1, 2, 3, 4, 5|.Combinations(3).Print
```

```
[1,2,3] [1,2,4] [1,2,5] [1,3,4] [1,3,5] [1,4,5] [2,3,4] [2,3,5]
[2,4,5] [3,4,5]
```

К сочетаниям приводит задача о вариантах выбора k книг из n имеющихся.

Задача 6.38
Дан массив из n целых чисел, принадлежащих отрезку [-99;99]. Гарантируется, что n ⩾ 3. Найти все тройки чисел, сумма которых кратна 101.

```
##
var n := ReadInteger; // ввод n
var a := ArrRandom(n, -99, 99);
a.Println;
a.Combinations(3).Where(t -> t.Sum mod 101 = 0).Print
```

```
20
40 6 5 -58 14 2 80 -99 95 -32 -26 48 68 -4 -73 -20 10 -99 55 46
[40,6,55] [6,14,-20] [5,68,-73] [-58,48,10] [2,-99,-4] [2,-4,-99]
[-99,-4,-99] [95,-4,10] [-26,-20,46]
```

6.24. Последовательности в подпрограммах

Скорее всего, вы уже обратили внимание, что в этой главе последовательности и массивы все время являются параметрами и/или возвращаемыми значениями подпрограмм и методов. При написании собственных подпрограмм необходимо понимать, как правильно организовать работу с подобными параметрами и возвращаемыми значениями.

Если значение формального параметра в подпрограмме не изменяется, достаточно указать имя и тип параметра. Этого достаточно также в случае, когда подпрограмма изменяет значения элементов входного массива путем обычного поэлементного присваивания. В остальных случаях перед именем параметра нужно использовать ключевое слово **var**.

```
function SSqrt(a: sequence of integer) := a.Select(t -> Sqrt(t));

procedure Cube(a: sequence of integer; var b: sequence of integer);
begin
  b := a.Select(t -> t * t * t)
end;
```

```
procedure Half(a, b: sequence of integer);
begin
  b := a.Select(t -> t div 2)
end;


begin
  var a := Seq(3, 12, 52, 18, 6);
  SSqrt(a).Println;
  var b, d: sequence of integer;
  Cube(a, b);
  b.Println;
  var c := 10.To(20);
  Half(c, a);
  a.Println;
  Half(c, d);
  d.Print
end.
```

Внимательно изучите код и попробуйте самостоятельно найти допущенную ошибку, связанную с рассматриваемым материалом. Ваших знаний достаточно, чтобы найти ошибку, объяснить причину ее возникновения и предложить способ исправления.

В заголовке подпрограммы перед именем входного формального параметра может быть указано ключевое слово **params**. Далее всегда должно следовать описание динамического массива нужного типа, элементы которого будут переданы по значению. Если подпрограмма имеет несколько параметров, то описатель **params** может быть указан только для последнего из них; при этом не допускается указывать значение параметра по умолчанию.

Но зачем указывать **params**, если массив можно передать и без этого? В случае, если фактическим параметром будет только массив, наличие **params** действительно никакой роли не играет. Но, кроме массива, **params** позволяет передать в подпрограмму набор параметров, перечисленных через запятую, т.е. реализовать обращение к подпрограмме с переменным числом параметров. Именно такой вызов функции Min приведен ниже.

```
function Min(params a: array of real): real;
begin
  Result := a[0];
  for var i := 1 to a.High do
    if a[i] < Result then
      Result := a[i]
end;

begin
  Min(3, -5.1, 9.4, -2, 0, -8.2, 16.5).Println // -8.2
end.
```

В описании функции Min указано, что элементы массива имеют тип **real**. Если мы вместо массива будем передавать список параметров, как это указано в примере, в этом списке могут быть целочисленные литералы и будет выполнено автоприведение типа. Но если предварительно сформировать их этих значений массив, например, с помощью Arr( ), то все литералы должны будут изображать числа типа **real**. Еще одна причина использовать именно **params**.

6.25. Коллекция List

В программировании используется понятие ***коллекции*** – некоторой структуры, содержащей в себе набор элементов одного или различных типов и позволяющей манипулировать этими объектами посредством установленного набора операций. Коллекция предоставляет средства для помещения в себя данных, извлечения их, а также обеспечивает доступ к данным. Реализация набора операций для манипулирования данными существенно упрощает программирование, а также облегчает человеку понимание программного кода.

Одни коллекции хранят данные в порядке их поступления, другие некоторым образом эти данные переупорядочивают. Например, последовательности хранят элементы в порядке их включения, массивы – в соответствии с индексом элемента.

В PascalABC.NET можно напрямую обращаться к ***обобщенным*** стандартным коллекциям Microsoft .NET Framework, реализованным в

виде набора классов. Термин «обобщенная» означает, что тип элементов коллекции не фиксирован.

Обобщенная коллекция List<T> одновременно обладает свойствами ***линейного списка*** и динамического массива. Классы, на основе которых создаются объекты подобного типа, включены во многие современные языки программирования. Далее мы будем использовать термин «список List».

> ***Линейный список*** – это последовательность из нуля или более элементов (узлов), главной особенностью которого является такое относительное расположение узлов, будто они образуют одну линию. Если в линейном списке n>0 элементов, то узел j[k] (0<k<n) следует непосредственно за j[k-1] и предшествует j[k+1].

В списке List доступ к элементу обеспечивается по его индексу (номеру в списке, начинающемуся от нуля). Элементы или их последовательность можно добавлять в конец списка или в его произвольное место. Также, можно удалять один или более элементов из произвольного места списка. Можно считать, что список List – это расширяемый динамический массив с удобным удалением и добавлением элементов. Элементы списка хранятся в порядке их добавления. Имеется возможность поиска элемента, а если список упорядочен, то быстрого бинарного поиска. И, конечно, доступны все приемы для работы со списком, как с последовательностью.

6.25.1. Создание списка List

В PascalABC.NET список List является стандартным типом языка, поэтому его можно создать с использованием оператора присваивания вида

```
var имя := new List<тип_данных>;
```

В качестве типа данных, помещаемых в список List, можно указать любой допустимый тип, в частности, другой список.

```
var a := new List<real>;
var b := new List<array of integer>;
```

Конструктор списка дает возможность совместить создание списка с его заполнением элементами последовательности (или данных с типом, приводящимся к последовательности), указанной в качестве параметра.

```
var L1 := new List<integer>(SeqRandom(4, 10, 99));
var L2 := new List<integer>(L1.Sorted);
```

В приведенном примере список L1 заполняется данными от генератора случайных чисел. Список L2 заполняется данными из списка L1, которые предварительно сортируются в порядке возрастания.

Список можно заполнить непосредственно перечисленными данными, воспользовавшись «короткой функцией» Lst:

```
var L := Lst(25, -23, 47, 100, 0, 14);
```

Здесь тип данных списка будет автоматически выведен из типа перечисленных значений.

Список также можно создать при помощи расширения .ToList из данных, приводящихся к последовательности.

```
var L := SeqRandom(8, -999, 999).ToList;
```

Аргументом функции Lst может также быть объект любого типа, в том числе члены последовательности.

```
var L1 := Lst(Seq(14, 172, -5, 0, 39));
var L2 := Lst(Arr(14, 172, -5, 0, 39));
var L3 := Lst(ArrRandom(5, -99, 99));
var L4 := Lst(L3); // в L4 один элемент - список
```

6.25.2. Операции для работы со списком List

В дополнение к средствам, применяющимся при работе с последовательностями и массивами, список List имеет следующие:

- `.Count` – свойство, возвращающее количество элементов в списке;
- `.Add(объект)` – метод, добавляющий объект к концу списка;

- `.AddRange(s)` – метод, добавляющий к концу списка последовательность s, состоящую из объектов;
- `.Clear` – метод, очищающий список;
- `.CopyTo(имя_массива)` – метод, копирующий содержимое списка в существующий одномерный динамический массив. Если данные не помещаются в массиве, возникает исключение;
- `.CopyTo(имя_массива, индекс_в_массиве)` – метод, копирующий содержимое списка в существующий одномерный динамический массив, начиная с элемента, указанного индексом. Если данные не помещаются в массиве, возникает исключение;
- `.CopyTo(is, имя_массива, it, k)` – метод, копирующий содержимое k элементов списка в существующий одномерный динамический массив, начиная с элемента it. Копирование производится из списка, начиная с элемента, имеющего индекс is. Если данные не помещаются в массиве, возникает исключение;
- `.ToArray` – метод, копирующий содержимое списка в одномерный динамический массив и возвращающий этот массив в качестве результата. Не требует предварительного описания массива;
- `.GetRange(i, k)` – метод возвращающий список из k элементов, начиная с элемента, имеющего индекс i;
- `.Insert(индекс, объект)` – метод, вставляющий в список объект в виде элемента с указанным индексом;
- `.InsertRange(индекс, последовательность_объектов)` – метод, вставляющий последовательность объектов в список так, что первый вставленный элемент имеет указанный индекс;
- `.Remove(значение)` – метод, удаляющий из списка первый встреченный элемент с указанным значением;
- `.RemoveAt(индекс)` – метод, удаляющий из списка элемент с указанным индексом;
- `.RemoveAll(T -> boolean)` – метод, удаляющий из списка элементы, значения которых удовлетворяют условию;
- `.RemoveRange(i, k)` – метод, удаляющий из списка k элементов, начиная с элемента, имеющего индекс i;
- `Reverse(список)` – метод, изменяющий порядок следования элементов списка на обратный;
- `.TrueForAll(T -> boolean)` – метод, возвращающий True, если все значения всех элементов списка удовлетворяют заданному условию и False в противном случае.

К содержимому списка можно применять все методы для работы с массивами и последовательностями. В частности, обращаться к элементам, указывая индекс и использовать срезы, которые также будут представлять собой списки.

Внимание, важно! 1. Срезы списка List также будут списками List. 2. Если обращение к элементу списка происходит по индексу, можно изменить значение только всего этого элемента в целом.

6.25.3. Примеры использования списка List

Списки можно использовать во всех задачах, где ранее использовались последовательности и массивы. На основе списков особенно эффективно решаются задачи, связанные со вставкой и удалением элементов. Гвидо ван Россум, автор языка Python, вообще отказался от массивов, используя вместо них списки List.

Задача 6.39
Получить методом простого перебора все делители натурального числа, не превышающего 10^9.

Поскольку заранее неизвестно, какое количество делителей будет иметь заданное число n, здесь будет удобно применить список List. Занесем в него единицу и будем проводить проверку делимости от 2 до n/2. В конце добавим в список само число n.

```
##
var L := Lst(1);
var n := ReadInteger('Введите натуральное число');
for var i := 2 to n div 2 do
  if n mod i = 0 then L.Add(i);
if n <> 1 then L.Add(n);
L.Print
```

```
Введите натуральное число 19528423
1 41 67 2747 7109 291469 476303 19528423
```

Задача 6.15 (повтор)
Используя датчик случайных чисел, заполнить массив из двадцати элементов неповторяющимися значениями, принадлежащими отрезку [10; 99]. Элементы массива вывести в строку, разделяя их запятой с последующим пробелом.

Мы решали эту задачу, генерируя массив длиной 50 элементов, заполняя его случайными значениями и отбирая из него первых 20 неповторяющихся. Рассмотрим другой вариант решения.

```
##
var L := new List<integer>;
while L.Count < 20 do
begin
  var t := Random(10, 99);
  if not L.Contains(t) then
    L.Add(t)
end;
L.Print
```

Здесь время выполнения составляет 2.4 мкс, выигрывая у «длинного» варианта микросекунду. Короткий, внятный, корректный код.

6.26. Задачи для закрепления материала

Здесь приводятся задачи, решение которых требует привлечения знаний из различных разделов главы 6 – это позволит вам оценить, насколько усвоен материал.

6.26.1. Циклический сдвиг в массиве

Под циклическим сдвигом элементов массива влево понимают перемещение первого элемента в конец массива со сдвигом к началу остальных элементов. Под циклическим сдвигом элементов массива вправо понимают перемещение последнего элемента в начало массива со сдвигом к концу остальных элементов. Если сдвиг выполняется на k позиций, операция повторяется k раз.

Пусть дано слово «pascal». Выполним циклический сдвиг его букв влево и получим «ascalp». Циклический сдвиг слова «pascal» вправо дает «lpasca». А если в слове «информатика» выполнить циклический сдвиг вправо на 5 букв, получим «атикаинформ».

Наличие срезов в PascalABC.NET позволяет реализовать циклический сдвиг элементов массива одним оператором. Пусть *a* – массив, элементы которого нужно сдвинуть влево на 4 позиции. Тогда достаточно написать *a := a[4:] + a[:4]*.

В общем виде для сдвига массива *a* на *k* элементов (где натуральное *k* обязательно меньше длины массива) достаточно записать

```
a := a[k:] + a[:k]; // сдвиг влево;
a := a[^k:] + a[:k^]; // сдвиг вправо.
```

Циклические сдвиги элементов массива – скорее программистская экзотика, чем необходимость. Они используются в некоторых криптографических алгоритмах.

Задача 6.40

Дан целочисленный массив, каждый из восьми элементов которого принадлежит отрезку [0;9] и представляет собой цифру некоторого восьмизначного числа. Предполагается, что число может иметь левые (незначащие нули). Найти разность чисел, сформированнных из заданного массива, полученных путем циклического сдвига элементов влево на 5 и циклического сдвига вправо на 2.

Чтобы лучше уяснить задачу, рассмотрим пример. Заодно, его можно использовать в качестве контрольного при отладке программы. Пусть элементы массива равны соответственно 4, 7, 2, 0, 6, 8, 3, 2. Это дает исходное число 47206832. При циклическом сдвиге влево на 5 цифр получим 83247206, при циклическом сдвиге вправо на 2 цифры получим 32472068. Разность этих чисел равна 50775138.

Чтобы из элементов массива сформировать десятичное число, вспомним о его расширенной записи, когда каждый разряд числа представляется цифрой, умноженной на соответствующую степень числа 10. Такое формирование потребуется делать дважды и это разумный довод в пользу написания соответствующей функции.

```
function КЧислу(a: array of integer): integer;
begin
  Result := a[0];
  for var i := 1 to a.High do
    Result := Result * 10 + a[i]
end;
```

```
begin
  var a :=|4, 7, 2, 0, 6, 8, 3, 2|; // для отладки
  // var a := ArrRandom(8, 0, 9);  // для работы
  a.Println;
  Print(КЧислу(a[5:] + a[:5]) - КЧислу(a[^2:] + a[:^2]))
end.
```

```
4 7 2 0 6 8 3 2
50775138
```

6.26.2. Многоугольник с координатами вершин

Для школьных задач характерны два способа задания координат каждой из n вершин. Первый способ предусматривает использование двух статических массивов X и Y длиной n, так что координаты i-й точки описываются парой значений X_i, Y_i. При втором способе используется массив A длиной 2n, а координаты i-й точки описываются парой значений A_{2i-1}, A_{2i}. Вы можете использовать динамические массивы, сдвинув индексы на 1 и получив соответственно X_{i-1}, Y_{i-1} и $A_{2(i-1)}$, A_{2i-1} для $1 \leqslant i \leqslant n$.

§1. Нахождение периметра многоугольника

Периметр многоугольника равен сумме длин каждой его стороны. Стороны – это отрезки, заключенные между вершинами (1, 2), (2, 3), … (n-1, n), а также (n, 1). Длина каждой стороны находится по формуле, приведенной в Приложении П3.1 первой части книги. Запишем операторы PascalABC.NET, позволяющие вычислить длину отрезка, заключенного между точками i и j ($0 \leqslant i \leqslant n - 1$):

```
var L := Sqrt(Sqr(x[j] - x[i]) + Sqr(y[j] - y[i]));
```

§2. Нахождение площади многоугольника

Площадь многоугольника можно найти, разбив его на треугольники и просуммировав их площади. Но можно также воспользоваться формулой Гаусса (ее называют «формула землемера» и «метод шнурования»). Все детали метода отлично описаны в Интернет. Мой вариант решения с необходимыми пояснениями можно найти по ссылке: https://pascalabcnet.github.io/olymp_geometry.html, задача №370.

Задача 6.40

Для n пятиугольников, абсциссы и ординаты вершин которых записаны в виде целочисленных значений элементов массивов X и Y соответственно, найти номер и площадь многоугольника, имеющего минимальный периметр. Если таких многоугольников несколько, найти площадь каждого из них.

Общая схема решения: находим множество индексов, которые принадлежит координатам прямоугольников с минимальным периметром, а затем вычисляем требуемые площади. Вычисление периметров и площадей оформим в виде функций. Поскольку в задании говорится о пятиугольниках, в каждом массиве придется рассматривать пятерки элементов, относящихся к одному и тому же пятиугольнику.

```
function Длина(x1, y1, x2, y2: integer) :=
    Sqrt(Sqr(x2 - x1) + Sqr(y2 - y1));

function Периметр5(x, y: array of integer; k: integer): real;
begin
  // k - номер пятиугольника (0..n-1)
  Result := Длина(x[5 * k + 4], y[5 * k + 4], x[5 * k], y[5 * k]);
  for var i := 0 to 3 do
    Result += Длина(x[5 * k + i], y[5 * k + i],
        x[5 * k + i + 1], y[5 * k + i + 1])
end;


function Площадь5(ax, ay: array of integer; k: integer): real;
begin
  // k - номер пятиугольника (0..n-1)
  var (x, y) := (ax[5 * k], ay[5 * k]);
  var (xa, ya) := (x, y);
  (x, y) := (xa, ya);
  var S: int64 := 0;
  for var i := 1 to 4 do
  begin
    var (xb, yb) := (ax[5 * k + i], ay[5 * k + i]);
    S += xa * yb - ya * xb;
    (xa, ya) := (xb, yb);
  end;
  Result := Abs(S + xa * y - ya * x) / 2
end;
```

```
begin
  var n := ReadInteger; // количество пятиугольников
  var x := ArrRandom(5 * n, -99, 99); // абсциссы
  x.Println;
  var y := ArrRandom(5 * n, -99, 99); // ординаты
  y.Println;
  var P := (0..n-1).Select(i -> Периметр5(x, y, i)).ToArray;
  var min := P.Min;
  foreach var i in P.Indices(t -> t = min) do
    Writeln('Пятиугольник №', i + 1, ', площадь ', Площадь5(x, y, i))
end.
```

```
4
33 56 -86 62 -72 34 35 -14 -66 -27 -60 9 2 51 84 -80 -28 12 -39 76
91 -62 -37 78 89 -9 -53 80 -85 51 72 20 73 -17 -43 -77 -65 -46 11 66
Пятиугольник №3, площадь 1142.5
```

Внеся небольшие изменения в формулы, можно решать эту задачу для треугольников, четырехугольников и т.п.

6.26.3. Упорядоченность последовательности

В подобных задачах дается некоторая последовательность и требуется проверить, является ли она упорядоченной по тому или иному признаку.

§1. Проверка полной упорядоченности

Самый эффективный алгоритм – ввод данных, соединенный с перебором, который ведется либо до исчерпания последовательности, либо до первого нарушения упорядоченности. Но по-школьному формально, если сказано, что даны n значений, нужно сначала их ввести в массив и только потом анализировать.

Задача 6.41
Даны n натуральных чисел (n > 1). Проверить, являются ли они упорядоченными по возрастанию.

```
##
var n := ReadInteger;
var a := ReadArrInteger(n);
var (Слева, Справа, i) := (a[0], a[1], 2);
while (Справа > Слева) and (i < n) do
begin
  (Слева, Справа) := (Справа, a[i]);
  i += 1
end;
if Справа > Слева then
  Print('Упорядочены')
else
  Print('Неупорядочены')
```

```
5
1 3 6 5 8
Неупорядочены
```

Можно ли решить эту задачу короче? Обратите внимание, что мы все время сравниваем очередное значение с предыдущим, т.е. просматриваем пары (a_0, a_1), (a_1, a_2), ... (a_{n-2}, a_{n-1}). Такие кортежные пары для последовательности умеет создавать расширение a.Pairwise. Из последовательности длиной n будет создана последовательность, содержащая n-1 двухэлементный кортеж.

```
##
var n := ReadInteger;
if ReadSeqInteger(n).Pairwise.Any(t -> t[1] <= t[0]) then
  Print('Неупорядочены')
else
  Print('Упорядочены')
```

§2. Нахождение интервалов упорядоченности

Задача 6.42

Дана случайная целочисленная последовательность длиной n. Найти в ней участок наибольшей длины, на котором значения упорядочены по невозрастанию и вывести его элементы. Если таких участков несколько, указать первый из них. Гарантируется, что последовательность содержит не менее двух чисел.

Рассмотрим решение, в котором исходная последовательность находится в массиве. Для того, чтобы идентифицировать участок, доста-

точно знать его начальный или конечный индекс, а также длину. Найдем первый участок и примем его за максимальный. Хранить будем индекс элемента *im*, с которого участок начинается, а также длину участка *lm*. Если длина очередного участка *lc* превышает *lm*, заносим *lc* в *lm* и *ic* (индекс элемента, с которого начинается текущий участок) в *im*. Просмотрев всю последовательность, выводим значения элементов найденного участка или не выводим ничего, если значение *lm* меньше двух.

```
begin
  var n := ReadInteger('n =');
  var a := ArrRandom(n, -99, 99);
  a.Println;
  var (im, lm, lc) := (0, 1, 1);
  for var i := 1 to a.High do
  begin
    if a[i] <= a[i - 1] then
      Inc(lc)
    else
    begin
      if lc > lm then
        (im, lm) := (i - lc, lc);
      lc := 1
    end
  end;
  if lc > lm then
    (im, lm) := (a.High - lc + 1, lc);
  if lm > 1 then
    a[im:im + lm].Print
end.
```

```
n = 20
92 42 36 25 -26 -30 21 95 66 90 -62 0 32 3 -68 -82 95 -29 18 18
92 42 36 25 -26 -30
```

6.26.4. Слияние упорядоченных последовательностей

Задача 6.43

Имеются две упорядоченные по неубыванию последовательности натуральных чисел. Вывести через пробел последовательность, полученную в результате слияния исходных последовательностей, при котором ее элементы будут также упорядочены по неубыванию.

Смоделируем исходные последовательности с помощью датчика случайных чисел. Для проверки достаточно взять значения на отрезке [1; 99]. Рассмотрим возможные пути решения этой задачи.

Самый простой путь – объединить исходные последовательности посредством *a + b*, а затем отсортировать полученную последовательность по неубыванию. Неплохой вариант, если не требуется написание программы, эффективной по памяти. Зато безразлично, упорядочены исходные последовательности или нет.

Более сложный путь означает использование одного из алгоритмов слияния. При этом результирующую последовательность можно не хранить. Воспользуемся алгоритмом слияния с барьером. Его особенность в том, что в конец каждой исходной последовательности приписывается барьерное значение – величина, которой данные достичь не могут. Для неубывающей последовательности типа **integer** это может быть значение *MaxInt* (оно же – ***integer***.*MaxValue*). Достижение барьера – сигнал о том, что данные закончились. Наличие барьера существенно упрощает логику алгоритма слияния.

Сам алгоритм чрезвычайно прост. Сравниваем значения в первой и второй последовательности. Меньшее значение выводим и продвигаемся по последовательности, содержавший это значение. В случае равенства значений условно считаем, что одно из значений меньше. Процесс прекращаем, когда в обеих последовательностях будет достигнут барьер.

Исходные данные генерируем на основе последовательностей. Это разумно, потому что далее последует сортировка, возвращающая последовательность и нет смысла формировать массив случайных чисел, который лишь занимает память. Результат поместим в список List и это тоже разумно, поскольку затем производится вывод и добавляется барьерный элемент, т.е. происходит изменение количества элементов по сравнению с исходной последовательностью. А далее будем работать со списком List, как с обычным массивом.

```
begin
  var n := ReadInteger('Длина 1-й последовательности:');
  var a := SeqRandom(n, 1, 99).Order.ToList; // 1-й список
  a.Println;
  a.Add(MaxInt); // добавили в список барьер
  n := ReadInteger('Длина 2-й последовательности:');
  var b := SeqRandom(n, 1, 99).Order.ToList; // 2-й список
  b.Println;
  b.Add(MaxInt); // добавили в список барьер
  var (ia, ib) := (0, 0); // индексы для прохода по спискам
  while (a[ia], b[ib]) <> (MaxInt, MaxInt) do
    if a[ia] < b[ib] then
    begin
      a[ia].Print;
      ia += 1
    end
    else
    begin
      b[ib].Print;
      ib += 1
    end
end.
```

```
Длина 1-й последовательности: 9
10 17 25 49 56 58 60 68 98
Длина 2-й последовательности: 12
21 31 32 33 44 54 55 59 64 75 79 83
10 17 21 25 31 32 33 44 49 54 55 56 58 59 60 64 68 75 79 83 98
```

Обратите внимание на использованную в теле цикла **while** конструкцию – это почленное сравнение элементов кортежей! Здесь она эквивалентна условию *(a[ia] <> MaxInt)* ***or*** *(b[ib] <> MaxInt)*.

Как изменится программа, если понадобится сохранить результат слияния? Очень незначительно в случае использования для этой цели списка List.

```
begin
  var n := ReadInteger('Длина 1-й последовательности:');
  var a := SeqRandom(n, 1, 99).Order.ToList; // 1-й список
  a.Println;
  a.Add(MaxInt); // добавили барьер
  n := ReadInteger('Длина 2-й последовательности:');
  var b := SeqRandom(n, 1, 99).Order.ToList; // 2-й список
  b.Println;
  b.Add(MaxInt); // добавили барьер
  var res := new List<integer>;
  var (ia, ib) := (0, 0); // индексы для прохода по спискам
  while (a[ia], b[ib]) <> (MaxInt, MaxInt) do
    if a[ia] < b[ib] then
    begin
      res.Add(a[ia]);
      ia += 1
    end
    else
    begin
      res.Add(b[ib]);
      ib += 1
    end;
  res.Print
end.
```

Можно ли использовать массив, а не список List? Безусловно. Но придется вначале определить его размер, получив сумму длин исходных последовательностей, а затем поддерживать значение индекса для поочередной замены значений элементов этого массива. Алгоритм станет несколько сложнее, а выгоды практически ноль.

Но если список List универсальнее массива и полноценно его заменяет, быть может, неверно критиковать язык Python за отсутствие в нем массивов и наличие только List? Давайте попытаемся ответить на этот вопрос после изучения следующей главы, где пойдет речь о двумерных массивах – матрицах. Мы посмотрим, является ли равноценной заменой двухмерному массиву список List, каждый элемент которого, в свою очередь, является списком List.

6.26.5. Массивы – параметры функций

Задача 6.44
Даны два массива, первый из которых заполнен двухзначными натуральными числами, а второй – трехзначными. Для каждого массива найти отношение суммы квадратов элементов с четной суммой цифр к сумме квадратов элементов с нечетной суммой цифр. Массивы заполнить с использованием датчика случайных чисел. Количество элементов в каждом массиве не должно быть меньше десяти.

Составим две функции. Первая будет возвращать сумму цифр в натуральном числе, вторая – искомое отношение, найденное для заданного массива.

```
function СуммаЦифр(Число: integer): integer;
begin
  Result := 0;
  while Число > 0 do
  begin
    Result += Число mod 10;
    Число := Число div 10
  end
end;

function Отношение(Массив: array of integer): real;
begin
  var (s1, s2) := (0, 0);
  foreach var Элемент in Массив do
  begin
    if СуммаЦифр(Элемент).IsEven then
      s1 += Sqr(Элемент)
    else
      s2 += Sqr(Элемент)
  end;
  Result := s1 / s2
end;
```

```
begin
  var n := ReadInteger('Длина первого массива:');
  var a := ArrRandom(n, 10, 99);
  a.Println;
  var Отношение1 := Отношение(a);
  n := ReadInteger('Длина второго массива:');
  a := ArrRandom(n, 100, 999);
  a.Println;
  var Отношение2 := Отношение(a);
  Print(Отношение1, Отношение2)
end.
```

```
Длина первого массива: 11
90 96 56 64 47 71 91 95 96 82 44
Длина второго массива: 12
105 425 784 263 301 566 605 291 495 730 504 947
1.10120149323964 1.03113504912051
```

Глава 7

Матрицы

В этой главе…

Описание и создание матриц
Генерация матриц
Ввод и вывод
Выборка элементов
Модификация строк и столбцов
Транспонирование матрицы
Примеры решения задач

Вам случалось любоваться Матрицей? Ее гениальностью...

Агент Смит
«Матрица»

В предыдущей главе были рассмотрены одномерные массивы. Но массив может иметь и большее число измерений.

Если M(x,y) - точка на плоскости, в декартовой системе координат она имеет две координаты. Точка M(x,y,z) имеет три координаты. При рассмотрении положения движущейся в трехмерном пространстве точки добавляется значение времени t, и точка M(x,y,z,t) будет описываться четырьмя координатами. Можно придумать варианты, когда понадобится массив и с еще большим числом измерений. Количество чисел, необходимых для однозначного задания положения точки – физический эквивалент понятия размерности массива. Из многомерных массивов наиболее популярны двухмерные (можно также писать двумерные) массивы. В PascalABC.NET двухмерные динамические массивы называются ***матрицами***.

Матрица имеет два измерения, называемые строками и столбцами по аналогии с матрицами в математике. Матрица всегда имеет прямоугольную форму: количество элементов в каждой строке одинаково и количество элементов в каждом столбце тоже одинаково. Визуально матрицу всегда можно представить в виде таблицы.

Пусть имеется матрица, состоящая из m строк и n столбцов. В этом случае говорят, что она имеет размер $m \times n$. Не путайте размер с размерностью: в матрице размерность (число измерений массива) равна двум. Если элемент матрицы A находится на пересечении строки i и столбца j, его записывают как $A_{i,j}$ или *A[i, j]*. Первый индекс – всегда номер строки, второй – номер столбца.

7.1. Описание и создание матриц

Матрица – разновидность массива. Массивы бывают статическими и динамическими. Размеры (т.е. количество элементов) по каждому измерению статического массива должны быть известны компиля-

тору, иначе он не сможет зарезервировать нужный объем памяти компьютера. В случае динамического массива мы сообщаем компилятору лишь количество измерений массива. Конечно, в любом случае, мы указываем еще и тип элементов массива. Динамический массив создается во время работы программы, и тогда же определяются его размеры. Процедура SetLength, с помощью которой можно в процессе выполнения программы задать или поменять размер одномерного массива, работает и для двухмерных массивов. Но если с первоначальным заданием размера массива все просто, то с его переопределением есть проблема. Мы разберемся с этим позже.

Пусть нам требуется двухмерный динамический массив *a* размером 4×3. В PascalABC.NET элементы динамических массивы индексируются от нуля, поэтому получаются четыре строки с индексами от 0 до 3 и три столбца с индексами от 0 до 2.

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \end{pmatrix}$$

Каждую строку двухмерного массива можно представить, как обычный массив, что позволяет создавать массивы с разным количеством элементов в строке. Массив из таких строк образует массив массивов. Построить его несложно.

```
##
var A: array of array of integer;
var (m, n) := (4, 3); // число строк и столбцов
SetLength(A, m); // распределим память под m строк
for var i := 0 to m - 1 do
  SetLength(A[i], n); // в каждой строке создадим массив из n элементов
A[3][2] := 43; // строка с индексом 3, в ней элемент индексом 2
A[1, 0] := 21; // строка с индексом 1, в ней элемент индексом 0
Writeln(A) // [[0,0,0],[21,0,0],[0,0,0],[0,0,43]]
```

При обращении к элементу такого «двухмерного массива» допускается запись как двух индексов по отдельности, так и совместно. Можно обойтись без этих игр с SetLength, описав «нормальный» двухмерный массив, который в PascalABC.NET именуется матрицей.

```
##
var A: array [,] of integer; // внимание, тут запятая!
var (m, n) := (4, 3); // число строк и колонок
SetLength(A, m, n);
A[3, 2] := 43; // писать A[3][2] не допускается
A[1, 0] := 21;
Writeln(A) // [[0,0,0],[21,0,0],[0,0,0],[0,0,43]]
```

Для двухмерных массивов, подобно одномерным, можно совмещать описание с выделением памяти. Для этого массив создается с использованием ключевого слова **new**.

```
var p: array [,] of integer := new integer[4, 3]; // с описанием
var q := new real[4, 3]; // с автовыведением типа
```

Можно также выполнить инициализацию, добавив ***конструктор массива***.

```
var a := new integer[4, 3] ((1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12));
var b := new real[2, 3] ((2.1, 3.7, 5), (1, 2, 3));
var d: array of array of integer := ((1, 2, 3), (4, 5), (6, 7, 8));
```

После такой инициализации массив *a* будет содержать следующие значения

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

Конструктор двухмерного массива имеет два уровня скобок. Первый (внешний) уровень ограничивает сам конструктор, второй (внутренний) – ограничивает каждую строку. В приведенном примере массив *a* имеет четыре строки и три колонки. Соответственно, конструктор содержит четыре группы по три значения в каждой. Пример с инициализацией массива *b* напоминает нам о том, что заданные в конструкторе значения автоматически приводятся к нужному типу.

Массив *d* инициализируется как массив массивов. Отметьте, что в данном случае конструктор позволяет создать и инициализировать

непрямоугольный массив: во второй строке два элемента, а в остальных по три. К элементу массива *d* можно также обращаться в виде *d[i][j]* или *d[i, j]*. У массивов *a* и *d* разный тип данных. А это означает что их, например, нельзя передавать в подпрограммы один вместо другого. Обратите внимание на то, что в терминологии PascalABC.NET массив *d* – **не матрица**!

Статический двухмерный массив также не является матрицей PascalABC.NET. Как правило, такие массивы используются для целей совместимости с базовым Паскалем. При описании статического массива требуется задать границы его индексов явно.

```
var a: array [1..4, 1..3] of integer; // двухмерный массив
var b: array[-1..2] of array[1..3] of real; // массив массивов
```

Как и для динамических массивов, описание можно совместить с присваиванием значений.

```
var a: array[1..4, 1..3] of integer :=
    ((1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12));
var b: array[1..2, 1..3] of real := ((2.1, 3.7, 5), (1, 2, 3));
```

7.2. Генерация матриц

Генерация матриц подразумевает создание матрицы и последующее присваивание ее элементам некоторых значений. В случае, когда нужно присвоить всем элементам матрицы одинаковые значения, говорят о ***заливке*** матрицы. Генерация матриц в PascalABC.NET подразумевает вызов некоторой функции, возвращающей двухмерный динамический массив необходимых размеров. Это позволяет заранее не описывать матрицы, а пользоваться автоматическим выведением типа.

7.2.1. Заполнение заданными значениями

По умолчанию матрица заполняется построчно, т.е. сначала заполняется первая строка, затем вторая и т.д. Имеются способы заполнения матрицы и по столбцам, но в подобных случаях об этом сообщается явно.

Матрицу можно заполнить (и при этом создать, либо пересоздать), указав в правой части оператора присваивания вызов функции Matr одного из следующих форматов:

```
Matr(ЧислоСтрок, ЧислоСтолбцов, МассивДанных);
Matr(МассивМассивовДанных);
```

В качестве источника данных может быть использован массив длиной *ЧислоСтрок × ЧислоСтолбцов* (контролируется во время выполнения программы), содержащий данные, записанные в построчном порядке. Во втором формате данные размещаются в массиве массивов. В последнем случае данных может быть указано меньше, чем требуется. Недостающие элементы, если таковые будут, заполняются нулями. Рассмотрим пример.

```
##
var a := Matr(2, 3, |11, 12, 13, 21, 22, 23|);
a.Println;
Writeln;
var b := Matr(|11, 12|, |21, 22|);
b.Println;
Writeln;
var c := Matr(|11, 12, 13|, |21|, |31, 32|);
c.Print
```

```
11  12  13
21  22  23

11  12
21  22

11  12  13
21   0   0
31  32   0
```

Обратите внимание, что .Print выводит матрицу построчно. Разберитесь, почему в матрице *c* появились нули и почему именно на этих местах.

Как уже упоминалось, при решении задач ЕГЭ требуется явно инициализировать матрицу, если этого требует алгоритм. Проще всего поступить так:

```
var a := MatrFill(m, n, 0); // заливка нулем
```

7.2.2. Заполнение случайными значениями

Имеются два генератора, заполняющие матрицу целыми, либо вещественными значениями.

- MatrRandom(m, n, a, b) – заполнение матрицы размера m×n целыми числами, принадлежащие отрезку [a; b]. Имеется синоним MatrRandomInteger;
- MatrRandomReal(m, n, a, b) – заполнение матрицы размера m×n вещественными числами из промежутка [a; b).

Значения параметров, принимаемые по умолчанию: m=5, n=5, a=0, b=100.

Следующий пример иллюстрирует создание матриц и заполнение их случайными значениями, а также удобный способ организации вывода матриц при помощи расширения .Print. Это расширение умеет «пропускать через себя» матрицу, поэтому, если у вас «синдром одной строки», можете смело встраивать .Print в цепочку.

```
##
var a := MatrRandom(6, 9, -50, 50);
a.Println;
Writeln;
var b := MatrRandomReal(4, 3, -5, 5);
b.Print
```

```
 39   8  50 -21  15 -42  12  25  19
-38 -35   7 -37  27   4 -12  44 -12
  0 -49  45  13  11 -12 -43  46  39
 -7  19  18  46 -46  -8  40 -49 -13

 -4.30  -1.49  -4.66
  0.62   3.09  -0.52
 -4.76   3.59   1.23
  1.63   1.03   4.25
```

Еще раз отметим аккуратный вывод, который дает .Print. По умолчанию под вывод целочисленного значения отводятся четыре позиции, под элемент вещественного типа – семь позиций, в двух из которых размещается дробная часть.

Если нужна иная разметка вывода, можно указать количество позиций явно.

```
##
MatrRandom(6, 9, -50, 50).Println(6);
Println;
MatrRandomReal(4, 3, -5, 5).Println(11, 7)
```

```
19   -32     6    27    35    40   -22    10    30
23   -18   -35    32    16   -41    34    32    48
33   -31   -16    25    -1    34    42   -11    18
-1    11   -23    -5   -17    -9    43    29    47
 2   -16   -48   -23     2     7    18   -47   -39
26    -2   -43    -2   -28   -20    -7    36    47

 -3.5468194  4.4170922  2.0108772
  2.2497611  3.4569828 -0.4248126
 -2.7983750  2.9217856  4.9491709
  0.2331057 -2.2594955  4.9472349
```

7.2.3. Заполнение фиксированным значением

MatrFill(m, n, x) возвращает матрицу размера m×n, заполненную значением выражения x. Тип элементов матрицы будет совпадать с типом значения x.

```
##
var a := MatrFill(3, 3, 777);
a.Println;
Writeln;
var b := Matr(2, 2, 4 * |11|).Print // альтернатива MatrFill
```

```
777 777 777
777 777 777
777 777 777

 11  11
 11  11
```

7.2.4. Заполнение значениями, зависящими от индексов

MatrGen(m, n, (i, j) -> f(i, j)) возвращает матрицу размера m×n, каждый элемент которой заполняется значением некоторой функции f от своих индексов. Например, квадратная матрица с единичными эле-

ментами на диагонали от $A_{0,0}$ до $A_{5,5}$ и нулями в остальных элементах строится вызовом

```
var a := MatrGen(6, 6, (i, j) -> i = j ? 1 : 0);
```

А вот так получается таблица умножения (привет турбопаскалям и питонам):

```
## MatrGen(10, 10, (i, j) -> (i + 1) * (j + 1)).Println
```

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

Еще пример. Заполнение матрицы размером 3×8 первыми 24 числами Фибоначчи.

```
## Matr(3, 8, ArrGen(24, 1, 1, (i, j)-> i + j)).Println(6)
```

```
    1     1     2     3     5     8    13    21
   34    55    89   144   233   377   610   987
 1597  2584  4181  6765 10946 17711 28657 46368
```

Подобный прием удобно использовать для оперативного формирования и вывода различных таблиц.

7.2.5. Ввод значений элементов с клавиатуры

В базовом Паскале имеется единственный способ ввести значения элементов матрицы: организовать перебор всех элементов во вложенных циклах с последовательным присваиванием введенного значения очередному элементу. Порядок ввода можно задать как по строкам (второй индекс меняется быстрее первого), так и по столбцам (первый индекс меняется быстрее).

```
begin
  var (m, n) := (3, 2);
  var a := new real[m, n];
  for var i := 0 to m - 1 do
    for var j := 0 to n - 1 do
      Read(a[i, j]);
  a.Println
end.
```

```
2.5 -1.4 3.9 0.15 2 0.03
   2.50  -1.40
   3.90   0.15
   2.00   0.03
```

Чтобы каждый раз не писать подобные типовые вложенные циклы, в PascalABC.NET введены функции ReadMatrInteger(m, n) и ReadMatrReal(m, n), возвращающие матрицу размера m×n типа **integer** или **real** соответственно, заполненную принятыми с клавиатуры значениями. Для прочих типов данных придется пользоваться приведенным выше решением на базе вложенных циклов.

```
##
var (m, n) := (3, 2);
var a := ReadMatrReal(m, n);
a.Println
```

7.3. Вывод матриц

Расширение .Print (и его разновидность .Println) для матриц имеет некоторые особенности. Во-первых, оно возвращает не последовательность, а матрицу тех же размеров. Во-вторых, необязательным параметром здесь является не строка-разделитель, а количество позиций, отводимых под длину поля для выводимого элемента. А главное – вывод осуществляется построчно с сохранением формы матрицы.

- .Print(n) – для всех целочисленных типов матриц выводит каждый элемент в n позициях. По умолчанию n=4 ;
- .Print(n, k) – для вещественных матриц отводит под вывод значения элемента n позиций, сохраняя k знаков в дробной части (с округлением). По умолчанию n=7, k=2.

В отличие от базового Паскаля, не допускающего указывать имя массива в процедуре вывода Write, PascalABC.NET позволяет записы-

вать в качестве параметров процедур Write и Print любые массивы. При выводе массив заключается в квадратные скобки и поэтому такой способ пригоден в основном для целей отладки. В частности, при выводе матрицы каждая строка дополнительно будет заключена в квадратные скобки, а сам вывод пойдет в одну строку.

7.4. Переопределение размеров матрицы

Переопределить размеры матрицы можно посредством процедуры SetLength. Для примера рассмотрим изменение размера матрицы с 3×4 на 2×7. Первоначально в матрице содержится 12 элементов, а после изменения размеров их станет 14, поэтому значения двух элементов не будут определены. Но мы помним, что все данные числового типа, не получившие значений, в .NET будут автоматически инициализированы нулями.

Казалось бы, достаточно просто задать новые размеры матрицы при помощи процедуры SetLength (мы так уже поступали с одномерными массивами) – и задача решена. Посмотрим, так ли это:

```
##
var a := MatrGen(3, 4, (i, j)-> 10 * (i + 1) + j + 1);
a.Println;
Writeln;
SetLength(a, 2, 7);
a.Print
```

```
11  12  13  14
21  22  23  24
31  32  33  34

11  12  13  14   0   0   0
21  22  23  24   0   0   0
```

Увы, результат оказался вовсе не таким, каким мы его ожидали!

Если визуально наложить полученную матрицу на исходную, станет хорошо видно, что сохранились значения элементов, принадлежащие общей части обеих матриц. Прочие элементы оказались обнулены.

Секрета тут нет. Матрицы в памяти компьютера хранятся построчно, т.е. второй индекс у элементов увеличивается быстрее первого. После того, как процедура SetLength перераспределила память, в первой и второй строках стало по семь элементов вместо четырех. Третьей строки в новой матрице нет, поэтому третья строка исходной матрицы не рассматривалась.

Как же получить матрицу иных размеров без потери данных? Для этого придется извлечь из матрицы данные и получить на их основе новую матрицу нужного размера.

§1. Преобразование матрицы в последовательность

Элементы матрицы можно преобразовать в последовательность, считывая их по строкам или колонкам. Для этого используются следующие расширения:

– a.ElementsByRow считывает элементы матрицы *a* построчно;
– a.ElementsByCol считывает элементы матрицы *a* поколонно.

Полученную последовательность можно преобразовать в массив и сформировать на его основе матрицу посредством функции Matr.

```
##
var a := MatrGen(3, 4, (i, j)-> 10 * (i + 1) + j + 1);
a.Println;
Writeln;
a := Matr(2, 7, a.ElementsByRow.ToArray + 2 * |0|);
a.Print
```

```
11  12  13  14
21  22  23  24
31  32  33  34

11  12  13  14  21  22  23
24  31  32  33  34   0   0
```

Здесь пришлось добавить два нулевых элемента к массиву, поскольку функция Matr в данном формате требует точного соответствия длины массива количеству элементов в создаваемой матрице. Известно, что если размер массива планируется менять, лучше использовать список List. Но в данном случае вместо одного оператора нам пришлось бы отдельно писать добавление элементов к списку, а затем приводить список к массиву.

Другой вариант – создать матрицу на основе последовательности посредством функций MatrByRow и MatrByCol, которые будут описаны далее. В этом случае недостающие данные будут заполнены нулями, а лишние отсекутся.

```
##
var a := MatrGen(3, 4, (i, j)-> 10 * (i + 1) + j + 1);
a.Println;
Writeln;
a := MatrByRow(2, 7, a.ElementsByRow);
a.Print
```

§2. Преобразование матрицы в массив массивов

Элементы матрицы можно извлечь в массив массивов, считывая их по строкам или колонкам. Для этого используются следующие расширения:

– a.Rows считывает элементы матрицы *a* построчно;
– a.Cols считывает элементы матрицы *a* поколонно.

Полученный массив массивов можно преобразовать в матрицу посредством функций Matr, MatrByCol и MatrByRow. Все варианты подразумевают, что размеры массива массивов и создаваемой матрицы должны совпадать. В приводимом примере строки и столбцы в матрице меняются местами (извлекаем данные построчно, а матрицу формируем поколонно).

```
##
var a := MatrGen(3, 4, (i, j)-> 10 * (i + 1) + j + 1);
a.Println;
Writeln;
a := MatrByCol(a.Rows);
a.Print
```

```
11  12  13  14
21  22  23  24
31  32  33  34

11  21  31
12  22  32
13  23  33
14  24  34
```

7.5. Сведения о размерах матрицы

- a.RowCount – расширение, возвращающее количество строк;
- a.ColCount – расширение, возвращающее количество столбцов.

Очень полезная вещь в процедурах и функциях. Позволяет не передавать в них лишние параметры. Да и в цикле перебрать элементы строки от 0 до a.ColCount-1 тоже вполне удобно.

7.6. Выборка элементов матрицы

Выбирать можно строку, столбец, а также элементы в порядке прохода матрицы по строкам или столбцам. Можно выбрать все элементы матрицы вместе с их индексами. Результат может быть последовательностью, одномерным массивом или массивом массивов. Все эти операции обеспечивает большой набор расширений.

- a.Col(k) – возвращает в виде массива колонку матрицы *a* с номером *k* (отсчет номеров ведется от нуля);
- a.ColSeq(k) – возвращает в виде последовательности колонку матрицы *a* с номером *k* (отсчет номеров ведется от нуля);
- a.Cols – возвращает массив колонок матрицы *a*, в которой каждая колонка, в свою очередь, является массивом;
- a.Row(k) – возвращает в виде массива строку матрицы *a* с номером *k* (отсчет номеров ведется от нуля);
- a.RowSeq(k) – возвращает в виде последовательности строку матрицы *a* с номером *k* (отсчет номеров ведется от нуля);
- a.Rows – возвращает массив строк матрицы *a*, в которой каждая строка, в свою очередь, является массивом;
- a.ElementsByCol – возвращает последовательность элементов матрицы *a*, выбирая их по колонкам;
- a.ElementsByRow – возвращает последовательность элементов матрицы *a*, выбирая их по строкам;
- a.ElementsWithIndices – возвращает последовательность трехэлементных кортежей, в которой каждый элемент формируется на основе элемента a_{ij} и имеет структуру вида (значение, индекс *i*, индекс *j*);
- a.MatrSlice(ar, ac) – возвращает новую матрицу, строки которой выбираются из исходной матрицы по индексам строки, находящимся в массиве *ar*, а столбцы – по индексам столбцов, находящимся в массиве *ac* (срез матрицы):

– a.MatrSlice(fr, tr, fc, tc) – возвращает новую матрицу, строки которой выбираются из исходной матрицы по индексам от *fr* до *tr*, а колонки – индексам от *fc* до *tc* (срез матрицы).

Ниже приведена программа, выполняющая различные операции, связанные с извлечением данных из матриц.

```
function СоздатьМатрицу(m, n: integer): array[,] of integer;
begin
  Result := MatrRandom(m, n, -50, 50);
  Result.Println(4);
  Println('-' * 4 * n)
end;

begin
  var a := СоздатьМатрицу(3, 5);
  var b := СоздатьМатрицу(2, 7);
  Println('Min(A) =', a.ElementsByRow.Min);
  Println('Среднее по B:', b.ElementsByRow.Average);
  Print('Сумма по колонкам для A:');
  a.Cols.Select(t -> t.Sum).Println;
  Print('Сумма кубов модулей элементов последней строки A:');
  a.RowSeq(a.RowCount - 1)
      .Select(t -> Abs(t ** 3)).Sum.Println;
  Print('Максимальный элемент в B:');
  var max := b.ElementsWithIndices.MaxBy(t -> t[0]);
  Writeln('B[', max[1]+1, ',', max[2]+1, '] =', max[0]);
  Writeln('Выборка строк и столбцов матрицы A по индексам');
  a.MatrSlice(|0, 2|, ArrGen(2, i -> 2 * i + 1)).Println(4);
  Writeln('Срез матрицы B');
  b.MatrSlice(0, 1, 3, 5).Println(4)
end.
```

```
   1  46  22 -10  33
 -34 -41  44   4 -18
 -39 -20  36  -4 -22
--------------------
  47  47  -4  34 -10  -2  29
 -40  37  11 -29 -36 -43 -28
----------------------------
Min(A) = -41
Среднее по B: 0.928571428571429
Сумма по колонкам для A: -72 -15 102 -10 -7
Сумма кубов модулей элементов последней строки A: 124687
Максимальный элемент в B: B[1,1] = 47
```

```
Выборка строк и столбцов матрицы A по индексам
  46 -10
 -20  -4
Срез матрицы B
  34 -10  -2
 -29 -36 -43
```

7.6.1. Цикл по всем элементам матрицы

Процедура .foreach является расширением, позволяющим построчно перебрать все элементы матрицы. Имеются две разновидности, отличающиеся набором параметров лямбда-процедур:

- a.ForEach(T -> ()) применяет указанное лямбда-процедурой действие к каждому элементу *T* матрицы *a*;
- a.ForEach((T, i, j) -> ()) применяет указанное лямбда-процедурой действие к каждому элементу *T* матрицы *a*, предоставляя доступ к его индексам *i* и *j*.

Например, организовать вывод элементов массива в желаемом оформлении можно следующим образом.

```
##
var a := MatrRandom(3, 5, -9999, 9999);
a.ForEach((v, i, j)-> begin
    $'{v,10:# ##0}'.Print; // формат в качестве примера !!!
    if j = a.ColCount-1 then Println
    end)
```

```
   322      4 799      9 543     -1 905     -3 759
 4 847      6 600     -9 545      8 367      8 560
-1 762      5 268     -3 794     -6 507     -4 447
```

7.6.2. Модификация строк и столбцов

Помимо выборки отдельных строк и столбцов, в PascalABC.NET включены расширения для их замены, обмена местами выбранной пары строк/столбцов и преобразования всех элементов матрицы.

- a.ConvertAll(T -> T1) – функция, возвращающая матрицу типа *T1*, в которой каждый элемент матрицы *a* типа *T* преобразован в соответствии с заданным лямбда-выражением;
- a.ConvertAll((T, i ,j) -> T1) – функция, возвращающая матрицу, в которой каждый элемент матрицы *a* преобразован в соответствии с заданным лямбда-выражением, в котором участвуют индексы строки и столбца;

- a.Fill((i, j) -> T) – процедура, заполняющая каждый элемент матрицы *a* в соответствии с заданным лямбда-выражением, в котором участвуют индексы строки и столбца;
- a.SetCol(k, v) – процедура, заменяющая в матрице *a* элементы столбца с индексом *k* элементами массива или последовательности *v* того же типа и размера;
- a.SetRow(k, v) – процедура, заменяющая в матрице *a* элементы строки с индексом *k* элементами массива или последовательности *v* того же типа и размера;
- a.SwapCols(k1, k2) – процедура, обменивающая местами столбцы с индексами *k1* и *k2*;
- a.SwapRows(k1, k2) – процедура, обменивающая местами строки с индексами *k1* и *k2*;
- a.Transform(T -> T) – процедура, преобразующая каждый элемент матрицы *a* в соответствии с заданным лямбда-выражением;
- a. Transform((T, i ,j) -> T) – процедура, преобразующая каждый элемент матрицы *a* в соответствии с заданным лямбда-выражением, в котором участвуют индексы строки и столбца.

В качестве примера создадим целочисленную матрицу A размером 7×7 из случайных чисел на отрезке [-50;50] и найдем среднее значение m среди всех ее элементов. Далее, заменим все элементы, отличающиеся по абсолютной величине от m больше чем на 20, суммой индекса строки и столбца такого элемента, взятой со случайным знаком. В полученной матрице заполним нулями все строки, в которых положительных элементов будет больше, чем прочих. Выведем исходную матрицу, найденное значение m, матрицу, полученную после выборочной замены элементов и результирующую матрицу.

```
begin
  var a := MatrRandom(7, 7, -50, 50);
  a.Println;
  Println('-' * 7 * 4);
  var m := a.ElementsByRow.Average;
  Println('m =', m);
  var rs: () -> integer := () -> Random(1, 2) = 1 ? -1 : 1;
  a.Transform((t, i, j) -> Abs(t - m) > 20 ? rs * (i + j) : t);
  a.Println;
  Println('-' * 7 * 4);
  var b := a.Rows.Select(row -> row.Where(t -> t > 0).Count).ToArray;
  var nCols := a.ColCount;
  var c := ArrFill(nCols, 0);
  for var i := 0 to a.RowCount - 1 do
    if 2 * b[i] > nCols then
      a.SetRow(i, c);
  a.Println
end.
```

```
 -45   5  47   0  10 -22  36
  -4  -2 -23 -36 -16   1  31
 -10 -21  49  20 -40  13  20
 -34 -20  50 -22  47 -45 -50
  -5  39  45  24   7   0  26
   5  -9  -9   6 -13  38 -19
 -24 -33 -50 -21 -16 -23  37
----------------------------
m = -1.14285714285714
   0   5   2   0  10  -5   6
  -4  -2   3   4 -16   1   7
 -10 -21  -4   5  -6  13   8
  -3 -20   5  -6   7  -8  -9
  -5   5  -6  -7   7   0  10
   5  -9  -9   6 -13  10 -19
   6  -7   8 -21 -16  11 -12
----------------------------
   0   0   0   0   0   0   0
   0   0   0   0   0   0   0
 -10 -21  -4   5  -6  13   8
  -3 -20   5  -6   7  -8  -9
  -5   5  -6  -7   7   0  10
   5  -9  -9   6 -13  10 -19
   6  -7   8 -21 -16  11 -12
```

Остановимся на некоторых операторах приведенной программы. Переменная rs – это имя лямбда-функции без параметров, возвраща-

ющей случайное значение типа integer, равное 1 или -1. Процедура Transform преобразует матрицу по указанному в условии задачи правилу: если для некоторого элемента *t* матрицы *Abs(t - m) > 20*, то значение элемента заменяется суммой его индексов, умноженной на значение функции *rs*. Значения элемента массива *b[i]* – это количество положительных элементов в строке и индексом *i*. Массив *c* содержит нули, его размер равен количеству столбцов в матрице. Строки матрицы последовательно просматриваются в цикле ***for***, поскольку нам понадобятся индексы строк в случае замены. В теле этого цикла проверяется заданное условие и при его выполнении делается замена строки посредством *SetRow*. В самом деле, если *b[i]* – количество положительных элементов, а *nCols* – их общее количество, исходное условие выглядит как *b[i] > nCols – b[i]* или *2*b[i] > nCols*.

7.7. Транспонирование матрицы

Транспонированием матрицы в математике называется замена ее строк столбцами. Исходная матрица размером m×n превращается в матрицу размером n×m.

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \end{pmatrix} \rightarrow \begin{pmatrix} a_{00} & a_{10} & a_{20} & a_{30} \\ a_{01} & a_{11} & a_{21} & a_{31} \\ a_{02} & a_{12} & a_{22} & a_{32} \end{pmatrix}$$

Операция транспонирования матрицы *a* может быть выполнена посредством функции *Transpose(a)*, возвращающей транспонированную матрицу.

7.8. Массивы размерности выше двух

Общепринятое название массивов, имеющих размерность выше двух, – многомерные массивы. PascalABC.NET не имеет специальных средств работы с многомерными массивами – реализованы лишь их описание, создание, инициализация и вывод. Также, многомерные массивы можно присваивать и передавать в качестве параметров. При описании и инициализации статических многомерных массивов

нужно задавать границы их индексов по каждому измерению и соответствующий им конструктор массива. Для динамических массивов, как обычно, задается количество элементов по каждому измерению.

```
##
var a: array[1..2, 1..3, 1..2] of integer :=
  (((1, 2), (3, 4), (5, 6)),
  ((7, 8), (9, 10), (11, 12)));
Println(a);
a[1, 2, 2] := 0;
Println(a);
```

```
[[[1,2],[3,4],[5,6]],[[7,8],[9,10],[11,12]]]
[[[1,2],[3,0],[5,6]],[[7,8],[9,10],[11,12]]]
```

7.9. Примеры решения задач

7.9.1. Суммы, произведения, экстремумы

Задача 7.1

В массиве размером 5x6 найти значение максимального элемента, индексы минимального элемента и произведение элементов третьего столбца. Массив заполнить случайными значениями, принадлежащими отрезку [-99; 99].

Формируем массив, выводим его элементы. Далее пользуемся возможностями PascalABC.NET.

```
##
var a := MatrRandom(5, 6, -99, 99);
a.Println;
Println('Максимум =', a.ElementsByRow.Max);
Print('Индексы минимального элемента (счет от 1):');
var s := a.ElementsWithIndices.MinBy(t -> t[0]);
Println(s[1] + 1, s[2] + 1);
Print('Произведение 3-го столбца:', a.Col(4).Product)
```

```
   8  -8 -17 -36 -68  29
 -53  95   5 -26 -44  18
 -74 -13  16  26 -77 -40
  -6 -11 -85  20  37 -26
  20  82  65  10 -48 -58
Максимум = 95
Индексы минимального элемента (счет от 1): 4 3
Произведение 3-го столбца: 409161984
```

В демонстрационных целях решим эту же задачу, используя традиционные циклы. Конечно, мы не будем изображать здесь базовый Паскаль.

```
begin
  var a := MatrRandom(5, 6, -99, 99);
  a.Println;
  var (mx, imin, jmin) := (integer.MinValue, 0, 0);
  for var i := 0 to a.RowCount - 1 do
    for var j := 0 to a.ColCount - 1 do
    begin
      mx := Max(mx, a[i, j]);
      if a[i, j] < a[imin, jmin] then
        (imin, jmin) := (i, j)
    end;
  Println('Максимум =', mx);
  Println('Индексы минимального элемента (счет от 1):',
      imin + 1, jmin + 1);
  var p := 1.0;
  for var i := 0 to a.RowCount - 1 do
    p *= a[i, 4];
  Print('Произведение 3-го столбца:', p)
end.
```

Задача 7.2

В массиве размером m×n найти суммы элементов по каждому столбцу и по каждой строке. Массив заполнить случайными значениями, принадлежащими отрезку [-99; 99].

Воспользуемся расширениями .Rows и .Cols, возвращающими массивы строк и столбцов. Каждый элемент такого массива в свою очередь будет массивом, что позволит применить расширение .Sum.

```
##
var (m, n) := ReadInteger2('Введите m и n:');
var a := MatrRandom(m, n, -99, 99);
a.Println;
Print('Суммы по строкам:');
a.Rows.Select(Строка -> Строка.Sum).Println;
Print('Суммы по столбцам:');
a.Cols.Select(Столбец -> Столбец.Sum).Print
```

```
Введите m и n: 4 5
 -27  96 -26 -50 -45
  74  65 -69 -74  79
 -98 -87  49  78  20
 -17  67  29  58 -55
Суммы по строкам: -52 75 -38 82
Суммы по столбцам: -68 141 -17 12 -1
```

Задача 7.3
В массиве размером m×n найти минимальный элемент среди максимальных элементов, полученных для каждого столбца. Массив заполнить случайными значениями, принадлежащими отрезку [-99; 99].

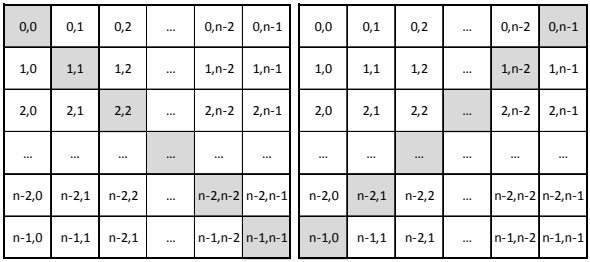
Нужно получить последовательность максимальных элементов по каждому столбцу, а затем найти в ней минимальный элемент. Получение элементов последовательности и нахождение ее элемента с экстремальным значением – однопроходный алгоритм, поэтому можно использовать именно последовательность, а не массив.

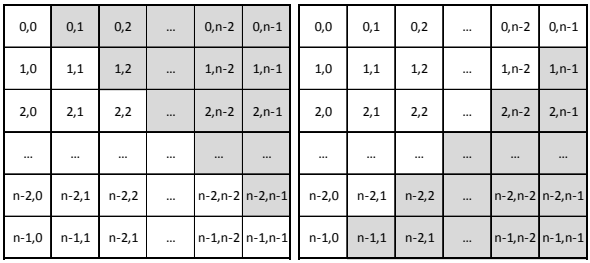
```
##
var (m, n) := ReadInteger2('Введите m и n:');
var a := MatrRandom(m, n, -99, 99);
a.Println;
a.Cols.Select(Столбец -> Столбец.Max).Min.Print
```

```
Введите m и n: 4 7
 -88  67  82 -89  24  81  47
  51 -67 -50  77 -20 -37 -69
  74 -35  61 -22   7  -2  20
  -9  99  97  35 -53 -47  -4
24
```

7.9.2. Диагонали квадратной матрицы

В матрицах различают две основные диагонали. Главная диагональ выходит из левого верхнего угла и, если матрица квадратная, приходит в правый нижний угол. Побочная диагональ выходит из правого верхнего угла и приходит в левый нижний угол. Если матрица прямоугольная, диагонали заканчиваются либо в нижней строке матрицы, либо ограничиваются первым (последним) столбцом. В школьных задачах диагонали прямоугольных матриц встречаются очень редко.





Типовые задачи, в которых фигурируют диагонали, предполагают обработку данных, принадлежащих той или иной области, либо комбинации областей. Например, элементы, расположенные на главной диагонали или выше нее. На рисунке показаны базовые условия, определяющие принадлежность элементов той или иной характерной области. Более сложные условия получаются путем комбинирования базовых.

Для перебора элементов, лежащих на диагонали, достаточно обычного цикла. На главной диагонали матрицы размером *n×n* расположены элементы *a[i,i]*, на побочной – *a[i, n-1-i]*. Для перебора элементов, принадлежащих иным областям, потребуется вложенный цикл.

Задача 7.4

В массиве размером n×n найти разность суммы четных элементов, принадлежащих главной диагонали и суммы нечетных элементов, принадлежащих побочной диагонали. Массив заполнить случайными значениями, принадлежащими отрезку [-99; 99].

Все вычисления делаются в цикле по *i* от 0 до *n-1*. Заводим переменную *s* для результата и обнуляем ее. Если значение *a[i,i]* четное, добавляем его к *s*. Если значение *a[i,n-1-i]* нечетное, вычитаем его из *s*.

```
##
var n := ReadInteger;
var a := MatrRandom(n, n, -99, 99);
a.Println;
var s := 0;
for var i := 0 to n - 1 do
  s := s + if a[i, i].IsEven then a[i, i] else 0 -
       if a[i, n - 1 - i].IsOdd then a[i, n - 1 - i] else 0;
s.Print
```

```
5
  74 -12   8  39   5
 -20  40 -63  -5  69
  88 -70  19 -43 -44
   1 -20 -15   3  80
  72 -76  22 -77  37
95
```

Проверим работу программы ручным счетом. Сумма четных значений на главной диагонали: 74+40=114. Сумма нечетных значений на побочной диагонали: 5+(-5)+19=19. Разность этих сумм 114-19=95. Результат сходится с полученным ответом.

Задача 7.5
В массиве размером n×n найти сумму квадратов элементов, расположенных на главной диагонали и ниже ее. Массив заполнить случайными значениями, принадлежащими отрезку [-99; 99].

В подобных задачах главное – не ошибиться со значением индексов по столбцам, определяющим принадлежность элементов к заданной области.

```
##
var n := ReadInteger;
var a := MatrRandom(n, n, -99, 99);
a.Println;
var s := 0;
for var i := 0 to n - 1 do
  for var j := 0 to i do
    s += Sqr(a[i, j]);
Print(s)
```

```
6
  86   0  -8 -86 -87  -5
 -23  55  77  64  17 -70
  23 -98 -97 -72  16 -40
 -14  44  24 -39  30  11
 -54  27 -77  21 -71   4
  77 -70 -57  46  22 -37
67824
```

7.9.3. Заполнение данными

Такие задачи подразумевают заполнение матрицы значениями, которые определяются определенным алгоритмом и часто бывают связаны с одним или обоими индексами.

> ***Задача 7.6***
> Создать единичную матрицу размером n×n. Единичной называется матрица, на главной диагонали которой расположены единицы, а прочие элементы нулевые.

Можно создать матрицу, по умолчанию заполненную нулями, а затем в цикле поместить единицы на ее главную диагональ.

```
##
var n := ReadInteger;
var E := MatrFill(n, n, 0); // явно заполнили нулями
for var i := 0 to n - 1 do
  E[i, i] := 1;
E.Print(2)
```

```
5
 1 0 0 0 0
 0 1 0 0 0
 0 0 1 0 0
 0 0 0 1 0
 0 0 0 0 1
```

Можно также создать матрицу, а затем преобразовать ее элементы посредством генератора MatrGen.

```
##
var n := ReadInteger;
var E := MatrGen(n, n, (i, j) -> i = j ? 1 : 0);
E.Print(2)
```

Задача 7.7
Дана вещественный массив размером m×n, заполненный случайными числами из промежутка [-9; 9), округленными с точностью до 0.01. Получить новый массив путем деления всех элементов массива на его наибольший по модулю элемент. Значения элементов при вычислении округлять с точностью 0.01.

Вспоминаем, что округление значения *x* с точностью до 0.01 можно получить посредством вызова функции *Round(x, 2)*. А еще, у нас есть возможность сформировать матрицу из одномерного массива подходящей длины.

```
##
var (m, n) := ReadInteger2;
var s := SeqRandomReal(m * n, -9, 9).Select(t -> Round(t, 2));
var a := Matr(m, n, s.ToArray);
a.Println;
Writeln;
var c := a.ElementsByRow.MaxBy(t -> Abs(t));
var b := Copy(a);
b.Transform(t -> Round(t / c, 2));
b.Print
```

```
3 4
   3.55   4.90   6.18   7.95
  -0.22  -3.52  -5.87   2.80
   2.38   3.13   3.63   3.53

   0.45   0.62   0.78   1.00
  -0.03  -0.44  -0.74   0.35
   0.30   0.39   0.46   0.44
```

Поскольку последовательность *s* ленива, памяти она не занимает, но в следующей строке кода служит основой для создания массива. За пределами вызова Matr этот массив не используется и будет отдан сборщику мусора. Это правильный подход: ненавязчиво подсказать программе, что нам далее не нужен какой-то объект. Если вместо последовательности создать массив, он будет «болтаться» до завершения программы. В данном случае это непринципиально, но ведь мы учимся писать программы грамотно! Матрица *b* создана прямым копированием матрицы *a*. Если написать просто *var b := a*, будет передана лишь ссылка и матрица получит второе имя *b*. При этом процедура *Transform* фактически будет работать с матрицей *a*.

Задача 7.8
Заполнить массив размером m×n числами натурального ряда «змейкой». Нечетные строки заполняются в порядке слева направо, четные – в порядке справа налево.

Мы должны получить матрицу, заполненную наподобие приведенной ниже:

```
 1  2  3  4  5
10  9  8  7  6
11 12 13 14 15
20 19 18 17 16
```

Для нечетных строк $A_{i,j}$ = n×i+j+1, для четных $A_{i,j}$ = n×i+n-j. Отсюда получаем конечную формулу (не забываем про нумерацию от нуля):

$$A_{i,j} = n \cdot i + \begin{cases} j+1, & \text{если } i \text{ нечетное} \\ n-j, & \text{в ином случае} \end{cases}$$

Теперь можно писать программный код.

```
##
var (m, n) := ReadInteger2;
var a := new integer[m, n];
for var i := 0 to m - 1 do
  for var j := 0 to n - 1 do
    a[i, j] := n * i + if i.IsEven then j + 1 else n - j;
a.Println
```

```
4 5
   1   2   3   4   5
  10   9   8   7   6
  11  12  13  14  15
  20  19  18  17  16
```

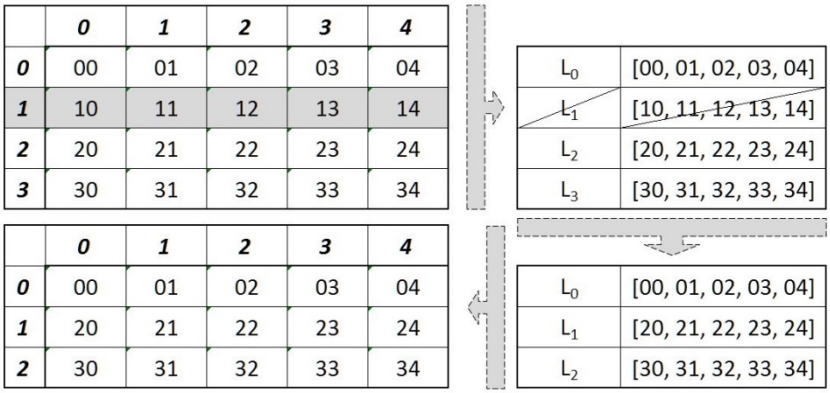
7.9.4. Удаление строк и столбцов

Матрица преобразуется к массиву строк или столбцов, затем нужная строка или столбец удаляется, а из оставшихся снова формируется матрица. Для удаления удобно использовать срезы или списки List.

Задача 7.9

Массив размером m×n заполнен случайными значениями, принадлежащими отрезку [-99; 99]. Удалить из него строку, содержащую максимальный элемент. Если таких строк несколько, удалить первую из них.

Рассмотрим пример удаления второй строки из матрицы размером 4×5. Поскольку индексация строк и столбцов ведется от нуля, вторая строки имеет индекс 1. Преобразуем матрицу в массив из четырех строк так, что каждая строка будет, в свою очередь, массивом длины 5, содержащим элементы из колонок. Полученный массив поместим в список List, удалим из него элемент с индексом 1, после чего из полученного списка снова создадим матрицу. На рисунке представлена схема необходимых преобразований.




```
begin
  var (m, n) := ReadInteger2;
  var a := MatrRandom(m, n, -99, 99);
  a.Println;
  Writeln;
  var im := a.ElementsWithIndices.MaxBy(t -> t[0])[1];
  var L := a.Rows.ToList;
  L.RemoveAt(im);
  a := MatrByRow(L);
  a.Print
end.
```

```
4 5
 -42   2  51  77 -68
 -89  89 -85   5   8
 -55 -67   8 -87  19
 -40  65  53   7 -67

 -42   2  51  77 -68
 -55 -67   8 -87  19
 -40  65  53   7 -67
```

Запоминаем номер строки в переменной *im*. Расширение ElementsWithIndices сформирует последовательность кортежей вида (ЗначениеЭлемента, НомерСтроки, НомерСтолбца). MaxBy по t[0] найдет максимальный по значению элемент, а индекс [1] выберет из кортежа НомерСтроки. Расширение a.Rows создаст массив из строк, который затем размещается в списке List, позволяющем легко удалить нужный элемент. Удаление по индексу производится посредством расширения RemoveAt. Остается снова построчно собрать матрицу при помощи функции MatrByRow.

Более традиционное решение задачи будет состоять в построчном копировании данных во временную матрицу *c* и последующем присваивании *a := c*.

```
begin
  var (m, n) := ReadInteger2;
  var a := MatrRandom(m, n, -99, 99);
  a.Println;
  Writeln;
  var im := a.ElementsWithIndices.MaxBy(t -> t[0])[1];
  var c := new integer[m - 1, n];
  for var i := 0 to im - 1 do
    c.SetRow(i, a.Row(i));
  for var i := im + 1 to m - 1 do
    c.SetRow(i - 1, a.Row(i));
  a := c;
  a.Print
end.
```

Задача 7.10

Массив размером m×n заполнен случайными значениями, принадлежащими отрезку [-99; 99]. Добавить строку, содержащую суммы элементов по колонкам.

Поскольку матрицы хранятся построчно, здесь будет достаточно увеличить количество строк на единицу, а потом заменить элементы последней строки требуемыми суммами. Добавленная строка будет автоматически заполнена нулями и не повлияет на вычисление сумм по колонкам. Но если к задаче применяются эти странные требования ЕГЭ по явной инициализации, надо убрать комментарии из четвертой снизу строки кода программы.

```
##
var (m, n) := ReadInteger2;
var a := MatrRandom(m, n, -99, 99);
a.Println;
Writeln;
SetLength(a, m + 1, n);
//a.SetRow(m, ArrFill(n, 0)); //снять комментарий "для ЕГЭ"
a.SetRow(m, a.Cols.Select(Колонка -> Колонка.Sum));
a.Print
```

```
4 5
  -9 -78  28 -47 -84
 -86   4 -74 -13  91
 -29  -1 -74 -45 -41
 -15 -16 -53 -67 -24

  -9 -78  28 -47 -84
 -86   4 -74 -13  91
 -29  -1 -74 -45 -41
 -15 -16 -53 -67 -24
-139 -91-173-172 -58
```

Задача 7.11

Массив размером m×n заполнен случайными значениями, принадлежащими отрезку [-99; 99]. Добавить колонку, содержащую построчные суммы элементов.

И здесь нас выручит процедура SetLength – увеличим количество столбцов на 1. Дальнейшее решение очень схоже с решением предыдущей задачи.

```
##
var (m, n) := ReadInteger2;
var a := MatrRandom(m, n, -99, 99);
a.Println(5);
Writeln;
SetLength(a, m, n + 1);
a.SetCol(n, ArrFill(m, 0)); //снять комментарий "для ЕГЭ"
a.SetCol(n, a.Rows.Select(Строка -> Строка.Sum));
a.Print(5)
```

```
4 5
   76   25  -31  -50   -6
   27   26  -28   80   95
  -84  -26   68   39  -89
  -96   64   17   95  -36

   76   25  -31  -50   -6   14
   27   26  -28   80   95  200
  -84  -26   68   39  -89  -92
  -96   64   17   95  -36   44
```

7.9.5. Перестановки строк и столбцов

Задача 7.12

Переставить строки матрицы C размера m×n: первую с последней, вторую с предпоследней и т.д. В полученной матрице умножить каждый элемент третьей сверху строки на сумму элементов предпоследнего столбца. Значения m и n ввести с клавиатуры, элементы матрицы получить по формуле $C_{ij} = Sin(2i-j)$ и округлить до трех знаков после запятой.

```
begin
  var (m, n) := ReadInteger2;
  var c := MatrGen(m, n, (i, j)-> Round(Sin(2 * i - j), 3));
  c.Println(7, 3);
  Println('-' * 7 * n); // разделитель
  for var i := 0 to (m - 1) div 2 do
    c.SwapRows(i, m - i - 1);
  c.Println(7, 3);
  Println('-' * 7 * n);
  var s := c.Col(n - 2).Sum;
  c.SetRow(2, c.Row(2).Select(t -> t * s));
  c.Println(7, 3)
end.
```

```
5 8
  0.000 -0.841 -0.909 -0.141  0.757  0.959  0.279 -0.657
  0.909  0.841  0.000 -0.841 -0.909 -0.141  0.757  0.959
 -0.757  0.141  0.909  0.841  0.000 -0.841 -0.909 -0.141
 -0.279 -0.959 -0.757  0.141  0.909  0.841  0.000 -0.841
  0.989  0.657 -0.279 -0.959 -0.757  0.141  0.909  0.841
-------------------------------------------------------
  0.989  0.657 -0.279 -0.959 -0.757  0.141  0.909  0.841
 -0.279 -0.959 -0.757  0.141  0.909  0.841  0.000 -0.841
 -0.757  0.141  0.909  0.841  0.000 -0.841 -0.909 -0.141
  0.909  0.841  0.000 -0.841 -0.909 -0.141  0.757  0.959
  0.000 -0.841 -0.909 -0.141  0.757  0.959  0.279 -0.657
-------------------------------------------------------
  0.989  0.657 -0.279 -0.959 -0.757  0.141  0.909  0.841
 -0.279 -0.959 -0.757  0.141  0.909  0.841  0.000 -0.841
 -0.784  0.146  0.942  0.871  0.000 -0.871 -0.942 -0.146
  0.909  0.841  0.000 -0.841 -0.909 -0.141  0.757  0.959
  0.000 -0.841 -0.909 -0.141  0.757  0.959  0.279 -0.657
```

Здесь единственный подвох – случай с нечетным количеством строк *m*. Нужно сделать *m/2* перестановок строк. «Третья сверху» строка будет иметь индекс 2, а предпоследний столбец – индекс *n-2*.

7.9.6. Выборка строк и столбцов

Задача 7.13

Массив *a* размером m×n заполнен случайными значениями, принадлежащими отрезку [-99; 99]. Получить одномерный массив *b*, содержащий упорядоченные по возрастанию отрицательные элементы массива *a*, и одномерный массив *c*, содержащий нечетные элементы массива *a*, записанные в порядке просмотра по столбцам.

Задача простая и ее решение не должно вызвать никаких проблем.

```
##
var (m, n) := ReadInteger2;
var a := MatrRandom(m, n, -99, 99);
a.Println;
var s := a.ElementsByCol; // ленивая последовательность
var b := s.Where(t -> t < 0).Order.ToArray;
b.Println;
var c := s.Where(t -> t.IsOdd).ToArray;
c.Print
```

```
4 5
 -65 -61  46 -21  60
  12 -50  13  23  63
 -52 -15 -48  90 -13
  68  14 -70 -93  65
-93 -70 -65 -61 -52 -50 -48 -21 -15 -13
-65 -61 -15 13 -21 23 -93 63 -13 65
```

Задача 7.14

Массив *a* размером m×n заполнен случайными значениями, принадлежащими отрезку [-99; 99]. Вычеркнуть строки, начинающиеся с нечетного элемента. Затем вычеркнуть столбцы, заканчивающиеся четным элементом.

Конечно, вычеркивать строки и столбцы матрицы неэффективно. Гораздо проще скопировать в новую матрицу нужные строки и столбцы, но для этого нужно получить их индексы, инвертируя условия вычеркивания.

```
begin
  var (m, n) := ReadInteger2;
  var a := MatrRandom(m, n, -99, 99);
  a.Println;
  Writeln;
  var v := a.Col(0).Indices(t -> t.IsEven).ToArray;
  a := a.MatrSlice(v, Arr(0..n - 1));
  (m, n) := (a.RowCount, a.ColCount);
  v := a.Row(m - 1).Indices(t -> t.IsOdd).ToArray;
  a := a.MatrSlice(Arr(0..m - 1), v);
  a.Print
end.
```

```
4 5
 -46 -29   3  46 -45
   3 -61  36  47 -83
 -16  -8 -72  19  -3
  18 -34  55  10  67

   3 -45
 -72  -3
  55  67
```

В основе алгоритма лежит использование среза матрицы, позволяющего выбрать строки и столбцы, из которых формируется новая матрица. Индексы этих строк и столбцов задаются в виде одномерных массивов. Пока не будут выбраны строки, нельзя определить,

последнюю из них, чтобы получить условие отбора столбцов, поэтому вначале выполняем срез строк, оставляя все колонки, а затем выполняем второй срез, оставляя только нужные из них. Массив *v* служит для хранения индексов, определяющих срез.

В приведенном примере в массив *v* первоначально будут помещены индексы строк 0, 2 и 3. Лишь строка с индексом 1 подлежит вычеркиванию, как начинающаяся с нечетного элемента (3). Расширение MatrSlice вернет матрицу

```
-46 -29   3  46 -45
-16  -8 -72  19  -3
 18 -34  55  10  67
```

Далее, в массив *v* попадут индексы столбцов 2 и 4, поскольку эти столбцы заканчиваются нечетными элементами (55 и 67). И затем расширение MatrSlice вернет матрицу

```
  3 -45
-72  -3
 55  67
```

7.9.7. Матрицы и сортировка

Задачи сортировки в матрицах, как правило, связаны с перестановкой строк или столбцов так, чтобы элементы выбранного столбца или строки оказались упорядоченными по некоторому ключу.

Задача 7.15

Массив *a* размером m×n заполнен случайными значениями, принадлежащими отрезку [-99; 99]. Переставить его столбцы так, чтобы элементы второй строки оказались упорядоченными по неубыванию своих значений.

Ключ сортировки – значение элементов строки номер два (индекс строки равен 1). Выделим эту строку посредством расширения a.Row(1), сформируем кортеж из пар (значение, индекс), упорядочим его элементы по неубыванию значений и построим массив индексов *b*, определяющий требуемую расстановку столбцов. Далее получим срез матрицы, указав в качестве массива индексов столбцов массив *b* и поместим этот срез на место исходной матрицы.

Рассмотрим пример. Пусть дан массив a размером 4×5

```
 72  15  92  67  83
 60  19  -9  44  -3
 42 -69  51  18 -63
 55 -23  43  68  27
```

Создадим последовательность кортежей (значение, индекс) на основе его второй строки

```
(60,0) (19,1) (-9,2) (44,3) (-3,4)
```

Отсортируем ее по неубыванию значений (первому элементу)

```
(-9,2) (-3,4) (19,1) (44,3) (60,0)
```

Выделим из кортежей вторые элементы (индексы)

```
2 4 1 3 0
```

Теперь у нас есть значения индексов колонок и можно строить срез.

```
begin
  var (m, n) := ReadInteger2;
  var a := MatrRandom(m, n, -99, 99);
  a.Println;
  Writeln;
  var b := a.Row(1)
      .Select((v, j) -> (v, j))
      .OrderBy(t -> t[0])
      .Select(t -> t[1])
      .ToArray;
  a := a.MatrSlice(Arr(0..m - 1), b);
  a.Print;
end.
```

```
4 5
 72  15  92  67  83
 60  19  -9  44  -3
 42 -69  51  18 -63
 55 -23  43  68  27

 92  83  15  67  72
 -9  -3  19  44  60
 51 -63 -69  18  42
 43  27 -23  68  55
```

Задача 7.16
Массив *a* размером m×n заполнен случайными значениями, принадлежащими отрезку [-99; 99]. Переставить его строки так, чтобы построчные средние значения оказались упорядоченными по невозрастанию.

Ключ сортировки – средние значения сумм по каждой строке. Но так уж ли нужны нам эти средние значения? Вспоминаем, что среднее значения ряда чисел равно их сумме, деленной на количество чисел. У нас количество чисел постоянно и равно *n*. Следовательно, что упорядочивать по средним, что по суммам – разницы нет. Суммы целочисленные и их быстрее получить, поэтому можно сделать выбор в пользу сумм. Алгоритм решения этой задачи очень похож на алгоритм, использованный в предыдущей задаче.

```
begin
  var (m, n) := ReadInteger2;
  var a := MatrRandom(m, n, -99, 99);
  a.Println;
  Writeln;
  var ИндексыСтрок := a.Rows
      .Select((Строка, Индекс) -> (Строка.Sum, Индекс))
      .OrderByDescending(Кортеж -> Кортеж[0])
      .Select(Кортеж -> Кортеж[1])
      .ToArray;
  a := a.MatrSlice(ИндексыСтрок, Arr(0..n - 1));
  a.Print
end.
```

```
4 5
  97 -40  94  87 -85
  35 -67  23 -66  47
  94 -62  85 -64  94
  25  60  29 -81  52

  97 -40  94  87 -85
  94 -62  85 -64  94
  25  60  29 -81  52
  35 -67  23 -66  47
```

7.9.8. Матрицы в подпрограммах

Задача 7.17
Даны массивы *a* и *b* размером m×n каждый, заполненные случайными значениями, принадлежащими отрезку [-99; 99]. Получить массив *c*, каждый элемент которого вычисляется по формуле $c_{ij} := 2 \times a_{ij} - 3b_{ij}$.

Создание исходных массивов оформим функцией – это необязательно, но наглядно. Преобразование легко выполняется посредством расширения Transform.

```
function СоздатьМассив(Строк, Столбцов: integer) :=
    MatrRandom(Строк, Столбцов, -99, 99);

begin
  var (Строк, Столбцов) := ReadInteger2;
  var a := СоздатьМассив(Строк, Столбцов);
  a.Println(5);
  Writeln
  var b := СоздатьМассив(Строк, Столбцов);
  b.Println(5);
  Writeln
  var c := new integer[Строк, Столбцов];
  c.Transform((v, i, j) -> 2 * a[i, j] - 3 * b[i, j]);
  c.Print(5)
end.
```

```
3 6
   34  -26   61  -81  -17  -68
  -64   82  -34   79   57    4
   15  -47  -62  -70  -91  -15

   98  -92   -6  -63   55   99
   46  -53   49  -64    9   24
  -47  -17  -44   16  -77   59

 -226  224  140   27 -199 -433
 -266  323 -215  350   87  -64
  171  -43    8 -188   49 -207
```

Задача 7.18

В поезде 18 вагонов, в каждом из которых 36 мест. Информация о проданных на поезд билетах хранится в двумерном массиве, номера строк которых соответствуют номерам вагонов, а номера столбцов – номерам мест. Если билет на то или иное место продан, то соответствующий элемент массива имеет значение 1, в противном случае – значение 0. Составить программу, определяющую пять вагонов с наибольшим количеством свободных мест.

Информацию о билетах будем хранить в матрице размером 18×36, заполнив его при помощи датчика случайных чисел. Максимально используем функции.

```
function ЗаполнениеВагонов := MatrRandom(18, 36, 0, 1);

function СвободныеМеста(Заполнение: array [,] of integer) :=
    Заполнение.Rows
      .Select(Вагон -> Вагон.Count(Место -> Место = 0))
      .Numerate.OrderByDescending(t -> t[1])
      .Select(t -> t[0]).Take(5).ToArray;

begin
  var РезультатПродаж := ЗаполнениеВагонов;
  РезультатПродаж.Println(2);
  СвободныеМеста(РезультатПродаж).Print
end.
```

```
1 0 1 0 0 1 1 1 1 0 0 0 1 0 0 1 0 1 1 0 1 0 0 0 1 0 1 0 0 1 0 0 1 0 1 0
0 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 1 0 0 0 0 1 0 1 0 0 1 1 0 0 1 1 0 1 1 0 1
1 1 1 0 1 0 1 0 0 0 0 0 1 0 0 1 1 0 0 1 1 0 1 1 1 0 0 1 0 1 1 1 0 0 0 0
0 0 0 1 1 0 0 1 0 0 1 1 1 0 1 1 1 1 0 1 0 0 1 1 0 1 0 0 1 1 0 0 0 1 0 0
1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 1 0 0 1 1 1 1 0 0 1 0 0 1 0 0 1 0 0 0 1
0 1 1 0 0 1 1 1 0 1 1 1 0 1 1 0 1 1 1 1 0 1 0 1 0 1 0 0 1 1 1 1 0 0 1 0 0 0
0 1 0 1 0 1 1 1 0 0 0 0 1 1 0 0 1 1 0 0 0 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0 1
1 1 0 1 0 0 0 0 1 1 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 1
1 0 1 0 1 0 0 1 0 1 1 1 0 0 1 0 0 0 1 0 1 1 1 0 1 0 0 1 0 0 0 0 0 1 0 1 1
0 0 0 1 1 1 0 0 0 1 1 1 0 0 1 0 1 0 1 0 1 1 0 1 1 1 1 0 1 0 1 1 0 1 0 0 0
0 1 1 1 0 0 0 0 1 1 0 1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0 1 1 0 1 0 0 0 0 1
1 0 1 0 0 0 1 1 0 1 0 0 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 0 1 0 1 0 1 0 1 0
0 1 0 1 1 0 0 1 1 1 0 1 1 0 1 0 1 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 1 0 1 1
1 0 0 1 1 1 1 0 0 1 1 1 1 1 0 0 1 1 0 1 1 1 0 0 1 0 1 0 1 0 0 1 1 0 0 0 1 1
0 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 0 0 0 0 0 1 1 1 0 0 1 0 0 1 0 1 0 1 0
0 0 0 0 0 1 0 1 0 1 0 1 1 1 1 1 1 1 0 0 0 0 1 1 0 1 0 1 1 1 1 1 1 0 1 1 0
1 1 0 0 1 0 1 0 1 0 0 0 0 1 0 0 1 1 1 0 1 0 1 1 0 1 0 1 1 0 0 1 0 1 0 1
0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 0 0 1 0 0 0 1 1 0 0 0 0 1 0 0 1 1 1 1
18 2 11 1 8
```

Задача 7.19

Даны три двухмерных массива, заполненные случайными значениями, принадлежащими отрезку [-99; 99]. В каждом массиве найти и вывести координаты минимального элемента.

```
function МойМассив(m, n: integer): array [,] of integer;
begin
  Result := MatrRandom(m, n, -99, 99);
  Result.Println(5)
end;

procedure КоординатыМинимума(Матрица: array [,] of integer);
begin
  var МинимальныйЭлемент := Матрица.ElementsWithIndices
      .MinBy(Элемент -> Элемент[0]);
  Writeln('Минимальный элемент со значением ',
      МинимальныйЭлемент[0], ' в строке ',
      МинимальныйЭлемент[1] + 1, ', столбце ',
      МинимальныйЭлемент[2] + 1);
  Writeln
end;

begin
  loop 3 do
  begin
    var (m, n) := ReadInteger2('Количество строк и столбцов:');
    КоординатыМинимума(МойМассив(m, n))
  end
end.
```

```
Количество строк и столбцов: 3 4
  -24  -16  -20  -69
  -53  -31   -8    6
   85   -7   92  -95
Минимальный элемент со значением -95 в строке 3, столбце 4

Количество строк и столбцов: 5 5
   31  -19   -1  -16  -72
   82  -92   53   13  -18
  -27  -22   22  -64   34
  -96   33   77  -28   -4
  -64   68   82   39   97
Минимальный элемент со значением -96 в строке 4, столбце 1

Количество строк и столбцов: 1 12
  -61  -84  -18   11   43   -6  -43  -61   77   56   60   50
Минимальный элемент со значением -84 в строке 1, столбце 2
```

7.10. Вместо заключения

Здесь заканчивается материал второй части книги. В третьей части вы познакомитесь с обработкой символов и строк, пользовательскими типами данных, линейными списками – стеком и очередью, а также научитесь писать собственные модули и работать с библиотеками. Знание материала третьей части может оказаться полезным в связи с переводом сдачи ЕГЭ по предмету «Информатика и ИКТ» на компьютерную основу. Появилась работа с файлами, обработка символов и строк. Но все же, материал третьей части в основном адресован тем, кто решит глубже изучить программирование.

Приложение

В приложении…

Чтение программ Turbo/Free Pascal

В КИМ ЕГЭ и ОГЭ прошлых лет приводились фрагменты текстов на Паскале с использованием массивов. Тексты на Паскале в КИМ даются в самой примитивной его версии, Turbo Pascal (далее – TP). Компилятор TP был выпущен в 1983 году и просуществовал 20 лет, после чего был вытеснен Borland Delphi. Все прочие компиляторы и интерпретаторы, использующие Паскаль, считаются в рамках изучаемого в школах подмножества языка совместимыми с TP. В демонстрационных версиях ФИПИ на 2021 год таких заданий нет. Но поскольку в кодификаторе ЕГЭ массивы упоминаются, имеется некоторая вероятность появления фрагментов программ с массивами.

Приводимый материал поможет разобраться, как в языке TP описываются подпрограммы и массивы, а также освоить особенности работы с ними. Это совсем несложно, поскольку языковые средства TP весьма скромны.

Вспомним структуру типичной программы на TP:

```
program Vasya; { заголовок программы }
const описание констант; { раздел описания констант }
type описание типов; { раздел описания типов }
{ здесь описываются подпрограммы }
var описание переменных; { раздел описания переменных }
begin
    тело программы
end.
```

§1. Размеры и типы массивов

Массивы в TP всегда статические, т.е. имеют фиксированные границы индексов, причем эти границы должны быть заданы константами до описания массива. Для этого используется раздел констант, которая начинается служебным словом **const**.

Если массив планируется передавать в подпрограмму в качестве параметра, обязательно объявляется ***тип***, который затем приписывается массиву и делается это в разделе **type**.

В программе на ТР при отсутствии подпрограмм массивы описываются примерно так:

```
const
  m = 10;
  n = 7;

var
  a: array[1..n] of integer;
  b, c: array[1..m, 1..n] of integer;

begin
  { тело программы }
end.
```

Если подпрограммы присутствуют, описание массивов несколько усложняется:

```
const
  m = 10;
  n = 7;

type
  arr1 = array[1..n] of integer;
  arr2 = array[1..m, 1..n] of integer;

procedure P(k: integer; p: arr2; var q: arr1);
begin

end;

var
  a: arr1;
  b, c: arr2;

begin
  { тело программы }
  P(5, c, a);
  P(n, b, a);
end.
```

Здесь в разделе **type** объявлены два типа: *arr1* и *arr2*. Эти типы далее используются в описании формальных параметров процедуры и в описании массивов – фактических параметров. Новые типы для массивов в ТР приходится вводить с тем, чтобы обеспечить совпаде-

ние типов фактических и формальных параметров. Возможно это удивит, но если написать

```
var
  x: array[1..5] of integer;
  y: array[1..5] of integer;
```

то по правилам языка TP это будут массивы **разного** типа! Но если описать массивы как *x, y:* ***array****[1..5]* ***of integer*** их тип будет одинаков. Поскольку фактические и формальные параметры-массивы описываются в разных местах, их типы не совпадут. Вот и вводят специальные типы для параметров-массивов. Компилятор тут особо не мудрствует. Он требует, чтобы в качестве формального параметра было указано имя типа, а статический массив с описанием границ – это не имя.

В PascalABC.NET при использовании динамических массивов игры с типами отсутствуют и, к примеру, все **array of integer** имеют один и тот же тип.

§2. Особенности функций

Функции в TP могут возвращать только простые переменные, но не массивы. В отличие от PascalABC.NET в функциях не определена переменная **Result** – вместо нее используется имя функции. Появление переменной **Result** в правой части выражений PascalABC.NET означает всего лишь текущее значение этой переменной, а в TP использование имени функции в правой части трактуется, как рекурсивный вызов. Для примера рассмотрим функцию, вычисляющую факториал числа *n*, равный произведению натуральных чисел от 1 до *n* включительно. При *n*<2 факториал равен 1, в противном случае он находится по формуле *n! = n×(n-1)*!

```
function F(n: longint): longint;
begin
  if n < 2 then
    F := 1
  else
    F := n * F(n-1)
end;
```

```
begin
  Write('10! = ', F(10))
end.
```

Как видите, ничего особого в программе на ТР нет. Вместо **Result** в теле рекурсивной функции используется ее имя *F*. В PascalABC.NET эквивалентный код выглядел бы так:

```
function F(n: integer): integer;
begin
  if n < 2 then
    Result := 1
  else
    Result := n * F(n - 1)
end;

begin
  Write('10! = ', F(10))
end.
```

Рассмотрим вариант программы, не использующий рекурсию.

```
function F(n: longint): longint;
var
  i: integer;
  p: longint;

begin
  p := 1;
  for i := 2 to n do
    p := p * i;
  F := p
end;

begin
  Write('10! = ', F(10))
end.
```

В описании функции появился раздел **var** для переменных, которые используются в теле функции. Вместо Result в теле функции используется отдельная переменная. Результат, возвращаемый функцией, присваивается имени этой функции.

И в этом случае эквивалентный код PascalABC.NET записывается без труда. Заменяем целочисленные типы на **integer**, вместо *p* пишем **Result**, а затем убираем все лишнее.

```
function F(n: integer): integer;
begin
  Result := 1;
  for var i := 2 to n do
    Result := Result * i;
end;


begin
  Write('10! = ', F(10))
end.
```

§3. Задание 6 ЕГЭ-2021 (базовый, 4 мин)

Здесь требуется понять алгоритм и выполнить его, т.е. немного «побыть компьютером».

```
var s, n: integer;
begin
  readln (s);
  n := 1;
  while s < 51 do
  begin
    s := s + 5;
    n := n * 2
  end;
  writeln(n)
end.
```

«Определите, при каком наименьшем введённом значении переменной s программа выведет число 64.»

В этом коде нет никаких особенностей ТР, лишь описания переменных вынесены за основную программу.

В приведенном коде выводится значение *n*, т.е. нужно определить, при каком значении *s* будет получено *n* = 64. Заголовок цикла **while** говорит о том, что цикл будет выполняться лишь пока *s* не превышает 51, поэтому рассматривать значения *s*, большие 51, смысла нет. В теле цикла значение *s* увеличивается на 5, а *n* удваивается, проходя последовательно значения 2, 4, 8, … Перед входом в цикл *n* = 1, поэтому номер прохода по циклу равен показателю степени двойки. Нужно получить значение $n = 64 = 2^6$, откуда следует, что по циклу делается

6 проходов. При этом значение *s* увеличивается на 6×5 = 30. Отсюда легко найти исходное значение *s* = 51 – 30 = 21.

Чтобы проверить полученное решение на компьютере, можно вместо ввода *s* организовать цикл перебора значений, например, от -1000 до 1000. Потребуется добавить в код семь строк и изменить две, так что вы можете не уложиться в рекомендованные 4 минуты. Но это будет ваш выбор. Оцените свои программистские навыки перед тем, как пытаться сделать проверку подобным образом.

```
var s, n: integer;
begin
  for var i := -1000 to 1000 do //
  begin //
    s := i; //
    n := 1;
    while s < 51 do
    begin
      s := s + 5;
      n := n * 2
    end;
    if n = 64 then //
    begin //
      writeln(n, ' ', i); //
      break //
    end //
  end //
end.
```

В приведенном коде добавленные и измененные строки в конце помечены комментариями. Программа выведет значения 64 21, которое показывает, что значение *n* = 64 впервые достигается при *i* = 21. Но в программе *s* получает значение *i*, поэтому 21 и есть то значение, которое следует записать в качестве ответа.

§4. Задание 22 ЕГЭ-2021 (повышенный, 7 мин)

Проверяется умение анализировать алгоритм, содержащий ветвление и цикл.

«Ниже … записан алгоритм. Получив на вход число x, этот алгоритм печатает два числа: L и M. Укажите наибольшее число x, при вводе которого алгоритм печатает сначала 4, а потом 5.»

```
var x, L, M, Q: integer;
begin
  readln(x);
  Q := 9;
  L := 0;
  while x >= Q do
  begin
    L := L + 1;
    x := x - Q;
  end;
  M := x;
  if M < L then
  begin
    M := L;
    L := x;
  end;
  writeln(L);
  writeln(M);
end.
```

В коде нет никаких особенностей ТР, лишь описания переменных вынесены за основную программу.

Анализ ввода и вывода позволяет переформулировать поставленную задачу: требуется определить, какое значение x нужно ввести, чтобы получить $L = 4$ и $M = 5$.

Переменная Q получает начальное значение 9, которое в программе не изменяется. Следовательно, цикл выполняется, пока $x \geqslant 9$.

В теле цикла значение L увеличивается на 1. Первоначально $L = 0$, так что можно считать L счетчиком количества проходов по циклу. В теле цикла значение x уменьшается на $Q = 9$. Это означает, что введенное значение x уменьшится на $9 \times L$.

Переменная M получит значение x. Если M при этом окажется меньше L, то M получит значение L, а M получит значение x. Но ведь это всего лишь обмен значениями M и L, так что окончательно M – большее из значений x и L, а L – меньшее из этих значений.

Итак, что делает приведенный код? Из введенного значения x он последовательно вычитает число 9, пока x не станет меньше 9, а

затем выводит значения *x* и количество сделанных вычитаний в порядке возрастания.

Анализ показывает, что либо *x* - 9×4 = 5, либо *x* - 9×5 = 4. В первом случае получаем *x* = 41, во втором *x* = 49. Выбираем большее значение и окончательно *x* = 49.

Чтобы проверить полученное решение на компьютере, вместо ввода *x* организуем цикл перебора значений, например, от 1000 до -1000.

```
var x, L, M, Q: integer;
begin
  for var i := 1000 downto -1000 do
  begin
    x := i;
    Q := 9;
    L := 0;
    while x >= Q do
    begin
      L := L + 1;
      x := x - Q;
    end;
    M := x;
    if M < L then
    begin
      M := L;
      L := x;
    end;
    if (L = 4) and (M = 5) then
    begin
      Print(i);
      break
    end
  end
end.
```

Программа выведет значение 49, подтвердив правильность решения.