Александр Осипов

PascalABC.NET: выбор школьника
Часть 3

Александр Осипов

PascalABC.NET: выбор школьника
Часть 3

Символы и строки
Типы данных
Множества
Стеки, очереди, словари
Модули
Приложение (ОГЭ и ЕГЭ)

49 типичных задач:
постановка, 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 3. /А. В. Осипов. –
Ростов-на-Дону : – 146 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

Оглавление ... 3
Предисловие разработчика PascalABC.NET ... 7
Предисловие рецензента ... 9
От автора ... 11
Глава 8. Символы и строки ... 13
8.1. Основные понятия ... 15
8.2. Символьный тип данных ... 15
8.2.1. Принадлежность символа к группе ... 18
§1. Является ли символ буквой? ... 18
§2. Является ли символ цифрой? ... 19
§3. Является ли символ буквой либо цифрой? ... 19
§4. Является ли символ пробельным? ... 19
§5. Является ли символ знаком препинаниня? ... 20
§6. Принадлежит ли буква к верхнему регистру? ... 20
§7. Принадлежит ли буква к нижнему регистру? ... 20
§8. Принадлежит ли символ интервалу? ... 20
8.2.2. Операции преобразования символов ... 20
§1. Смена регистра буквенного символа ... 21
§2. Преобразование цифрового символа в число ... 21
§3. Получение символа, соседнего с указанным ... 21
§4. Смещение по символам кодовой таблицы ... 22
§5. Пример программы ... 22
8.2.3. Ввод символов ... 23
8.2.4. Вывод символов ... 25
8.2.5. Примеры решения задач ... 25
8.3. Строки ... 27
8.3.1. Ввод строк ... 29
8.3.2. Вывод строк ... 30
8.3.3. Арифметические операции со строками ... 30
8.3.4. Сравнение строк ... 31
8.3.5. Копирование строк ... 34
8.3.6. Выделение подстроки ... 34
8.3.7. Срезы строк ... 34
8.3.8. Модификация строки ... 35
§1. Смена регистра символов ... 36
§2. Удаление символов в начале и конце ... 36

§3. Удаление подстрок 37
§4. Инверсия 38
§5. Вставка подстроки 39
§6. Замена подстроки 39
8.3.9. Проверки в строке 40
8.3.10. Разбиение строки на слова 40
8.3.11. Сцепление (слияние) строк 42
8.3.12. Поиск в строке 45
§1. Поиск в прямом направлении 45
§2. Поиск в обратном направлении 46
§3. Быстрый поиск всех вхождений подстроки 47
8.4. Регулярные выражения 49
8.4.1. Некоторые метасимволы 50
8.4.2. Некоторые квантификаторы 51
8.4.3. Экранирование 51
8.4.4. Директивы нулевой длины 51
8.4.5. Примеры шаблонов 51
8.4.6. Наличие подстроки в строке 52
8.4.7. Результаты поиска в элементах Match 53
8.4.8. Замена всех подстрок в строке 53
8.4.9. Поиск первого вхождения подстроки 54
8.4.10. Поиск всех вхождений подстроки 54
8.5. Извлечение данных из строк 55
8.5.1. Преобразование числа к строке 55
8.5.2. Преобразование строки к числу 56
§1. Преобразование строки к целому числу 56
§2. Преобразование строки к вещественному числу 57
§3. Преобразование строки к массиву чисел 58
8.5.3. Преобразование строки в массив 58
8.6. Форматирование данных для вывода 59
8.6.1. Выравнивание строки пробелами 60
8.6.2. Составное форматирование 60
§1. Целочисленные данные 61
§2. Вещественные данные с фиксированной точкой 62
§3. Вещественные данные с плавающей точкой 62
§4. Числовые данные в общем формате 62
§5. Формат для шестнадцатеричного представления 63
§6. Функция и статический метод Format 63

§7. Процедура форматного вывода WriteFormat 63
8.6.3. Интерполированные строки 63
8.7. Примеры решения задач 64
Глава 9. Типы данных 71
9.1. Записи 73
9.2. Автоклассы 74
9.3. Тип данных BigInteger 75
9.3.1. Инициализация данных 75
9.3.2. Приведение BigInteger к другому типу 76
Глава 10. Множества 79
10.1. Создание множества Set of T 80
10.2. Создание множества HashSet 82
10.3. Операции над множествами 84
10.3.1. Перебор элементов в цикле foreach 84
10.3.2. Добавление элемента 85
10.3.3. Удаление элемента из множества 85
10.3.4. Проверка наличия элемента 86
10.3.5. Объединение множеств (+) 87
10.3.6. Разность множеств (-) 88
10.3.7. Пересечение множеств (*) 89
10.3.8. Симметрическая разность 90
10.3.9. Сравнение множеств 90
§1. Равенство множеств (=) 90
§2. Неравенство множеств (<>) 91
§3. Строгое вложение (<) 91
§4. Нестрогое вложение (<=) 91
§5. Строго содержит (>) 92
§6. Нестрого содержит (>=) 92
10.3.10. Дополнительные методы HashSet 93
10.4. Примеры решения задач 93
Глава 11. Стеки, очереди, словари 97
11.1. Стек (Stack) 98
11.1.1. Операции для работы со стеком 100
11.1.2. Пример использования стека 101
11.2. Очередь (Queue) 102
11.2.1. Операции для работы с очередью 103
11.2.2. Пример использования очереди 104
11.3. Словари 104

11.3.1. Структура «ключ – значение» ... 105
11.3.2. Создание словаря ... 106
11.3.3. Операции для работы со словарем ... 108
11.3.4. Пример работы со словарем ... 109
Глава 12. Модули ... 111
12.1. Стандартные модули ... 113
12.1.1. Модули для работы с графикой ... 113
§1. Модуль GraphABC ... 113
§2. Модуль GraphWPF ... 114
§3. Модуль ABCObjects ... 114
§4. Модуль WPFObjects ... 114
§5. Модуль Graph3D ... 114
§6. Модуль ABCSprites ... 115
12.1.2. Учебные модули ... 115
§1. Модуль Robot ... 115
§2. Модуль Drawman ... 116
§3. Модуль NumLibABC ... 116
§4. Модуль SF ... 117
§5. Модуль School ... 117
12.2. Примеры решения задач ... 120
Приложение ... 123
П1. Алгоритмы к ЕГЭ ... 124
§1. Минимум и максимум ... 124
§2. Корни квадратного уравнения ... 125
§3. Позиционные системы счисления ... 125
§4. Сумма и произведение для последовательности ... 127
§5. Простые переборные задачи ... 129
§6. Заполнение элементов массивов ... 131
§7. Операции с элементами массива ... 132
§8. Второй экстремум в массиве ... 135
§9. Подсчет числа экстремумов в массиве ... 137
§10. Операции с элементами, отобранными по условию ... 138
§11. Сортировка массива ... 138
§12. Слияние массивов ... 139
§13. Обработка отдельных символов в строке ... 140
§14. Обработка подстрок в строке ... 143
П2. ЕГЭ: перенаправление ввода/вывода ... 146

Предисловие разработчика PascalABC.NET

Книга Осипова А.В. «PascalABC.NET: выбор школьника. Часть 3» – завершающая книга трилогии по языку программирования PascalABC.NET для школьников. Эта книга затрагивает такие важные темы как символы и строки, множества, стеки, очереди и словари. Кроме того, рассмотрены некоторые типовые задачи ЕГЭ и ОГЭ.

Книга получилась более простой, чем предыдущая. Связано это прежде всего с большим количеством несложных примеров, в которых если и используются лямбда-выражения, то текст решения тем не менее понятен.

Строкам и множествам в третьей части уделено значительное внимание – описание их возможностей занимает почти 100 страниц. А вот изложение стеков, очередей и словарей занимает лишь немного более 10 страниц. Связано это с тем, что в школьных задачах стеки и очереди используются крайне редко, а словари обычно заменяются более простыми структурами. В такой деликатной теме как регулярные выражения, на мой взгляд, справочная часть преобладает над примерами и, в частности, упущен такой замечательный пример как разбиение строки на слова с помощью регулярного выражения '\w+', позволяющий вычислять позицию вхождения каждого слова.

Я бы рекомендовал читать данную книгу «от задач» - разбирать текст задач, а затем, если решение непонятно, разбирать соответствующую теорию.

Что касается самих задач, материал абсолютно новый, авторский, в нём соблюдён баланс кода и авторских пояснений. И конечно же невозможно не упомянуть особую манеру автора излагать новые понятия: здесь и экскурс в историю того или иного понятия, и эпиграфы, сопровождающие каждую часть, и обращение автора к новичкам с указанием наиболее частых проблем их понимания и написания кода.

В приложение вынесено более 20 задач, которые могут пригодиться школьникам при подготовке к ОГЭ и ЕГЭ. Очевидно, данных задач

недостаточно для подготовки к соответствующим экзаменам, и мы надеемся увидеть ещё одну книгу – целиком посвящённую задачам ОГЭ и ЕГЭ.

Я бы рекомендовал книгу для старших школьников – прежде всего для знакомства с арсеналом современного программирования. С этой задачей книга справляется на 100%. Книга также 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 третья, заключительная, часть книги А. В. Осипова «PascalABC.NET: выбор школьника». Если сказать кратко, она про типы-контейнеры. Вообще говоря, лично я ужаснулся бы, столкнувшись с необходимостью писать об этом. Очень уж большой объём материала, очень сложно вычленить действительно необходимое содержание. При этом информацию нельзя назвать особенно увлекательной: справочники обычно не являются примером занимательной литературы. Однако автор смог каким-то невероятным образом не превратить третью часть в справочник. Думаю, что помогла ему в этом заранее поставленная цель: книга должна помочь школьнику в подготовке к ЕГЭ по информатике и ИКТ. Этим обусловлен выбор содержания и подбор примеров. Следует признать, что задумка вполне удалась, и школьник, усвоивший предложенный материал, будет снабжён всеми необходимыми инструментами для решения на ЕГЭ задач, связанных с программированием.

Структурированные типы данных в сегодняшней школе на уроках информатики (особенно на базовом уровне) представлены плохо. Многочисленные задачи по программированию если и требуют использования каких-либо структурных типов, то, главным образом, массивов и строк. Это неудивительно, если принять во внимание количество часов, отводимых в стандарте на изучение программирования: за 17 часов в непрофильном классе многие и до строк не добираются. Однако ограничиваться только массивами и строками в современном программировании невозможно. Изучение стандартных контейнеров является необходимым этапом обучения программиста. Надо только понимать, что в книге собран материал, который главным образом нацелен на подготовку к ЕГЭ. Некоторые важные типы данных (например, файлы) не рассматриваются. Но поскольку на экзамене школьнику обязательно придётся получать данные из файлов, в приложении автор показал удобный способ сделать это, не погружаясь в изучение файлового типа.

Ещё Н. Вирт чётко сформулировал: *программа = структуры данных + алгоритмы*. В приложении к третьей части рассмотрены самые необходимые алгоритмы, без знания которых сложно претендовать

на высокую оценку на экзамене. Рассматривая приведённые примеры, мы видим, в каких случаях специализированные структуры данных оказываются удобными. Решения задач приведены как в традиционном стиле (с использованием циклов), к которому ученики привыкли в школе, так и в функциональном. Читатель может убедиться в том, что второй способ лаконичнее первого, и даже понятнее, при условии, что уже изучены методы последовательностей и технология LINQ.

В последней главе бегло рассмотрены модули, имеющиеся в PascalABC.NET. Информации не очень много, но, чтобы начать пользоваться инструментом, нужно, по крайней мере, знать о его существовании. В реальной педагогической практике использование учебных модулей позволяет в значительной степени разнообразить процесс обучения, не зацикливаясь только на вычислительном программировании.

Как мы уже привыкли, каждая глава содержит большое количество примеров. Они не просто иллюстрируют текст, они сами являются текстом. Пропускать примеры, не изучая их подробнейшим образом, это всё равно, что читать в книге только нечётные страницы: много потеряете. Автор потратил немало сил и времени, чтобы подготовить эти примеры. Потребуется много труда, чтобы разобраться в них. Не только в геометрии нет царского пути!

Я надеюсь, что трилогия поможет всем желающим глубже познакомиться с современным программированием и станет для читателя очередной ступенькой в огромный мир информатики.

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

От автора

Если осталось еще что-нибудь доделать, считай, что ничто не сделано

(Марк Анней Лукан, древнеримский поэт)

К первому сентября 2020 года на официальном сайте PascalABC.NET были опубликованы в свободном доступе первая и вторая части книги. Вторая часть сразу писалась для версии PascalABC.NET 3.7, а первая претерпела второе издание для актуализации версии. На содержание обеих частей также оказал влияние факт публикации демонстрационной версии «компьютерного» ЕГЭ-2021 от ФИПИ.

Третья часть сохраняет традицию предыдущих частей: нумерация глав во всей книге едина. Продолжено рассмотрение средств языка PascalABC.NET и решений задач. Освоение этой части книги позволит читателю в полной мере овладеть подмножеством языка, достаточным не только для решения задач ОГЭ и ЕГЭ, но и для олимпиадного программирования. Все, что останется неясным, читатель сможет попытаться найти в книге «PascalABC.NET: Введение в современное программирование», опубликованной 7 октября 2019 года на официальном Интернет-сайте PascalABC.NET (http://pascalabc.net).

Как и в работе над предыдущими частями книги, я постоянно консультировался с руководителем проекта PascalABC.NET Станиславом Станиславовичем Михалковичем, по-прежнему обеспечивающим технический надзор за точностью излагаемого материала. А Вадим Олегович Дженжер продолжал все так же скрупулезно вычитывать тексты и проверять примеры. Считаю своей обязанностью в очередной раз выразить этим людям огромную благодарность за большую проделанную работу и проявленное терпение.

А.Осипов

Глава 8

Символы и строки

В этой главе…

Основные понятия
Символьный тип данных
Строки
Регулярные выражения
Извлечение данных из строк
Форматирование данных для вывода
Примеры решения задач

Символ всегда заключает в себе больше, чем его очевидное и сразу приходящее на ум значение.

Карл Густав Юнг,
швейцарский психолог

Прочтение двух предыдущих частей книги может создать убеждение, что компьютер умеет обрабатывать только числа. Тем более, что в основе работы процессора лежит двоичная арифметика. А еще вы знаете, что компьютер умеет выводить текст, который надо заключать в кавычки. И вы будете правы: действительно, компьютер умеет обрабатывать только числа. По-простому – вычислять. Более того, компьютеры изобретались именно для того, чтобы облегчить человеку вычисления.

Но прошло совсем немного времени и люди подумали, что ведь вот есть алфавит и в нем имеется первая буква, вторая и т.д. Так почему бы не обозначить эти буквы их номерами и не заставить компьютер обрабатывать такие номера? Так появился принцип обработки символов на компьютере. Позднее номера (коды) были присвоены также цифрам и различным значкам. Появились кодовые таблицы.

Современный компьютер оперирует байтами. В байте 8 бит, с помощью которых можно закодировать до $2^8 = 256$ различных символов. Много это? К сожалению, очень мало. Потому что на свете очень много различных языков с самыми различными значками.

Однобайтовая кодовая таблица делится на две половины. Первая – младшая, с десятичными кодами от 0 до 127, отведена для хранения символов, не зависящих от выбранного в операционной системе национального языка. Вторая половина – старшая, с кодами от 128 до 255, содержит символы национального алфавита. Каждая кодовая таблица имеет свое наименование, позволяющее ее выбрать. Но не все так просто! Только для русского языка и на одних только персональных компьютерах исторически существуют три различные кодовые таблицы. Операционная система Windows, предназначенная для России, использует кодовую таблицу с номером 1251 (правильно она называется «кодовая страница 1251»).

Современные операционные системы используют двухбайтовые кодовые таблицы стандарта Unicode. Существуют также кодовые таблицы большей длины. В России «внутри Windows» может работать программная платформа .NET Framework, использующая двухбайтную таблицу. Название PascalABC.NET недвусмысленно намекает, что компилятор работает именно в .NET, и он действительно использует Unicode при хранении символов в оперативной памяти.

8.1. Основные понятия

Текст любой (в том числе, этой) книги набран при помощи букв, цифр и различных знаков, часть из которых даже не отображается (они называются служебными). В совокупности всё это составляет некий ***алфавит***. Алфавит рассматривается как упорядоченный ***набор символов***.

Текст программы также состоит из символов, составляющих алфавит языка программирования. Символы складываются в более крупные конструкции, образуя ***строки символов***. Чтобы отделить в тексте одну строку от другой, договариваются, какие символы будут служить разделителями строк. Обычно для этой цели выбираются один или два служебных символа. Исторически сложилось, что признаком окончания строки являлся переход на новую строчку при печати. В зависимости от типа печатающего устройства, символ завершения строки мог иметь код $13 = 0D_{16}$ – «перевод строки», $10 = 0A_{16}$ – «возврат каретки», либо использовалась их последовательность 0D0A (она и сейчас используется, к примеру, в Windows).

8.2. Символьный тип данных

Данные символьного типа имеют тип **char** и занимают в памяти два байта. Используется кодировка стандарта Unicode. В тексте программы символьная константа (так называемый ***литерал***) всегда заключается в одинарные кавычки. Если надо записать сам символ одинарной кавычки, то кавычка удваивается: ''''.

В программе встречаются понятия как собственно символа, так и его внутреннего кода. В базовом Паскале использовалась однобайтная

кодировка, а кодовая таблица выбиралась в зависимости от локализации операционной системы. Так, Turbo/Borland Pascal и Free Pascal работали с кириллицей («русскими буквами») преимущественно в кодовой таблице CP866, Borland Delphi, работая в среде Windows, мог использовать таблицу CP1251. PascalABC.NET использует кодировку Unicode.

> Новички часто смешивают понятие символа, обозначающего цифру и имеющего тип **char**, с самой этой цифрой, которая может быть неотрицательным однозначным числом одного из целочисленных типов. Важно понимать, что символ – это «картинка», рисунок, изображение числа, и за символом в программе кроется не значение изображенной цифры, а код этого символа, взятый из кодовой таблицы.

Как и целочисленные типы, символьный тип относится к так называемому ***порядковому*** типу данных. Благодаря этому в большинстве операторов, методов и расширений PascalABC.NET на месте, где можно указать данное целого типа, разрешено указывать символьные данные. Это делает изучение данного раздела достаточно комфортным, потому что материал в основе вам уже знаком.

Данные символьного типа, как и любого другого, описываются с указанием ключевого слова **var**:

```
var c1, p135, rz: char; // три переменные
var Символ1: char; // одна переменная
var s: sequence of char; // последовательность символов;
var ca: array[1..35] of char; // статический массив символов
var ar: array of char; // динамический массив символов
var m: array[,] of char; // матрица символов
```

Описание можно соединить с инициализацией:

```
var a: char := 'a'; // тип указан явно
var b:= 'b'; // автовыведение типа
var kt:= ('A', 'B', 'C'); // кортеж из трех символов
```

Далее по тексту этого раздела при описании конструкций языка как символы, так и переменные типа **char** будут условно обозначаться «c», если прямо не указано иное.

Выводить символы можно любыми средствами, которые использовались для вывода других типов данных: Print и Println, Write и Writeln, а также при помощи расширений .Print и .Println. По умолчанию инициализация данных типа **char** производится двоичными нулями, что может создать проблемы при выводе.

Получить **десятичный** код символа можно несколькими способами:
- Ord(c) – код символа с в Unicode (тип **word** длиной 2 байта);
- **char**.Code – то же, точечная нотация;
- OrdAnsi(c) – десятичный код символа в однобайтной кодировке Windows (тип **byte** длиной 1 байт).

Произвести обратную операцию – получить символ по его внутреннему **десятичному** коду, можно тоже несколькими способами:
- Chr(код) – символ с указанным кодом Unicode;
- #код – символ с указанным кодом Unicode; принимает только литерал;
- ChrAnsi(код) – символ с кодом в однобайтной кодировке Windows.

Приведенная ниже программа выводит коды букв кириллицы из таблицы Unicode. Здесь используется знание о том, что кириллица в этой таблице идет подряд, исключая буквы Ё и ё. Генератор Range строит последовательность из 64 символов кириллицы (вот оно – первое проявление порядкового типа данных: Range работает и с символами).

```
## Range('А', 'я').Select(c -> c.Code).Println
```

```
1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051 1052
1053 1054 1055 1056 1057 1058 1059 1060 1061 1062 1063 1064 1065
1066 1067 1068 1069 1070 1071 1072 1073 1074 1075 1076 1077 1078
1079 1080 1081 1082 1083 1084 1085 1086 1087 1088 1089 1090 1091
1092 1093 1094 1095 1096 1097 1098 1099 1100 1101 1102 1103
```

Эту же задачу можно решить и более традиционно, с помощью цикла. Параметр цикла for может быть любого порядкового типа. Следовательно, и типа **char**.

```
##
for var c := 'А' to 'я' do
  Print(Ord(c))
```

Соединяясь друг с другом, символы образуют ***строки***. Будем пока условно воспринимать их, как последовательность символов. А еще лучше – как динамический массив, только с индексами, идущими от единицы. Потому что, написав для строки s, например s[5], получим пятый от начала строки символ. Для соединения символов служат знаки операции «+» и «*». Знак + сцепляет символы между собой, знак * позволяет размножить символ нужное число раз путем умножения его на числовое значение.

```
## Print((2 * ('м' + 'а') + ' ') * 5)
```

```
мама мама мама мама мама
```

А вот еще приятная неожиданность: если с символом сложить число, оно будет автоматически преобразовано к строке символов и эта строка сцепится с указанным в выражении символом.

```
## ('$' + 132.5).Print  // $132.5
```

Поскольку за каждым символом скрывается его код, символы можно сравнивать друг с другом так, как мы сравниваем числовые величины. Один символ больше другого, если больше числовое значение его кода, поэтому истинны отношения 'В' > 'А', '3' < '7', 'Ё' < 'ё'. Подробности можно найти в подразделе 8.3.4.

8.2.1. Принадлежность символа к группе

Анализируя различные символы, мы можем относить их к той или иной группе – буквам, цифрам, знакам препинания, скобкам, пробелам и переносам строк и т.п.

§1. Является ли символ буквой?

Буквами считаются латинские и кириллические буквы в обоих регистрах, а также буквы иных алфавитов, например, греческого. Именно эти буквы PascalABC.NET позволяет использовать в именах объектов программы.

Расширение c.IsLetter возвращает True, если символ *c* принадлежит к группе букв и False в противном случае. Попробуем узнать, сколько «букв» нам может быть доступно в текущей кодовой таблице.

```
## Range(0, 32767).Where(n -> Chr(n).IsLetter).Count.Print // 25450
```

Немало, оказывается! 25.5 тысяч. Жаль, большая часть из них – иероглифы. Но давайте разберемся, как работает эта программа. Range строит последовательность от 0 до 32767 (поскольку коды двухбайтные). Последовательность фильтруется методом Where, использующим выражение Chr(n).IsLetter, которое строит символ с очередным кодом и проверяет, является ли он буквой. Метод .Count возвращает количество элементов последовательности, прошедшее фильтр.

§2. Является ли символ цифрой?

Расширение c.IsDigit возвращает True, если символ *c* принадлежит к группе цифр (0, 1, ... 9) и False в противном случае.

```
##
var a := |'П', 'и', '=', '3', '.', '1', '4'|;
Print('Текст содержит цифр:', a.Count(c -> c.IsDigit))
```

§3. Является ли символ буквой либо цифрой?

Метод **char**.IsLetterOrDigit(c) возвращает True, если символ *c* принадлежит к группе букв или цифр и False в противном случае. Этот метод принадлежит не конкретному объекту, а классу, поэтому мы должны указывать его вместе с именем класса **char**. Такие методы называются ***классовыми*** или ***статическими.***

§4. Является ли символ пробельным?

Пробельные символы – термин, пришедший из типографской практики. Пробел – это пустое место, интервал между символами. Пробельными в типографском и издательском деле называют непечатаемые (и неотображаемые) символы. Символы с десятичными кодами 9 .. 13 относятся к так называемым управляющим символам – раньше они управляли внешними устройствами, такими как механический принтер. Символ с кодом 32 – это пробел.

Статический метод **char**.IsWhiteSpace(c) возвращает True, если символ *c* принадлежит к группе пробельных символов и False в противном случае.

§5. Является ли символ знаком препинаниня?

Статический метод **char**.IsPunctuation(c) возвращает True, если символ *c* принадлежит к группе знаков пунктуации (разделителям) и False в противном случае. Познакомиться с этими символами нам также поможет программа.

```
## (0..1300).Select(n -> Chr(n))
      .Where(c -> char.IsPunctuation(c)).Print
```

```
!"#%&'()*,-./:;?@[\]_{}¡«·»¿;·
```

Здесь просмотр ограничен символом с кодом 1300, чтобы не заполнять вывод знаками, которые в России никогда не используются и скорее всего, корректно не будут отображены при попытке их напечатать. Достаточно убедиться, что среди выведенных символов имеются все употребляемые знаки препинания.

§6. Принадлежит ли буква к верхнему регистру?

Расширение c.IsUpper возвращает True, если символ *c* принадлежит к буквенным символам верхнего регистра (прописным) и False в противном случае. Если символ не является буквой, всегда возвращается False.

§7. Принадлежит ли буква к нижнему регистру?

Расширение c.IsLower возвращает True, если символ *c* принадлежит к буквенным символам нижнего регистра (строчным) и False в противном случае. Если символ не является буквой, всегда возвращается False.

§8. Принадлежит ли символ интервалу?

Конструкция *c* **in** *c1..c2* вернет True, если символ *c* принадлежит группе символов от *c1* до *c2* включительно и False в противном случае.

8.2.2. Операции преобразования символов

К операциям преобразования символов отнесены: смена регистра буквенных символов, переход к следующим и предыдущим символам (в порядке следования их кодов в таблице), а также преобразование символа из группы «цифра» к соответствующему этому символу однозначному числу.

§1. Смена регистра буквенного символа

- c.ToUpper – расширение возвращает буквенный символ *c*, приведенный к верхнему регистру, если он принадлежит к нижнему регистру. В противном случае символ возвращается без изменения;
- UpCase(c) – функция, делающая то же самое;
- c.ToLower – расширение возвращает буквенный символ *c*, приведенный к нижнему регистру, если он принадлежит к верхнему регистру. В противном случае символ возвращается без изменения;
- LowCase(c) – функция, делающая то же самое.

§2. Преобразование цифрового символа в число

Расширение c.ToDigit возвращает буквенный символ *c*, преобразованный к изображаемому им целому неотрицательному однозначному числу типа **integer**. Если символ не является цифрой, т.е. расширение c.IsDigit возвращает для него **False**, при выполнении программы будет выдано сообщение «Ошибка времени выполнения: not a Digit» и программа завершится аварийно.

§3. Получение символа, соседнего с указанным

- c.Pred – расширение возвращает буквенный символ, код которого в кодовой таблице предшествует коду символа *c*;
- Pred(c) – функция, делающая то же самое;
- c.Succ – расширение возвращает буквенный символ, код которого следует за кодом символа *c*;
- Succ(c) – функция, делающая то же самое.

Какой код предшествует символу с кодом 0? Никакой, потому что этим символом начинается кодовая таблица. Поэтому попытка получить Chr(0).Pred при выполнении программы приведет к выдаче сообщения «Ошибка времени выполнения: Значение было недопустимо малым или недопустимо большим для знака». И программа будет аварийно завершена. То же самое произойдет при попытке получить символ, следующий за имеющим код 65535 – последним символом в кодовой таблице.

§4. Смещение по символам кодовой таблицы

- Dec(c) – процедура, заменяющая значение переменной, содержащей символ *c* на символ, предшествующий ему. Ведь это обычная операция декремента, только для символа;
- Dec(c, n) – процедура, заменяющая значение переменной *c*, содержащей символ *c* на символ, находящийся в кодовой таблице на *n* позиций раньше. Тоже декремент;
- Inc(c) – процедура, заменяющая значение переменной, содержащей символ *c* на символ, следующий за ним. Это инкремент.
- Inc(c, n) – процедура, заменяющая значение переменной, содержащей символ *c* на символ, находящийся в кодовой таблице на *n* позиций дальше. И это инкремент.

§5. Пример программы

Задача 8.1
Для массива из n случайных символов, десятичные коды которых принадлежат отрезку [32; 128], выполнить следующие действия:
– для цифровых символов найти и вывести сумму квадратов чисел, которые представляются этими числами;
– найти и вывести общее количество буквенных символов;
– вывести под одному разу все встреченные буквы, приведенные к верхнему регистру и отсортированные по возрастанию.

На первый взгляд, в задаче много незнакомого материала. Но ведь не зря в начале главы 8 отмечалось, что в программе мы можем работать с символами, а компьютер будет обрабатывать их целочисленные коды. А с числами вы уже умеете делать буквально всё!

Запросим значение n. Затем сгенерируем целочисленную последовательность из n чисел, принадлежащих отрезку [32; 128]. Каждый элемент последовательности спроецируем на символ посредством функции Chr. Результат пропустим через фильтр, оставив лишь буквенные и цифровые символы, остальные ведь нам не нужны, зачем же место занимать? Отфильтрованную последовательность сохраним в символьном массиве. Выведем массив на монитор. Далее выполняем задание, используя проецирование, фильтрацию и сортировку.

```
begin
  var n := ReadInteger;
  var a := SeqRandom(n, 32, 128)
      .Select(t -> Chr(t))
      .Where(c -> char.IsLetterOrDigit(c))
      .ToArray;
  a.Println;
  a.Where(c -> c.IsDigit).Select(c -> Sqr(c.ToDigit)).Sum.Println;
  var s := a.Where(c -> c.IsLetter); // только буквы
  s.Count.Println;
  s.Select(c -> c.ToUpper).Distinct.Order.Print
end.
```

```
70
4tZ3bhWpkxYRuOKkOmrYnIxoDjV6Phtef8kMvIxq9qtYLmA
206
41
ABDEFHIJKLMNOPQRTUVWXYZ
```

Попробуйте разобраться в коде самостоятельно. Если у вас нормальная программа чтения pdf-файла, она позволит скопировать этот код и запустить его в PascalABC.NET. Наводите курсор на непонятные места и получайте подсказки. Со своей стороны скажу, почему буквы оказались упорядочены. Ключом сортировки, конечно же, служит код символа. А буквы в кодовой таблице следуют друг за другом в алфавитном порядке. И еще. Расширение .Print по умолчанию выводит элементы символьного массива без пробелов, так что вывод сливается в строку. Хотите выводить пробел – укажите его явно: *.Print(' ')*.

Как отлаживать подобные программы? Укажите небольшую длину строки, например 10. И проверьте результаты вручную.

8.2.3. Ввод символов

В языке Паскаль ввод с клавиатуры всегда завершается нажатием клавиши Enter, а вводимые значения разделяются пробелами или нажатием все той же клавиши Enter. Нажатие клавиши Enter в операционной системе Windows посылает комбинацию десятичных кодов 13 и 10, а в Unix-подобных системах – только 10. Но это тоже символы! Если не принять мер, они будут восприняты, как вводимые данные.

Вводимые данные сначала сохраняются в некоторой области памяти (буфере), а затем анализируются. Каждый раз, когда нужна очередная порция данных, происходит обращение сначала к буферу, а если там данных недостаточно – тогда уже к клавиатуре.

За прием данных с клавиатуры отвечает процедура Read(Элемент1, Элемент2, …) или ее разновидность Readln(Элемент1, Элемент2, …). Процедура Readln (от английских слов Read Line – чтение строки) не воспринимает финальное нажатие клавиши Enter, как данные, более того, она полностью очищает буфер после ввода, даже если были прочитаны не все символы.

При вводе числовых или логических данных наличие или отсутствие служебных символов с кодами 13 и 10 в буфере непринципиально. Ведь в написании данных этих типов нет служебных символов, поэтому они служат лишь границами очередного введенного значения. Это позволяет использовать как Read, так и Readln. Но перед вводом символов буфер должен быть пуст, а очищает его процедура Readln.

> Если перед вводом **символьных** данных вводились еще какие-либо данные, предшествующий ввод должен был осуществляться посредством процедуры Readln, очищающей буфер ввода. Вводимые символьные данные **недопустимо разделять** пробелами или чем-либо еще: любое нажатие клавиши воспринимается, как символ.

Имеются функции, осуществляющие ввод символьных данных с приглашением:

```
ReadlnChar('ТекстПриглашения'); // ввод одного символа
ReadlnChar2('ТекстПриглашения'); // ввод двух символов
ReadlnChar3('ТекстПриглашения'); // ввод трех символов
ReadlnChar4('ТекстПриглашения'); // ввод четырех символов
```

Приглашение может быть опущено и тогда используется формат ReadlnChar, ReadlnChar2, ReadlnChar3, ReadlnChar4.

Если очищать буфер ввода не нужно, используются разновидности без «ln»: ReadChar, ReadChar2, ReadChar3, ReadlnChar4; приглашение ко вводу также может присутствовать.

```
      ##
      var(a, b, c) := ReadChar3('-->');
      Write(a, b, c, ' --> ', c, b, a)
```

```
--> сон
сон --> нос
```

Здесь ввод используется лишь один раз, поэтому можно применять как ReadChar3, так и ReadlnChar3. В результатах видно, что все три символа вводились без пробелов. Для вывода использован оператор Writeln, чтобы символы не разделялись пробелами. Программа вывела принятые символы вначале в порядке их ввода, а затем в обратном порядке.

Основываясь на вводе символов, можно создавать различные формы диалога с программой. Вот код, запрашивающий подтверждение:

```
      var c := ReadlnChar('Введите "д" (да) или "н" (нет)');
```

Теперь достаточно проанализировать введенный символ, например, посредством **case** с **of**. Вы ведь уже догадались, что в case можно делать ветвление и по символам?

```
      case c of
        'д', 'Д': { код для "да" } ;
        'н', 'Н':{ код для "нет"} ;
      else // код для прочего
      end
```

8.2.4. Вывод символов

С выводом все просто: используются уже знакомые процедуры Write, Writeln, Print, Println, а также расширения .Print и .Println. Имеется лишь одна особенность вывода символов при помощи расширений: разделителем по умолчанию является не пробел, а пустой символ.

8.2.5. Примеры решения задач

Задача 8.2

Дана последовательность символов, в конце которой записана звездочка. Получить и вывести с точностью 0.1% соотношение русских и латинских букв в этой последовательности. Букву «Ё» учитывать не требуется.

К сожалению, PascalABC.NET не предоставляет каких-либо особенных функция для ввода последовательностей символов, так что придется использовать цикл **while**. Чтобы не вводить по одному символу на строке, нажимая каждый раз клавишу Enter (ввод в любой версии языка Паскаль завершается нажатием этой клавиши), используем ReadChar, а не ReadlnChar. С одной стороны, это позволит вводить символы произвольными порциями, а с другой – формально выполнить условие задачи, поскольку сохраняется возможность ввода символов по одному. Буквенные символы перекодируем к нижнему (можно верхнему, но какому-то одному) регистру, а затем проверяем принадлежность символа к кириллице или латинице. Количество найденных букв накапливаем в двух счетчиках и после окончания ввода находим искомый процент.

```
begin
  var (Кириллицы, Латиницы, Символ) := (0, 0, ' ');
  while Символ <> '*' do
  begin
    Символ := ReadChar.ToLower;
    if Символ in 'a'..'z' then
      Латиницы += 1
    else if Символ in 'a'..'я' then
      Кириллицы += 1
  end;
  Write(Латиницы / Кириллицы * 100:0:1, '%')
end.
```

```
Программирование в PascalABC.NET - это просто!*
46.2%
```

Можно было, конечно, после * набрать еще символы и убедиться, что они не будут учтены.

Задача 8.3

Назовем словом последовательность непробельных символов. Найти и вывести количество слов, содержащихся в последовательности длиной n символов. Все слова разделены ровно одним пробельным символом, первый и последний символы не являются пробельными.

Задача простейшая. Количество слов на единицу больше количества пробельных символов. Не забываем, что значение n нужно вводить посредством ReadlnInteger, чтобы очистить буфер ввода.

```
##
var n := ReadlnInteger;
var k := 1;
loop n do
  if char.IsWhiteSpace(ReadChar) then
    k += 1;
k.Print
```

```
46
Программирование в PascalABC.NET - это просто!
6
```

8.3. Строки

Мы посвятили работе с символами немало страниц, но при обработке символьной информации значительно удобнее работать со строками.

В PascalABC.NET строка – это последовательность символов практически неограниченной длины (на самом деле, строка не может занимать в памяти больше 2.1 Гбайт), принадлежащая некоторому алфавиту. Символы в строке нумеруются **от единицы** и чтобы обратиться к символу с номером *k* в строке *s* нужно написать *s[k]*.

Имеются два типа строк. Первый тип – строки длиной не превышающей 255 символов, унаследованные от языка Turbo Pascal. Они именуются ***короткими строками***». При описании короткой строки после ключевого слова **string** в квадратных скобках указывается ее максимальная длина. Короткие строки оставлены в языке лишь для совместимости со старыми программами. Второй, основной тип строк – современные строки; при их описании длина не указывается. Мы будем рассматривать именно такие строки.

```
var st, МояСтрока, p18: string; // три строки
var Строчка: string; // одна строка
var sos: sequence of string; // последовательность строк
var ar: array of string; // динамический массив строк
var sh1: string[27]; // короткая строка, максимум 27 символов
```

Описание строк можно соединять с инициализацией:

```
var s1: string := 'Это строка'; // тип указан явно
var s2 := '*** И это строка ***'; // автовыведение типа
var kt := ('Это', 'тоже', 'строка'); // кортеж из трех строк
```

Строка может состоять из одного символа, и даже может иметь и нулевую длину, т.е. не содержать ни одного символа. В последнем случае она записывается парой одиночных кавычек, следующих друг за другом.

Если указано **var** d := 'R', какого типа будет переменная *d*? Типа **char**, конечно же, – тут произойдет автовыведение типа. Если нужен тип **string**, для односимвольного литерала придется указать его явно или использовать явное приведение.

```
var s1 := 'Я'; // символ типа char
var s2: string := 'Я'; // строка string длины 1 – указан тип
var s3 := string('Я'); // строка string длины 1 – явное приведение
```

К любому одиночному символу строки можно обратиться, указав номер его позиции в строке, например *a[5]*, *str15[2 * i + 3]*. Можно выполнить замену одиночного символа в строке при помощи оператора присваивания, например, *a[8] := 'Z'*.

Текущую длину строки, т.е. количество символов в ней, можно определить следующим образом:
- s.Length – свойство, возвращает длину строки *s*;
- Length(s) – функция, делающая то же самое.

Несмотря на то, что символы в строке нумеруются от **единицы**, при использовании любых методов класса **string**, к которому относятся строки, считается что символы в строке индексируются от **нуля**. В функциях и процедурах, не относящихся к этому классу, а также в срезах, символы считаются проиндексированными от **единицы**.

Запомнить это проще всего визуально. Если в коде программы строка *s* используется в записи вида ***string****.Метод(s)* или *s.Метод*, она считается индексированной от нуля. Если «точки» нет, то строка считается индексированной от единицы.

```
##
var s := '123456789';
Print(s.IndexOf('5'), Pos('5', s))
```

```
4 5
```

В примере находится и выводится индекс символа '5' в строке *s*. Поскольку *s.IndexOf* – метод класса **string**, считается, что индексы идут от нуля и выводится значение 4. Функция *Pos* не относится к классу **string**, поэтому выводится порядковый номер, равный 5.

8.3.1. Ввод строк

Строка при вводе с клавиатуры складывается из отдельных символов, поэтому на нее распространяются особенности ввода символов. Ввод завершается нажатием клавиши Enter, что не позволяет записать в строку символы с десятичными кодами 13 и 10 – их при необходимости нужно добавлять путем вставки. Нажатие клавиши пробела не разделяет вводимые строки, а вставляет в строку пробел, поэтому все символы, введенные до нажатия клавиши Enter образуют одну строку. При вводе последовательности или массива строк ввод каждой строки нужно завершать нажатием клавиши Enter.

> Никогда не смешивайте в одном операторе ввод строк со вводом данных другого типа.

Процедуры Read и Readln осуществляют ввод строк, имена которых перечисляются в списке ввода, но лучше не использовать Read при вводе строк, только Readln.

Строки можно вводить и при помощи функций ReadlnString и ReadlnString(p), где p – строка приглашения ко вводу. Эти функции возвращают значение введенной строки и могут быть использованы в выражениях вместо имени строк, что сокращает текст программы.

Существуют также разновидности функций ReadlnString2, ReadlnString3 и ReadlnString4, позволяющие ввести две, три или четыре строки. Все они также поддерживают необязательный параметр, задающий строку приглашения ко вводу.

```
##
var (s1, s2) := ReadlnString2('Введите две строки через Enter:');
('(' + s1 + ')').Println;
('(' + s2 + ')').Println;
('(' + s3 + ')').Println;
```

```
Введите две строки через Enter: первая строка
строка №2
(первая строка)
(строка №2)
```

8.3.2. Вывод строк

Выводить строки можно любыми средствами, которые использовались для вывода других типов данных: Print и Println, Write и Writeln, а также при помощи расширений .Print и .Println. Расширения рассматривают строку, как последовательность отдельных символов, но выводят эти символы не через пробел, как для других типов данных, а через пустой символ. Тем не менее, расширения .Print и .Println могут принимать в качестве параметра строку-разделитель, что иллюстрирует следующая простая программа

```
##
'Тестовая строка'.Println;
'Тестовая строка'.Println(' ');
'Тестовая строка'.Println('*');
'Тестовая строка'.Println(' = ')
```

```
Тестовая строка
Т е с т о в а я   с т р о к а
Т*е*с*т*о*в*а*я* *с*т*р*о*к*а
Т = е = с = т = о = в = а = я =   = с = т = р = о = к = а
```

8.3.3. Арифметические операции со строками

Операция сложения «+», примененная к строкам, объединяет их в общую строку. Если же одним из операндов является значение числового выражения (за исключением типа BigInteger, который будет рассмотрен в следующей главе), а вторым – строка, то числовое значение предварительно преобразуется к строковому виду, а потом выполняется объединение.

В операции умножения «*» один операнд должен быть строкой, второй - целочисленным выражением. В результате строка соединя-

ется сама с собой указанное значением выражения количество раз, давая новую строку. В одном операторе можно записывать и больше одного сложения и/или умножения.

```
## Println('--- 25 + 3 * 8 =' + (25 + 3 * 8) + ' ---')
```

```
--- 25 + 3 * 8 = 49 ---
```

Обязательно ли использовать скобки в выражении (25 + 3 * 8) ? Да, обязательно. Если этого не сделать, то будут выполнены следующие шаги вычислений:

```
'--- 25 + 3 * 8 = ' + 25 // --- 25 + 3 * 8 = 25
3 * 8 // 24
'--- 25 + 3 * 8 = 25' + 24 // --- 25 + 3 * 8 = 2524
'--- 25 + 3 * 8 = 2524' + ' ---' // --- 25 + 3 * 8 = 2524 ---
```

Можно «склеить» в строку и повторенный несколько раз отдельный символ, например, **var** s := 30 * '+';

Строку можно усечь или удлинить, изменяя ее длину. Это можно сделать при помощи процедуры SetLength (помните, та самая, управляющая длиной массива). Удлиняемая строка дополняется справа пробелами до нужной длины.

SetLength(s, len) – усекает или удлиняет строку *s* до длины *len*.

```
begin
  var P: string -> () := s -> Println('|' + s + '|');
  var s := 'Маша ела кашу';
  P(s); // '|' - чтобы видеть пробелы в начале и конце
  SetLength(s, s.Length - 4);
  P(s);
  SetLength(s, s.Length + 6);
  P(s)
end.
```

```
|Маша ела кашу|
|Маша ела |
|Маша ела       |
```

8.3.4. Сравнение строк

Содержимое строк, даже если их длина различна, можно сравнивать между собой. Строки сравниваются посимвольно слева направо до

первого несовпадения кодов символов, либо до окончания одной или обеих строк. Для строк определены результаты всех шести операций сравнения: <, <=, =, <>, >, >=. Большим считается тот символ, код Unicode которого больше (говорят, что символы сравниваются ***лексикографически***). Если все символы более короткой строки совпали с символами более длинной, то большей строкой считается более длинная строка.

```
##
Println('Кошка' > 'Мышка'); // False, 'К' < 'М'
Println('12345' > '1234');  // True, строка 1 длиннее
Println('ЛЕН' > 'ЛЁН'); // True, 'Ё' в Unicode идет перед 'А'
Println('лен' > 'лён'); // False, 'ё' в Unicode после 'я'
Println('Папа' <= 'мама'); // True, 'П' < 'м'
Println('Подвох' = 'Подвох') // False, первая 'о' - латинская
```

Чтобы лучше ориентироваться при сравнении строк, надо рассмотреть таблицу Unicode. Предполагаю, что не каждый читатель захочет этим заниматься, поэтому привожу некоторые сведения, полезные при сравнении.

– цифры '0'..'9' имеют десятичные коды #48..#56;
– латинские буквы 'A'..'Z' имеют десятичные коды #65..#90;
– латинские буквы 'a'..'z' имеют десятичные коды #97..#122;
– буква 'Ё' имеет десятичный код #1025;
– буквы кириллицы 'А'..'Я' (кроме Ё) имеют десятичные коды #1040..#1071;
– буквы кириллицы 'а'..'я' (кроме ё) имеют десятичные коды #1072..#1103;
– буква 'ё' имеет десятичный код #1105.

При сравнении строк возможны различные варианты и для их анализа может понадобиться делать ряд последовательных сравнений. Это может порождать не отличающиеся большой наглядностью конструкции из вложенных «if». В PascalABC.NET включены средства, позволяющие частично преодолеть это неудобство.

– CompareStr(s1, s2) – функция, выполняющее сравнение строк. Она возвращает значение типа **integer**. Если s1< s2, возвращает отрицательное значение, если s1>s2, возвращает положительное значение. Возвращает ноль, если s1=s2. Пытаться искать какую-то гарантированную закономерность в возвращаемых ненулевых значениях не следует.

- **string**.Compare(s1, s2) – статический метод, выполняющий сравнение строк. Он возвращает значение типа **integer**. Когда первая строка больше, возвращается 1, когда строки равны, возвращается 0 и когда первая строка меньше, возвращается -1. Получается своеобразная функция Sign( ). Метод удобно использовать с оператором case для организации ветвления.
- **string**.Compare(s1, s2, IgnoreCase) – статический метод, делающий то же самое и при этом игнорирующий регистр буквенных символов, если в качестве IgnoreCase указано True. Для запоминания не очень удачная мнемоника: запись string.Compare(s1, s2, True) ассоциируется с каким-то «настоящим», истинным сравнением, а тут наоборот – сравнение неточное, игнорирующее регистр.

```
##
var Эталон := 'Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C))))-Sin(D)';
var a := |
    'Sqrt(A/(2*pi*(B-Sqrt(b*b-C*C))))-Sin(d)',
    'Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C))))-Sin(D)',
    'Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C)))-Sin(D))',
    'Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C))))+Sin(D)'|;
Println(Эталон, ' - эталонная строка');
foreach var s in a do
begin
  var r := string.Compare(Эталон, s, True);
  case r of
    -1: Println(s, ' - эталон меньше');
    0: Println(s, ' - отлично!');
    1: Println(s, ' - эталон больше')
  end
end
```

```
Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C))))-Sin(D)  - эталонная строка
Sqrt(A/(2*pi*(B-Sqrt(b*b-C*C))))-Sin(d)  - отлично!
Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C))))-Sin(D)  - эталон меньше
Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C)))-Sin(D))  - эталон меньше
Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C))))+Sin(D)  - эталон больше
```

Почему в третьей строке «эталон меньше»? Внешне все совпадает с точностью до символа. Логично предположить, что минимум один из символов A, B или C на самом деле не латинский, ведь схожие с латиницей символы кириллицы имеют большие коды. Как найти место расхождения? Конечно программно!

```
##
var Эталон := 'Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C))))-Sin(D)';
var s := 'Sqrt(A/(2*Pi*(B-Sqrt(B*B-C*C))))-Sin(D)';
for var i := 1 to Эталон.Length do
  if Эталон[i] <> s[i] then
    Println(i, Эталон[i], OrdUnicode(Эталон[i]),
        OrdUnicode(s[i]))
```

```
26 C 67 1057
```

Символ номер 26 (буква С) в эталоне латинский, в сравниваемой строке – на кириллице.

8.3.5. Копирование строк

Строки копируются обыкновенным присваиванием вида s1 := s2. Но ведь они имеют ссылочный тип и при выполнении присваивания s1 := s2 в s1 копируется ссылка на s2. Любое изменение в одной из строк должно привести к изменению в другой строке, как это было с массивами, но на практике этого не происходит. Причина в том, что PascalABC.NET при модификации строк использует высокоэффективный механизм «Копирование при записи», иногда называемый на профессиональном жаргоне «коровьим» (COW – Copy-on-Write). Его суть в том, что непосредственно перед модификацией создается копия строки, с которой выполняется работа, а лишь затем ссылка обновляется. Так что присваивайте и не беспокойтесь ни о чем.

8.3.6. Выделение подстроки

Подстрока – это часть строки, полученная путем выборки некоторых ее символов, следующих подряд. Не забывайте, что символы в строке нумеруются от единицы.

- расширение s.Left(k) возвращает *k* левых символов строки *s*;
- расширение s.Right(k) возвращает *k* правых символов строки *s*.

Для получения подстроки удобно использовать срезы. Легче запоминается, короче запись.

8.3.7. Срезы строк

Со срезами вы знакомы по массивам. Срезы строк реализуются аналогично, но возвращают они подстроку, а не последовательность или

массив отдельных символов. Срез может использоваться в выражениях везде, где может использоваться строка. Нумерация символов в срезах строк ведется от единицы.

```
##
var s := 'Параграф';
s[:5].Println; // Пара
s[5:].Println; // граф
s[2:5].Println; // ара
s[3::2].Println; // рга
s[8::-2].Print // фраа
```

Имеется также расширение .Slice, позволяющее получать срезы. Нумерация символов здесь ведется **от нуля**.

- s.Slice(f, h) – возвращает срез строки *s*, начиная с позиции *f* и выбирая символы с шагом *h*;
- s.Slice(f, h, k) – возвращает срез строки *s* длины не более *k*, начиная с позиции *f* и выбирая символы с шагом *h*;

```
##
var s := 'Параграф';
s.Slice(0, 1, 4).Println; // Пара
s.Slice(4, 1).Println; // граф
```

Задача 8.4
Получить из букв слова «информатика» слова «карта», «атом», «комната» и «аромат»

```
##
var s := 'информатика';
//        12345678901
Println(s[10:] + s[5] + s[8:6:-1]); // карта
Println(s[7:9] + s[4:7:2]); // атом
Println(s[10] + s[4:7:2] + s[2] + s[7:9] + s[11]); // комната
Print(s[7] + s[5:3:-1] + s[6:9]) // аромат
```

Комментарий в третьей строчке кода дает возможность не считать каждый раз позиции символов.

8.3.8. Модификация строки

Под модификацией строки понимают ее изменение. Это может быть изменение длины строки, связанное с удалением каких-либо символов, либо вставкой дополнительных символов. Это также может быть

замена всех, либо части символов, входящих в строку, на другие. В большинстве случаев исходная строка не меняется, а возвращается новая строка.

§1. Смена регистра символов

Как и в случае с отдельными символами, в строке можно изменять регистр всех символов или их части. Используются расширения, возвращающие новую строку.

- расширение s.ToLower возвращает строку *s*, приведенную к нижнему регистру;
- расширение s.ToUpper возвращает строку *s*, приведенную к верхнему регистру.

§2. Удаление символов в начале и конце

Наиболее популярным при обработке строк является удаление пробелов перед первым непробельным символом (левые, или лидирующие пробелы) и после последнего непробельного символа (правые, или завершающие пробелы).

- s.TrimStart – метод, возвращающий исходную строку s с удаленными лидирующими пробелами;
- s.TrimEnd – метод, возвращающий исходную строку s с удаленными завершающими пробелами;
- s.Trim – метод, возвращающий исходную строку s с удаленными лидирующими и завершающими пробелами.

```
##
var P: procedure(s: string) := s -> Println('|' + s + '|');
var s := '   Маша    ела    кашу    ';
P(s); // '|' - чтобы видеть пробелы в начале и конце
P(s.TrimStart);
P(s.TrimEnd);
P(s.Trim)
```

```
|   Маша    ела    кашу    |
|Маша    ела    кашу    |
|   Маша    ела    кашу|
|Маша    ела    кашу|
```

Кроме пробела имеются и другие непечатаемые (пробельные) символы, поэтому встречаются задачи и на удаление всех таких символов. Для этого нужно указать в качестве аргумента удаляемые сим-

волы, разделяя их запятыми или использовать массив символов. Не забывайте, что в этом случае пробельные символы тоже должны быть указаны, если они могут быть нежелательными.

```
##
var s := '  ..,   +  Это нужно!  .  , .,++   ';
s.Trim(' ', '.', ',', '+').Println;
var d := |' ', '.', ',', '+'|;
s.Trim(d).Print
```

```
Это нужно!
Это нужно!
```

§3. Удаление подстрок

1. Удалить подстроку в позициях символов *[i; j]* можно при помощи срезов. Для этого нужно взять символы с от начала строки и до номера *i*, не включая его и соединить с символами, номера которых начинаются после *j* и следуют до конца строки. Такую операцию для строки с именем *s* можно записать как *s := s[:i] + s[j + 1:]*.

```
##
var s := 'У Вани машина, у Наташи - мяч';
//        12345678901234567890123456789
//        -------                  -----
s:=s[:8]+s[25:];
s.Print
```

```
У Вани – мяч
```

2. Можно воспользоваться процедурой *Delete(s, from, k)*, удаляющей из строки *s* подстроку длиной *k* символов, начиная с позиции *from*. Если удаляемых символов окажется меньше k, будут удалены символы до конца строки. Эта процедура сохранена в целях совместимости с базовым Паскалем.

```
##
var s := 'У Вани машина, у Наташи - мяч';
//        12345678901234567890123456789
//        -------                  -----
Delete(s, 8, 17);
s.Print
```

3. Расширение *s.Remove(from, k)* возвращает строку, полученную удалением из строки *s* подстроки длиной *k* символов, начиная с позиции *from*. Это расширение класса string, поэтому индексирование

ведется от нуля. Если удалить *k* символов окажется невозможно, выполнение программы завершится аварийно.

```
##
var s := 'У Вани машина, у Наташи - мяч';
//        012345678901234567890123456789
//        -------                  -----
s.Remove(7, 17).Print
```

Если расширение записать в виде *s.Remove(from)*, то удаление выполняется до конца строки.

4. Расширение *s.Remove(ss)* умеет делать еще один вид удаления – возвращать строку, полученную путем удаления из строки s всех вхождений всех подстрок, определенных параметром *ss*. В качестве этого параметра можно задать массив подстрок, либо перечислить подстроки через запятую.

```
##
var s := 'У Вани машина, у Наташи - мяч';
s.Remove('машина', 'мяч', '-').Print
```

```
У Вани , у Наташи
```

§4. Инверсия

Если в строке расположить все ее символы в обратном порядке, то полученную строку называют инвертированной относительно исходной или просто инверсией. Существуют также инверсии подстроки. PascalABC.NET предлагает несколько способов получения инверсии строки.

- s.Inverse – расширение, возвращающее инверсию строки *s*;
- Инвертирование строки *s* с помощью среза *s[::-1]*;
- ReverseString(s, from, k) – функция, возвращающая строку *s*, в которой инвертирована подстрока длиной *k*, начиная с позиции *from.*

```
##
var s := 'У всякого свой норов и обычай';
//        012345678901234567890123456789
s.Println;
ReverseString(s, 16, 5).Println;
s[16:21] := s[20:15:-1]; // реверс части строки срезами
s.Print;
```

```
У всякого свой норов и обычай
У всякого свой ворон и обычай
У всякого свой ворон и обычай
```

§5. Вставка подстроки

Средств для вставки подстроки в строку всего два и функционально они несколько отличаются. Процедура Insert изменяет исходную строку, а метод Insert возвращает измененную строку в качестве результата.

- Insert(ss, s, from) – процедура, вставляющая в строку *s* подстроку *ss*, начиная с позиции *from*. Указание несуществующей позиции приводит ко вставке подстроки перед первым (при *from* <1) или после последнего символа строки;
- s.Insert(from, ss) – метод, возвращающий строку, полученную путем вставки подстроки *ss* в исходную строку *s* с позиции *from*. Нумерация позиций, как и подобает методу класса **string**, ведется от нуля. Указание несуществующей позиции приводит к аварийному завершению программы с выдачей сообщения «Ошибка времени выполнения: Заданный аргумент находится вне диапазона допустимых значений».

```
##
var s := 'У меня есть конфета';
s.Println;
s.Insert(12, 'вкусная ').Println;
Insert('вкусная ', s, 13);
s.Print
```

```
У меня есть конфета
У меня есть вкусная конфета
У меня есть вкусная конфета
```

§6. Замена подстроки

Часто бывает нужно заменить в строке один контекст на другой. Например, поменять имя переменной в выражении. Метод s.Replace(s1, s2) возвращает строку, полученную из исходной строки *s* заменой всех вхождений подстроки *s1* на подстроку *s2*.

```
##
var s := 'Sin(2*x-1)*Sqr(Cos(x+5))+0.4*x**3';
s.Replace('x', 'УголНаклона').Print
```

```
Sin(2*УголНаклона-1)*Sqr(Cos(УголНаклона+5))+0.4*УголНаклона**3
```

Разновидность s.Replace(s1, s2, k) возвращает строку, полученную из исходной строки *s* заменой *k* первых вхождений подстроки *s1* на подстроку *s2*.

Более сложные замены можно осуществить на основе расширения .RegexReplace, использующего регулярные выражения.

8.3.9. Проверки в строке

В процессе работы со строкой может возникать потребность выполнить некоторые проверки ее содержимого. PascalABC.NET предоставляет для этой цели достаточное количество средств и мы рассмотрим лишь часть из них, которые могут встретиться в школьной практике. Результат проверки всегда имеет тип **boolean** и равен True в случае ее успешности.

Функция StringIsEmpty(s, p) проверяет, все ли символы строки *s*, начиная с позиции, заданной в **целочисленной переменной** *p*, являются пробельными. Если все, значение *p* станет равно *s.Length+1*, в противном случае оно не измениться.

Остальные проверки реализуются расширениями класса **string**:
- s.StartWith(ss) проверяет, начинается ли строка *s* с подстроки *ss*;
- s.EndWith(ss) проверяет, заканчивается ли строка *s* подстрокой *ss*;
- s.Contains(ss) проверяет, содержит ли строка s подстроку ss;
- s.InRange(s1, s2) проверяет, находится ли строка *s* между строками *s1* и *s2*, т.е. соблюдается ли условие $s1 \leqslant s \leqslant s2$.

Расширение s.IsMatch использует регулярные выражения и будет рассмотрено позднее.

На практике вместо проверок наличия вхождения подстроки в строку используют поиск мест вхождения.

8.3.10. Разбиение строки на слова

Операция разбиения строки на слова порождает массив строк, в котором каждый элемент является словом. Слово – чаще всего последовательность непробельных символов, ограниченная не менее чем

одним пробельным символом или концом строки. Безусловно, это не отвергает вариантов, когда слово должно удовлетворять более жестким требованиям, например, содержать только буквы, или только цифры, или не включать знаков препинания и т.д.

Расширение s.ToWords(cc) возвращает массив слов, полученных разбиением строки *s*; при этом в качестве разделителей слов могут использоваться один или более символов, перечисленных в *cc* через запятую, либо содержащихся в массиве символов *cc*. Параметр *cc* можно не указывать, тогда разделителями слов считаются пробельные символы.

Задача 8.5
Дана строка, содержашая слова, разделенные произвольным количеством пробелов. Вывести те из слов, у которых первая буква совпадает с последней.

Тестовая строка: « баркас табурет олово собака магазин астра и».

```
## ' баркас табурет олово  собака магазин астра  и'
   .ToWords.Where(t -> t[1] = t[^1]).Print;
```

```
табурет олово астра и
```

Расширение .ToWords формирует массив, каждый элемент которого содержит слово, полученное из заданной строки. Элементы массива проходят фильтрацию по условию *t[1] = t[^1]*. Очередное слово *t* – это строка, поэтому можно обращаться к ее отдельным символам, как к элементам массива, индексированным от единицы. Тогда *t[1]* – первая буква в слове, а *t[^1]* – последняя (вспоминаем, что символ ^ перед значением индекса обозначает отсчет с конца).

Как написать код без всех этих расширений и фильтров? Введем понятие курсора – позиции рассматриваемого символа в строке. Курсор всегда перемещается слева направо - к следующему пробелу или к непробельному символу (букве). Функция ВыводимоеСлово получает слово и оценивает, удовлетворяет ли оно условию вывода. Возврат этой функцией пустой строки (строки с длиной 0) означает, что выводить слово не нужно.

```
function ВыводимоеСлово(Строка: string; var Курсор: integer): string;
begin
  while (Курсор < Строка.Length) and (Строка[Курсор] = ' ') do
    Курсор += 1;
  var ПрежняяПозиция := Курсор;
  while (Курсор < Строка.Length) and (Строка[Курсор] <> ' ') do
    Курсор += 1;
  if (Строка[Курсор] <> ' ') and (Курсор = Строка.Length) then
    Курсор += 1;
  if Строка[ПрежняяПозиция] = Строка[Курсор - 1] then
    Result := Copy(Строка, ПрежняяПозиция, Курсор - ПрежняяПозиция)
  else
    Result := ''
end;

begin
  var s := ' баркас табурет олово  собака магазин астра    и';
  var Курсор := 1;
  while Курсор < s.Length do
  begin
    var ОчередноеСлово := ВыводимоеСлово(s, Курсор);
    if ОчередноеСлово.Length > 0 then
      ОчередноеСлово.Print
  end
end.
```

Сравним этот код с приведенным ранее на PascalABC.NET. Полстраницы против одной строки. В которой, вдобавок ко всему, еще и алгоритм виден. Такие преимущества дает знание PascalABC.NET. Хватит ли учителю времени, чтобы за один урок создать подобный код на базовом Паскале, отладить и разобрать его работу? Сомневаюсь. Поэтому в школах, где изучают старенькие турбопаскали, скорее всего учеников не учат решать подобные задачи.

8.3.11. Сцепление (слияние) строк

Слияние строк – операция, обратная разбиению строки. Несколько строк сливаются в одну общую. При этом в месте слияния может находиться символ-разделитель, подстрока из нескольких разделителей или не находиться ничего.

- Concat(s1, s2, …) – функция, возвращающая строку, которая является сцеплением (конкатенацией) строк *s1*, *s2*, … без использования разделителей. Часто заменяется операцией «+»;
- **string**.Join(ss, del) – статический метод, возвращающий строку, полученную сцеплением подстрок, находящихся в массиве *ss*. Подстрока *del* используется в качестве разделителя;
- ss.JoinToString(del) – расширение, возвращающее строку, полученную сцеплением подстрок, находящихся в массиве или последовательности *ss*. Подстрока *del* используется в качестве разделителя; по умолчанию используется символ пробела.

Традиционный путь обработки строки состоит в ее последовательном просмотре с целью выделения позиций, занимаемых очередным словом и некоторых действий над символами в этих позициях. Разбиение строки на слова при помощи расширения .ToWords, последующая обработка слов в полученном динамическом массиве и их сцепление в строку позволяет легко и быстро написать программу обработки строк. Возможно, при этом создаются не самые эффективные с точки зрения расходуемой памяти и времени выполнения программы, но это компенсируется небольшим временем их разработки и отладки.

Рассмотрим авторский вариант решения задачи из статьи Т. Богомоловой, Ирина Фалиной и В. Шухардиной «Ступенька мастерства: решаем задачи на обработку строк»., приведенной в журнале «Информатика» №8 (561). Решение приводится в базовом Паскале. Алгоритм предполагает использование метода барьерных элементов, который в данной реализации сводится к дописыванию пробела в конец исходной строки.

Задача 8.6

Слова в строке разделяются пробелами. Зашифровать текст таким образом, чтобы каждое слово текста было записано в обратном порядке.

```
var
  s, s1: string;
  t: char;
  a, i, j, b, p: integer;

begin
  write('st ==> ');
  readln(s);
  s := s + ' '; {Добавили барьерный элемент}
  i := 1; a := length(s);
  repeat
    if s[i] <> ' ' then
    begin
      s1 := ''; p := i;
      while s[i] <> ' ' do
      begin
        s1 := s1 + s[i];
        i := i + 1;
      end;
      b := length(s1);
      for j := 1 to b div 2 do
      begin
        t := s1[j]; s1[j] := s1[b - j + 1];
        s1[b - j + 1] := t
      end;
      delete(s, p, b);
      insert(s1, s, p);
    end
    else i := i + 1
  until i >= length(s);
  writeln(s)
end.
```

31 строка текста. Конечно, вы уже просмотрели код и поняли алгоритм. Нет? Хорошо, посмотрите еще пару минут. Снова не поняли? Ну что же, «тяжело в ученье – будет тяжело и в работе». Давайте посмотрим, как это может выглядеть в PascalABC.NET.

```
## ReadlnString('st ==>').ToWords.Select(w -> w.Inverse)
      .JoinToString.Println
```

И это все? В одну строку? Да, все.

Разберем код. *ReadlnString('st ==>')* считывает с клавиатуры строку, выдав приглашение ко вводу, взятое из приведенной в журнале программы и возвращает ее в качестве результата. Дадим получен-

ной строке условное имя *s*. Далее выполняется метод *s.ToWords*, разбивающий строку по пробельным символам и возвращающий массив полученных слов. Назовем этот массив *a*. Проецирование *a.Select(w -> w.Inverse)* применяет к каждому элементу массива *a* расширение Inverse, в результате чего получается инверсия находящихся в массиве слов. Результат проецирования - последовательность, которую обозначим *p*. Расширение *p.JoinToString* объединяет элементы в строку и, используя в качестве разделителя принятый по умолчанию пробел. Получаем некоторую строку *s*, которую *s.Println* и выводит на монитор. Описать все это оказалось значительно дольше, чем написать программу.

8.3.12. Поиск в строке

Под поиском понимается нахождение в строке номера позиции отдельного символа или подстроки. В языке Паскаль символы нумеруются от единицы и при поиске именно такая нумерация используется при вызове функций. В то же время, методы строк рассматривают строку как динамический массив символов и поэтому в качестве номера позиции выступает индекс, который отсчитывается от нуля.

§1. Поиск в прямом направлении

Функция Pos(ss, s) возвращает номер позиции первого вхождения подстроки *ss* в строку *s*. Если подстрока не найдена, возвращается ноль. Функция Pos(ss, s, from) делает то же самое, но поиск начинается не от начала строки, а с позиции *from*. Это дань базовому Паскалю.

Метод s.IndexOf(ss) возвращает индекс первого вхождения подстроки *ss* в строку *s*. Если подстрока не найдена, возвращается -1. Метод s.IndexOf(ss, from) делает то же самое, начиная поиск с символа, имеющего индекс *from*. Метод s.IndexOf(ss, from, k) распространяет поиск только на первые *k* символов, начиная с позиции *from*.

Метод s.IndexOfAny(cc) возвращает индекс первого вхождения в строку *s* любого символа из массива *cc*. Если ни один из символов не найден, возвращает -1. Метод может быть полезен, например, при поиске знаков препинания или начала числа в строке.

```
##
var s := 'Самое синее в мире Черное море мое';
//        12345678901234567890123456789012345
Pos('мо', s).Print; // первое вхождение
Pos('мо', s, Pos('мо', s) + 1).Println; // второе вхождение
s.IndexOf('мо').Print;  // первое вхождение
s.IndexOf('мо', s.IndexOf('мо') + 1).Println; // второе вхождение
s.IndexOfAny(|'и', 'е'|).Print
```

```
3 27
2 26
4
```

§2. Поиск в обратном направлении

Функция LastPos(ss, s) возвращает номер позиции последнего вхождения подстроки *ss* в строку *s*. Если подстрока не найдена, возвращается ноль. Функция LastPos(ss, s, from) делает то же самое, но поиск начинается с позиции *from* и ведется в обратном направлении. Фактически, значение *from* ограничивает поиск первыми *from* символами. Как и Pos, эта функция сохранена для обратной совместимости с базовым Паскалем.

Метод s.LastIndexOf(ss) возвращает индекс последнего вхождения подстроки *ss* в строку *s*. Если подстрока не найдена, возвращается -1. Метод s.LastIndexOf(ss, from) делает то же самое, начиная поиск с символа, имеющего индекс *from*. Метод s.LastIndexOf(ss, from, k) распространяет поиск только на последние *k* из *from* символов.

Метод s.IndexOfAny(cc) возвращает индекс последнего вхождения в строку *s* любого символа из массива *cc*. Если ни один из символов не найден, возвращает -1.

```
##
var s := 'Самое синее в мире Черное море мое';
//        12345678901234567890123456789012345
LastPos('мо', s).Print; // последнее вхождение
LastPos('мо', s, LastPos('мо', s) - 1).Println; // предпоследнее
s.LastIndexOf('мо').Print;  // последнее вхождение
s.LastIndexOf('мо', s.LastIndexOf('мо') - 1).Println;
s.LastIndexOfAny(|'и', 'е'|).Print
```

```
32 27
31 26
33
```

§3. Быстрый поиск всех вхождений подстроки

Метод s.IndicesOf(ss) возвращает последовательность индексов всех вхождений подстроки *ss* в строку *s*.

```
##
var s := 'Самое синее в мире Черное море мое';
//        12345678901234567890123456789012345
s.IndicesOf('мо').Print
```

```
2 26 31
```

По умолчанию подстроки не могут перекрываться. Если требуется выполнить поиск с учетом перекрытий подстрок, метод вызывается в виде s.IndicesOf(ss, True).

```
##
var s := 'aabaabaaaabaabaaab';
//        012345678901234567
s.IndicesOf('aabaa', False).Println;
s.IndicesOf('aabaa', True).Print
```

```
0 8
0 3 8 11
```

Этот метод основан на алгоритме Кнута – Морриса – Пратта (КМП-алгоритм) и имеет высокую скорость работы для строк очень большой длины. Для поиска всех вхождений подстроки без перекрытия в строках относительно небольшого размера эффективнее может оказаться поиск на основе ***регулярного выражения***.

В заключение рассмотрим традиционный поиск позиций всех вхождений слова в тексте при помощи функции Pos. Обратите внимание на использование переменной i в левой и правой частях оператора присваивания с вызовом Pos. Еще одна особенность программы – «накопление» результатов в символьной строке r.

```
begin
  var s :=
      'Лениво дышит полдень мглистый,' + NewLine +
      'Лениво катится река.' + NewLine +
      'И в тверди пламенной и чистой' + NewLine +
      'Лениво тают облака.' + NewLine +
      '(Ф. И. Тютчев)';
  Writeln(s, NewLine, '-' * 45);
  var (i, k, r) := (1, 0, 'Слово "Лениво" найдено в позициях ');
  repeat
    i := Pos('Лениво', s, i);
    if i = 0 then break;
    r += i + ' ';
    i += 1;
    k += 1;
  until False;
  if k = 0 then Println('Слово "Лениво" в строке не обнаружено')
  else
  begin
    r.Println;
    Println('Количество вхождений:', k)
  end
end.
```

Приведенная программа находит позиции всех трех вхождений подстроки «Лениво».

```
Лениво дышит полдень мглистый,
Лениво катится река.
И в тверди пламенной и чистой
Лениво тают облака.
(Ф. И. Тютчев)
---------------------------------------------
Слово "Лениво" найдено в позициях 1 33 86
Количество вхождений: 3
```

Почему позиция 33, когда второе вхождение начинается вроде бы с позиции 31? Символ NewLine – две позиции (код #13 #10).

Для сравнения рассмотрите реализацию этой же задачи на основе КМП-поиска.

```
##
var s :=
    'Лениво дышит полдень мглистый,' + NewLine +
    'Лениво катится река.' + NewLine +
    'И в тверди пламенной и чистой' + NewLine +
    'Лениво тают облака.' + NewLine +
    '(Ф. И. Тютчев)';
Writeln(s, NewLine, '-' * 45);
var r := s.IndicesOf('Лениво');
if r.Count = 0 then
  Println('Слово "Лениво" в строке не обнаружено')
else
begin
  Print('Слово "Лениво" найдено в позициях');
  r.Select(t -> t + 1).Println;
  Println('Количество вхождений:', r.Count)
end
```

8.4. Регулярные выражения

Некоторые люди во время решения некой проблемы думают: «Почему бы мне не использовать регулярные выражения?». После этого у них уже две проблемы...

Джейми Завински,
американский программист

Термин ***регулярное выражение*** (далее по тексту РВ) отражает свойство математических выражений, называемое ***регулярностью***. Желающие разобраться в том, что такое регулярность, могут обратиться к специальной литературе Часто можно также встретить английская аббревиатуру RegExp (Regular Expressions). РВ посвящены целые книги, но неверно думать, что РВ способны решать любые проблемы обработки текстовых строк. Их удел – поиск и замена подстрок.

Платформа Microsoft .NET Framework обеспечивает поддержку диалекта в стиле языка Perl, поэтому при изучении РВ можно пользоваться литературой по этому языку.

Внешне РВ представляет собой некоторую строку-шаблон, с которой сопоставляется обрабатываемый текст. Результатом сопоставления будут подстроки, удовлетворяющие РВ. На основании содержимого этих подстрок делается вывод о наличии или отсутствии или чего-либо в тексте, осуществляется выборка подстрок или их замена.

РВ состоит из ***метасимволов*** – специальных символьных комбинаций и ***литералов*** – всех прочих символов. Метасимволы объясняют, как интерпретировать литералы в РВ для выполнения поиска в заданной строке.

При работе с РВ учитывается набор значений параметров, определяющих специфику поиска, например, нужно ли учитывать регистр символов, останавливаться после первого успешного поиска или продолжать его и т.д. Эти параметры могут задаваться как в самом РВ, так и в операторе языка, использующего РВ.

Далее, когда речь будет идти о РВ, на основе которого производится поиск, наряду с термином «регулярное выражение» будет употребляться термин «шаблон».

8.4.1. Некоторые метасимволы

Обозначает	Метасимвол	Шаблон	Подходит для
Один любой символ	. (точка)	кл.н	клен, клин, клон
Цифра 0..9	\d	\d\d\d	125, 999, 103, 217
Не цифра	\D	\D\D	A8, C#, да
Буква	\w	\w\w\w	Кит, сом, _zT, ааа
Не буква	\W	\W\W	3a, R9, C#, !!
Пробельный символ	\s	\w\s1	a 1, Z 1, Б 1
Непробельный символ	\S	\S\s\S	A,1, !=!, #5?, ю+я
Символьный класс	[ ]	кл[аие]н	клан, клен клин
Инвертированный символьный класс	[^]	кл[^аи]н	клен, клон
Диапазон символов	[-]	[a-f]	a, b, c, d, e, f
Альтернатива «или»	(| | |)	ко[т|шка]	кот, кошка

8.4.2. Некоторые квантификаторы

Квантификатор определяет количество повторов метасимвола, предшествующего квантификатору.

Обозначает	Квантификатор	Шаблон	Подходит для
0 или 1 раз	?	сон.?	сон, соня, сон!
Не менее 1 раза	+	Бар+	Барс, Барон, Баркас
0 и более раз	*	бар*	бар, барон, барс
Точно n раз	{n}	\d{3}	120, 399, 618
Не более n раз	{,n}	\w{,2}	!
Не менее n раз	{n,}	.{3,}	08:58, //http:, ###
От m до n раз	{m,n}	\d{1,2}	6, 77, 38, 04, 5

8.4.3. Экранирование

Чтобы в шаблоне указать символ, который используется в РВ для служебных целей, перед ним записывают символ экранирования – обратную косую черту «\». К служебным относятся символы «+», «\», «*», «?», «|», «{», «[», «(», «)», «^», «$», «.», «#». Также, нужно экранировать обычную косую черту «/».

Пример шаблона с экранирующими символами: \(\d{3}\) – трехзначное число в круглых скобках.

8.4.4. Директивы нулевой длины

Это не реальные символы, а своего рода условия, выполнение которых проверяется.

/b – граница слова. Понимается как место, где соседствуют символы /w и /W;

^ - начало строки;

$ - конец строки.

8.4.5. Примеры шаблонов

1. Любому целому числу без знака удовлетворяет шаблон \d+

2. Любому целому числу, которое может иметь знак, удовлетворяет шаблон [+-]?\d+

3. Дата в формате дд/мм/гггг может быть выделена при помощи шаблона \d\d\/\d\d\/\d{4}

4. Номер автомобиля вида <буква><3 цифры><две буквы> можно найти по шаблону \w\d{3}\w\w

5. Номер телефона в России вида +7(ddd)ddd-dd-dd отыщет шаблон вида +7\(\d{3}\)\d{3}-\d\d-\d\d

6. Слово ищется посредством шаблона \b\S+\b

7. Слово длиной не менее двух букв, которое начинается и заканчивается одной и той же буквой, может быть найдено по шаблону \b(\S)\S*\1\b

8. Слова, содержащие только русские буквы, удовлетворяют шаблону \b[А-Яа-яЁе]+\b

Более подробно о регулярных выражениях можно прочитать в книге «PascalABC.NET: Введение в современное программирование».

8.4.6. Наличие подстроки в строке

Расширение s.IsMatch(reg, opt) проверяет, удовлетворяет ли строка *s* регулярному выражению *reg*. С помощью *opt* можно задавать дополнительные опции.

В примере рассматривается строка «Роза увяла от мороза», в которой ищется подстрока «роза». Если учитывать регистр, то такая подстрока одна, без учета регистра их две.

```
##
var s := 'Роза увяла от мороза';
s.IsMatch('роза').Println; // True
s.IsMatch('роза\s').Println; // False
s.IsMatch('Роза\s').Println; // True
s.IsMatch('роза\s', RegexOptions.IgnoreCase).Println; // True
```

В последней строке опция RegexOptions.IgnoreCase означает требование игнорировать регистр букв.

8.4.7. Результаты поиска в элементах Match

Расширение s.Matches(reg, opt) возвращает последовательность элементов типа Match из строки *s*, соответствующих регулярному выражению *reg*. С помощью *opt* можно задавать дополнительные опции.

Если m – объект типа Match, три его свойства могут оказаться особенно полезными: m.Value – найденная подстрока, m.Length – ее длина, m.Index – индекс ее первого элемента.

8.4.8. Замена всех подстрок в строке

Расширение s.RegexReplace(reg, ss, opt) возвращает строку, полученную заменами в строке *s* всех найденных вхождений подстроки, удовлетворяющей регулярному выражению, подстрокой *ss*. С помощью *opt* можно задавать дополнительные опции.

```
##
var s := 'Мама мыла раму, Маша ела кашу';
s.RegexReplace('М','Д').Println;
s:='23*x-Sin(x)';
s.RegexReplace('x','(x+1.5)').Println;
s.RegexReplace('\-Sin\(x\)','').Print
```

```
Дама мыла раму, Даша ела кашу
23*(x+1.5)-Sin((x+1.5))
23*x
```

В последней замене, показывающей удаление подстроки, обратите внимание на экранирование знака минус и круглых скобок. Начинающим полезно вначале убедиться при помощи IsMatch в правильности поиска, а затем делать замену.

Вторая форма расширения s.RegexReplace(reg, Match -> string, opt) отличается тем, что найденные подстроки заменяются их преобразованием, заданным лямбда-выражением.

```
##
var s := 'Мама мыла раму';
s.RegexReplace('.a', m -> UpperCase(m.Value)).Println;
s.RegexReplace('.a', m -> m.Index.ToString).Println;
s := 'А роза упала на лапу Азора';
s.RegexReplace('\w+', m -> m.Length.ToString).Println;
s := '       тестовая  строчка   ';
s.RegexReplace('\s+', m -> m.Length.ToString).Print
```

```
МАМА мыЛА РАму
02 мы7 10му
1 4 5 2 4 5
7тестовая2строчка3
```

8.4.9. Поиск первого вхождения подстроки

Расширение s.MatchValue(reg, opt) возвращает первую из подстрок в строке *s,* соответствующую регулярному выражению *reg*. С помощью *opt* можно задавать дополнительные опции. В отличии от расширения .IsMatch возвращает не логическое значение, показывающее успешность поиска, а подстроку – значение поля Value первого из элементов типа Match

```
##
var s := 'Роза увяла от мороза';
s.MatchValue('роза').Println;
s.MatchValue('роза\s').Println;
s.MatchValue('Роза\s').Println;
s.MatchValue('роза\s', RegexOptions.IgnoreCase).Println;
s.MatchValue('(?i)роза').Print // тоже без учета регистра
```

```
роза

Роза
Роза
Роза
```

8.4.10. Поиск всех вхождений подстроки

Расширение s.MatchValues(reg, opt) возвращает последовательность подстрок в строке *s*, соответствующую регулярному выражению *reg*. С помощью *opt* можно задавать дополнительные опции. Практически, в качестве результата возвращается последовательность значений поля Value из всех найденных элементов Matches.

```
##
var s := 'Роза увяла от мороза';
s.MatchValues('роза').Println;
s.MatchValues('роза\s').Println;
s.MatchValues('Роза\s').Println;
s.MatchValues('роза\s', RegexOptions.IgnoreCase).Println;
s.MatchValues('(?i)роза').Print
```

```
роза

Роза
Роза
Роза роза
```

В последнем обращении к MatchValues было найдено две подстроки. Первая получена как слово «Роза» с игнорированием регистра, вторая – как подстрока слова «мо**роза**».

8.5. Извлечение данных из строк

Выше рассматривалось разбиение строки на подстроки, в частности, на слова. При этом каждая подстрока получала такой же тип, как и исходная строка – **string**. Рассмотрим извлечение подстрок с преобразованием их типа.

8.5.1. Преобразование числа к строке

Числа преобразуют к строковому представлению для формирования строк вывода, для анализа цифр в числе. Это могут быть задачи нахождения числа цифр, поиска одинаковых цифр, получения перестановок цифр, для передачи данных в виде символьной строки и другие задачи. Метод n.ToString позволяет преобразовать к строке число любого типа.

Задача 8.7
Сколько раз встретится цифра 3 в записи целых чисел, принадлежащих отрезку [-4567; 24674] ?

```
## (-4567..24674).Select(d -> d.ToString
     .Count(c -> c = '3')).Sum.Print
```

```
12855
```

Все решение – одна строка. В приведенной записи строк две, но лишь потому, что на одной строке листа код не поместился.

8.5.2. Преобразование строки к числу

Преобразовать число к строке несложно, а с обратным преобразованием дела обстоят совсем не так. Строка может целиком представлять одно число, а может содержать в подстроках представления нескольких чисел и даже не всегда одного типа, отделенных друг от друга каким-либо разделителем или группой разделителей. Если все подстроки одного типа, то строка может быть преобразована к массиву чисел. Наконец, строка может содержать некорректные представления одного и более чисел.

§1. Преобразование строки к целому числу

Мы работали с целыми числами типов **integer** и **int64**. Но существуют также типы **shortint**, **byte**, **smallint**, **word** и это еще не все! Строка может быть преобразована к любому из этих типов, поэтому соответствующих подпрограмм и методов достаточно много.

Статический метод T.TryParse(s, n) пытается преобразовать строку *s* к целому числу *n* типа *T*. Если преобразование успешно, значение помещается в переменную *n* и возвращается **True**. В противном случае значение *n* не изменяется и возвращается **False**.

Задача 8.8
Организовать ввод целого числа с приглашением ко вводу и выдачей сообщения в случае, если при вводе были допущены ошибки.

```
##
var n: integer;
while True do
begin
  var s := ReadlnString('Введите целое число:');
  if integer.TryParse(s, n) then
    break
  else
    Writeln('Неверный ввод, повторите')
end;
Print('Число введено корректно')
```

```
Введите целое число: 345345y345
Неверный ввод, повторите
Введите целое число: 3245345
Число введено корректно
```

При встраивании в цепочку расширением TryParse пользоваться неудобно, поскольку требуется функция, возвращающая преобразованное к числовому типу значение, а не признак успешного преобразования. В подобных случаях для строки *s* можно использовать расширения s.ToInteger, s.ToInt64 и им подобные. Конечно, здесь нужна уверенность, что строка *s* является корректным изображением целого числа соответствующего типа, чтобы при выполнении программы не получить ее аварийного завершения. Например, предварительно выделить подстроку с помощью регулярного выражения.

А вот еще одна однострочная программа. Нахождение суммы всех цифр, встретившихся в строке.

```
## ReadString.MatchValues('\d')
    .Select(t -> t.ToInteger).Sum.Print
```

В условиях недостатка времени бывает удобно писать подобный код. Пусть он алгоритмически некрасив, зато пишется за минуту и отлаживать не нужно. Рассмотрите, как этот код работает.

§2. Преобразование строки к вещественному числу

Статический метод **real**.TryParse(s, n) пытается преобразовать строку *s* к целому числу *n* типа **real**. Если преобразование успешно, значение помещается в переменную *n* и возвращается **True**. В противном случае значение *n* не изменяется и возвращается **False**.

Расширение s.ToReal возвращает содержимое сроки *s*, преобразованное к значению типа **real**. В случае ошибки выполнение программы будет аварийно завершено.

§3. Преобразование строки к массиву чисел

Расширения s.ToIntegers и s.ToReals возвращают динамический массив чисел целого или вещественного типа, полученный на основе содержимого строки *s*. Элементы в строке должны разделяться не менее чем одним пробельным символом. В случае ошибки выполнение программы завершается аварийно.

Задача 8.9
Дана строка в которой через один или более пробелов записаны натуральные целые числа, не превышающие 10 млн. Какой процент от их общего количества составляют четные числа?

```
##
var s := ' 840005   1261  401 5060637 774  47216 5667899 8  ';
var a := s.ToIntegers; // получаем массив чисел
Write(a.Count(t -> t.IsEven) / a.Length * 100:0:1, '%')
```

```
37.5%
```

8.5.3. Преобразование строки в массив

Строку можно преобразовать в массив символов. Метод s.ToCharArray возвращает динамический массив типа **char**, полученный путем посимвольного разбиения исходной строки *s*. Обратное преобразование из массива или последовательности символов в строку выполняется при помощи расширения .JoinToString.

Пусть требуется выбрать из заданной строки все символы по одному разу, отсортировать их по возрастанию и заменить полученной строкой исходную.

```
##
var s := 'наша тестовая строка';
s := s.ToCharArray.Distinct.Sorted.JoinToString;
s.Print
```

```
авекнорстшя
```

8.6. Форматирование данных для вывода

Для оформления выводимых данных в языке Паскаль традиционно используется процедуры Write и Writeln, в которых для выводимого элемента данных может указываться минимальная ширина поля вывода w, а для вещественных значений – еще и количество цифр в дробной части t. Разделителем служит символ «двоеточие» (:). Если параметр w не указан или имеет недостаточную для отображения выводимого значения величину, значение выводится со своей фактической длиной. Если для вещественного числа параметр t не указан, в дробной части выводится фактическое количество цифр, но при этом общее количество цифр не может превышать 15. Если в дробной части больше цифр, чем указано в t, выводится t цифр и при этом делается округление. При указании значения t, требующего в выводимом числе отобразить более 15 цифр, все цифры после пятнадцатой будут представлены нулями.

```
##
var (a, b) := (-23.4723423290834, 16554);
Writeln(a, ' ', b, ' ', a * b);
Writeln(a, ' ', b, ' ', a * b:0:5);
Writeln(a, ' ', b, ' ', a * b:20);
Writeln(a, ' ', b, ' ', a * b:25:5);
Writeln(a, ' ', b, ' ', a * b:0:20);
Writeln('Тестовая строка', ' ', '*');
Writeln('Тестовая строка':10, ' ', '*');
Writeln('Тестовая строка':30, ' ', '*');
```

Обратите внимание на использование нуля при задании ширины поля вывода w. Это типичный прием вывода вещественных значений в Паскале, когда неизвестно количество цифр в целой части числа.

```
-23.4723423290834 16554 -388561.154915647
-23.4723423290834 16554 -388561.15492
-23.4723423290834 16554    -388561.154915647
-23.4723423290834 16554             -388561.15492
-23.4723423290834 16554 -388561.15491564700000000000
Тестовая строка *
Тестовая строка *
              Тестовая строка *
```

В случает, когда значение вещественного числа слишком мало или велико, будет использован формат вывода с плавающей точкой вида x.x…xE∓nn (x.x…x·$10^{\mp nn}$).

Выводимые числовые значения прижимаются **к правой границе** отведенного для вывода поля шириной w за счет добавления слева нужного количества пробелов. Это общепринято. Точно так же, принято прижимать строки **к левой границе** поля, поэтому они дополняются пробелами справа. Но при форматном выводе строк Паскаль добавляет пробелы **слева** и об этом надо помнить.

Существуют и другие способы вывода данных. Их общая идея состоит в том, чтобы формировать строки такими, какими они должны отображаться и затем выводить без изменения.

8.6.1. Выравнивание строки пробелами

Чтобы строка s длиной L имела нужную ширину w, можно дополнить ее w-L пробелами слева или справа. Например, вот так:

```
s := ' ' * (w - s.Length) + s;
s := s + ' ' * (w - s.Length);
```

Писать подобным образом для каждого выводимого элемента неудобно. Поэтому в классе **string** введены два метода:

- s.PadRight(w) возвращает строку *s*, дополненную пробелами **справа** до длины *w*;
- s.PadLeft(w) возвращает строку *s*, дополненную пробелами **слева** до длины *w*.

8.6.2. Составное форматирование

Выравнивание строки пробелами решает проблему с выводом только символьных данных. Для произвольного типа данных есть общее решение – использование ***составного форматирования*** (composite format в терминологии Microsoft .NET Framework).

Составное форматирование использует в качестве исходных данных строку составного формата (шаблон, задающий правила преобразования данных) и список объектов, подлежащих форматированию. Этот список следует за строкой составного формата и отделяется от

нее запятой. Элементы в списке нумеруются от нуля и разделяются запятыми.

Строка составного формата может содержать произвольную комбинацию некоторого количества неизменяемого текста, представленного литералами, и элементов форматирования, каждый из которых заключается в фигурные скобки.

Элемент форматирования является ***местозаполнителем*** (поле, в котором впоследствии будет размещено значение) и в простейшем случае содержит только отсчитываемый от нуля индекс – порядковый номер элемента, подлежащего форматированию, в списке.

```
##
Format('Длина окружности диаметра {0} равна {1}',
    2.16, 2.16 * Pi).Print
```

```
Длина окружности диаметра 2.16 равна 6.78584013175395
```

В общем случае элемент форматирования имеет вид

{*индекс*, *выравнивание* : *формат*},

где *выравнивание* – целое значение со знаком, указывающее общую ширину отформатированного поля, *формат* – строка формата для элемента. Если выравнивание имеет отрицательное значение, результат прижимается влево путем добавления справа пробелов до нужной ширины.

В Microsoft .NET Framework определен обширный набор форматов и здесь будет рассмотрена только небольшая их часть. В частности, не рассматриваются настраиваемые числовые форматы.

§1. Целочисленные данные

Используется формат d или D (decimal), за которым может быть указано желаемое количество цифр.

```
## Format('*{0,20:d}*{0,-15:D6}*{0:d1}*', -152).Print
```

```
*               -152*-000152        *-152*
```

Здесь трижды указан индекс 0, что позволило три раза использовать одно и то же значение -152 из списка. Числовое значение после d (D)

определяет минимальное желаемое количество выводимых цифр, значение выравнивания перед двоеточием – общее количество позиций. Обратите внимание на появление незначащих нулей.

§2. Вещественные данные с фиксированной точкой

Формат данных f или F определяет преобразование для отображения с фиксированной точкой (fixed), т.е. в виде целой и дробной части, разделенных точкой. По умолчанию в дробной части будут указаны две цифры с округлением значения.

```
      ## Format('*{0,20:f}*{0,-15:f6}*{0:f1}*', -152.327).Print
*             -152.33*-152.327000    *-152.3*
```

§3. Вещественные данные с плавающей точкой

Формат данных e или E определяет преобразование для отображения с плавающей точкой (exponential) – в виде одной цифры в целой части, нужного количества цифр в дробной части и показателя степени числа 10, отделенного буквой E. Этот формат обычно используется для величин, имеющих очень малые или очень большие значения. По умолчанию число знаков в дробной части равно шести.

```
      ## Format('*{0,20:e}*{0,-15:E6}*{0:e1}*', -152.327e-31).Print
*      -1.523270e-029*-1.523270E-029 *-1.5e-029*
```

§4. Числовые данные в общем формате

Формат данных g или G определяет преобразование для данных любого числового типа. Наиболее подходящий формат (d, f, e) компилятор выбирает самостоятельно. Для отображения значений **real** с максимальной точностью используйте формат g17.

```
      ##
      Format('*{0,20:g}*{0,-15:G6}*{0:g1}*', -152).Println;
      Format('*{0,20:g}*{0,-15:g6}*{0:g1}*', -152.327).Println;
      Format('*{0,20:g}*{0,-15:G6}*{0:g1}*', -152.327e-31).Print
*                -152*-152            *-2e+02*
*             -152.327*-152.327       *-2e+02*
*        -1.52327e-29*-1.52327E-29    *-2e-29*
```

§5. Формат для шестнадцатеричного представления

Формат x или X служит для преобразования целочисленных данных к их шестнадцатеричному представлению. По умолчанию отображается минимально необходимое количество цифр.

```
Format('*{0,20:x}*{0,-15:X6}*{0:x1}*', 1023).Println;
Format('*{0,20:x}*{0,-15:X6}*{0:x1}*', -1023).Print
```

```
*                3ff*0003FF         *3ff*
*            fffffc01*FFFFFC01      *fffffc01*
```

§6. Функция и статический метод Format

В приведенных выше примерах форматирование строк осуществлялось с использованием функции Format. Также можно использовать статическую функцию **string**.Format(cf, params), где *cf* – шаблон форматирования, *params* – список значений, подлежащих форматированию.

§7. Процедура форматного вывода WriteFormat

По аналогии с процедурами Write и Writeln, позволяющими при выводе указывать ширину поля, в PascalABC.NET включены процедуры WriteFormat(cf, params) и WritelnFormat(cf, params), позволяющие осуществлять вывод с использованием составного форматирования. Список параметров *params* форматируется при помощи строки составного формата *cf*, а затем выводится. Процедура WritelnFormat затем осуществляет переход к новой строке вывода.

8.6.3. Интерполированные строки

Интерполированная строка – это объединение строки составного формата и списка объектов, подлежащих преобразованию в строковое представление. Она представляет собой литерал, содержащий интерполированные выражения, перед которым записан символ $. Интерполированное выражение очень похоже на элемент форматирования строки составного формата: оно так же заключается в фигурные скобки и содержит описание формата данных, только вместо индекса подлежащего форматированию выражения указывается само это выражение. Интерполированные строки обычно короче и

нагляднее строк составного форматирования. Впрочем, сравните сами:

```
Format('Длина окружности диаметра {0} равна {1}', 2.16, 2.16 * Pi)
$'Длина окружности диаметра {2.16} равна {2.16 * Pi}'

Format('*{0,20:d}*{0,-15:D6}*{0:d1}*', -152)
$'*{-152,20:d}*{-152,-15:D6}*{-152:d1}*'

Format('*{0,20:f}*{0,-15:f6}*{0:f1}*', -152.327)
$'*{-152.327,20:f}*{-152.327,-15:f6}*{-152.327:f1}*'

Format('Сумма квадратов катетов a={0} и b={1} равна {2}',a, b, a * a + b * b)
$'Сумма квадратов катетов a={a} и b={b} равна {a * a + b * b}'
```

Интерполированная строка – это строка типа **string** и с ней можно работать обычным образом.

```
##
var (a, b) := (5.2, 4.17);
var s1 := $'Сумма квадратов катетов a={a} и b={b} равна {a*a+b*b}';
var s2 := $'{a*a} + {b*b} = {a*a+b*b}';
s2.Println;
s1.Print
```

```
27.04 + 17.3889 = 44.4289
Сумма квадратов катетов a=5.2 и b=4.17 равна 44.4289
```

8.7. Примеры решения задач

Задача 8.10

В произвольной строке, вводимой с клавиатуры, удалить пробельные символы, заменив каждую их группу ровно одним пробелом. Удалить также все пробельные символы, находящиеся до первого и после последнего непробельного символа.

Самый простой вариант – разбить строку на слова посредством расширения .ToWords, а затем элементы полученного массива объединить в строку с помощью расширения .JoinToString, указав в качестве разделителя пробел.

```
## ReadString.ToWords.JoinToString(' ').Print
```

```
   Это наша    тестовая    строка!
Это наша тестовая строка!
```

Задача 8.11
Из букв слова «апельсин» получить слово «спаниель».

Буквы нужно пронумеровать, чтобы не ошибаться в индексах.

```
апельсин
12345678
```

Нам нужно выбрать буквы с индексами 6, 2 - 1, 8 - 7, 3 – 5. Удобно воспользоваться срезами.

```
##
var s := 'апельсин';
var t := s[6] + s[2::-1] + s[8:6:-1] + s[3:6];
t.Print
```

Задача 8.12
Подсчитать в заданной строке количество гласных букв русского алфавита. Учитывать букву «ё».

Тут проще всего воспользоваться регулярным выражением, позволяющим находить одну гласную букву: [аеёиоуыэюя], получить последовательность этих букв и найти ее длину.

```
##
var s := 'Теперь ЕГЭ по информатике будет проводиться на ПК';
s.MatchValues('[аеёиоуыэюя]', RegexOptions.IgnoreCase).Count.Print
```

```
17
```

Обратите внимание на параметр RegexOptions.IgnoreCase – он позволяет не вписывать в регулярное выражение символы верхнего регистра. Если Вы его не помните, пишите выражение [аеёиоуыэюяАЕЁИОУЫЭЮЯ]. И конечно же, все эти символы можно записывать в произвольном порядке.

Задача 8.13
Строка, введенная с клавиатуры, содержит слова, разделенные одним или более пробельными символами. Слова могут содержать любые непробельные символы. Выведите в столбик те из слов, которые содержат по крайней мере два одинаковых соседних символа.

Здесь можно предварительно превратить строку в массив слов, а затем в каждом слове проверить наличие двух одинаковых соседних символов. Еще одно решение – выделять слова с одинаковыми парными символами, пользуясь регулярным выражением, которое в этом случае будет довольно непростым для начинающих.

Рассмотрим решение с массивом.

```
function Годится(Self: string): boolean; extensionmethod;
begin
  Result := False;
  for var i := 2 to Self.Length do
    if Self[i] = Self[I - 1] then
    begin
      Result := True
      Exit
    end
end;

begin
  var s := ReadString;
  foreach var Слово in s.ToWords do
    if Слово.Годится then
      Слово.Println
end.
```

```
У труженика мошенник выманил одну "н", а киллер у дилера "л" отнял
мошенник
киллер
```

Регулярные выражения все существенно облегчают. Если, конечно, вы их правильно пишете. Занятно, но это – полный код программы.

```
##
var s := ReadString;
s.MatchValues('\b\S*(\S)\1\S*\b').PrintLines
```

Рассмотрим шаблон.

- \b означает, что вначале ищется граница слова;
- \S* выбирает любое количество непробельных символов, пока не найдется подстрока, подходящая под шаблон (\S)\1;
- (\S)\1 означает непробельный символ с последующим его же повторением;
- Далее до границы слова выбираются непробельные символы

MatchValues формирует последовательность слов, удовлетворяющих шаблону, так что остается только ее вывести.

Есть два пути решения задач со строками: либо вы не понимаете регулярных выражений и пишете долго, либо все же разбираетесь с ними и быстро-быстро решаете задачи, связанные с поиском и заменой.

Задача 8.14
Проверить, является ли введенная с клавиатуры строка перевертышем. Перевертыш – фраза, которая после удаления всех небуквенных символов одинаково читается слева направо и справа налево.

Применение регулярного выражения может существенно упростить решение.

```
##
var s := ReadString.RegexReplace('[^\w]', '').ToLower;
if s = s.Inverse then
  Print('Перевертыш')
else
  Print('Не перевертыш')
```

```
А роза упала на лапу Азора
Перевертыш
```

Читаем строку с клавиатуры и убираем из нее все, что не является буквами. Далее сравниваем строку, приведенную к нижнему регистру, с ее инверсией, также приведенной к нижнему регистру.

Задача 8.15
Определим слово, как набор непробельных символов. В заданной строке каждое слово записать в обратном порядке следования его символов. Все пробельные символы должны остаться на своих местах.

Особенностью задачи является необходимость сохранения пробельных символов, поэтому разбить строку на слова с помощью .ToWords – не лучшее решение. Но можно получить на месте каждого слова элементы типа Match (см. подраздел 8.4.7), которые позволяют опре-

делить и само слово, и индекс его первого элемента в строке. Снова нас выручают регулярные выражения.

```
##
var s := '  Я   обожаю регулярные  выражения!';
foreach var m in s.Matches('\b\w+\b') do
begin
  var (i, w) := (m.Index + 1, m.Value);
  s[i:i + w.Length] := w.Inverse
end;
s.Print
```

```
  Я   юажобо еынрялугер  яинежарыв!
```

Для чего были добавлены единицы при определении *i* и *w*? Matches – метод класса **string**, поэтому он отсчитывает индексы от 0. А срезы отсчитывают индексы от 1.

Задача 8.16
Вывести в алфавитном порядке те из букв русского алфавита (включая букву «ё»), которые отсутствуют в заданной строке.

Простейшее решение состоит в том, чтобы исключить из массива, содержащего все 33 символа алфавита в алфавитном порядке, символы заданной строки и сделать это в одном регистре, например, нижнем.

```
##
var Алфавит := ('а'..'я').JoinToString + 'ё';
var Строка := 'В чащах юга жил бы цитрус? Да, но фальшивый экземпляр!';
Алфавит.Except(Строка.ToLower).Print
```

```
ёъ
```

В данном случае строка не включает всего лишь двух букв. Метод Except (разность) для последовательностей и массивов был рассмотрен в главе 6. Уже не раз отмечалось, что нет особой разницы между последовательностями чисел и строк.

Задача 8.17
С клавиатуры вводится строка, содержащая произвольный набор символов. Найти в ней количество цифр «5».

Здесь самое простое – использовать метод Count. Тогда решение можно записать в одну строку.

```
## ReadString.Count(c -> c = '5').Print
```

```
19Kj087HqlUzN6FQO7Oh23zsO9G2gD5tOc5wOJ3S3OR52MyO
3
```

> Часть задач, связанных с обработкой строк, будут рассмотрены в последующих главах.

Глава 9

Типы данных

В этой главе…

Основные понятия
Записи
Тип данных BigInteger
Пример решения задачи

Не следует множить сущее без необходимости.

Бритва Оккама

До сих пор мы делали вид, что в PascalABC.NET существуют два целочисленных типа данных (**integer**, **int64**), вещественный (**real**), логический (**boolean**), символьный (**char**) и строковый (**string**). Но на самом деле их больше. И не просто больше, а больше во много раз!

Язык Паскаль позволяет программисту определять собственные типы данных. При определении можно задавать состав и структуру данных, а также определять операции над данными. Все это делается в разделе описания типов данных, начинающимся ключевым словом **type**. Этот раздел всегда предшествует основной программе.

Можно присвоить уже имеющимся типам дополнительные имена, чтобы сделать программу более наглядной. В этом случае заменяющий тип называют ***синонимом***.

```
type
  Целое = integer;
  ДлинноеЦелое = int64;

begin
  var Произведение: ДлинноеЦелое := 1;
  for var i: Целое := 2 to 20 do
    Произведение *= i;
  Произведение.Print // 20! = 2432902008176640000
end.
```

Помимо создания синонимов, можно определять иные, более сложные типы данных на базе уже известных.

```
type
  Матрица = array[,] of integer;

begin
  var a :=  new Матрица(3, 8);
  a.Print(2)
end.
```

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

9.1. Записи

Записью (**record**) называется набор элементов, в котором каждый элемент имеет имя и называется ***полем записи***.

Пусть требуется написать программу для работы с простыми дробями. Дробь состоит из числителя и знаменателя, причем оба они имеют целочисленный тип. В этом случае можно описать пользовательский тип Дробь, представляющий собой запись с полями Числитель и Знаменатель, имеющими тип **integer**.

```
type
  Дробь = record
    Числитель, Знаменатель: integer
  end;

begin
  var a, b, c: Дробь; // три дроби
  var ma: array of Дробь; // динамический массив дробей
  // ...
end.
```

Запись имеет заголовок **record**, в котором указывается имя записи. Далее следуют описания полей записи без использования ключевого слова **var**. Заголовок не отделяется от описания полей привычной точкой с запятой. Описание записи завершается ключевым словом **end**, перед которым можно не указывать точку с запятой.

Для обращения к полям записи используется точечная нотация. Указывается имя переменной, идентифицирующее запись, а затем через точку – имя поля. Если поле, в свою очередь, имеет собственные поля, после имени поля ставится еще одна точка и т.д. Например, для приведенного выше фрагмента кода можно указать *a.Числитель* или *ma[3].Знаменатель*.

Записи можно присваивать (при этом выполняется копирование содержимого каждого поля) и сравнивать.

Рассмотрение работы с записями не входит в материал данной книги.

9.2. Автоклассы

В разделе 5.10 второй части книги приводились краткие сведения о классах. Если вы плохо помните, о чем там писалось, советую еще раз перелистать эту пару страниц.

Любой класс – тоже тип данных. Описание класса, как и записи, начинается с ключевого слова **type**. Автоклассы успешно заменяют записи, но при этом они намного эффективнее. При передаче записи в другую программную единицу (например, подпрограмму) происходит полное копирование всех данных, а при передаче объекта произойдет всего лишь копирование ссылки. Чтобы инициализировать поля записи, приходится для каждого поля записывать отдельный оператор присваивания. При создании объекта на основе автокласса достаточно в качестве параметров перечислить значения всех полей. Класс потому и называется автоклассом, что в нем автоматически создается конструктор (функция, которая будет вызвана при создании объекта) с инициализацией полей. Рассмотрим, как может выглядеть код с автоклассом, приведенный в предыдущем разделе.

```
type
  Дробь = auto class
    Числитель, Знаменатель: integer
  end;

begin
  var a := new Дробь(3, 7); // дробь 3/7
  var b := new Дробь(-6, 11); // дробь -6/11
  var c := new Дробь; // дробь без инициализации
  var ma: array of Дробь; // динамический массив дробей
  // ...
end.
```

Ключевое слово **new** здесь используется для того, чтобы создать объект на основе класса Дробь. Для массива *ma* приведено только описание, массив будет создаваться позднее. И каждый элемент такого массива должен также создаваться посредством **new**.

Размещенные в классе функции и процедуры (они помещаются до ключевого слова **end**) становятся методами класса.

9.3. Тип данных BigInteger

Этот целочисленный тип данных уже использовался в предыдущих частях книги. Работа с данными типа **BigInteger** достаточно специфична, поэтому рассматривается отдельно.

Тип данных **BigInteger** – это библиотечный тип Microsoft .NET, позволяющий записывать и обрабатывать целые числа практически неограниченной длины.

Для данных типа **BigInteger** реализованы арифметические операции сложения, вычитания, умножения и деления. Операция деления, в отличие от других целочисленных типов данных, возвращает не вещественное значение, а результат целочисленного деления, имеющий тип **BigInteger**. Остаток целочисленного деления a на b можно получить с помощью традиционной для языка Паскаль операции a mod b.

С операцией возведения в степень дела обстоят немного сложнее. Функция Power не работает с типом **BigInteger**. Возвести значение типа **BigInteger** можно только в степень с показателем, приводящимся к типу **integer**, для чего используется операция **, либо вызов статической функции Pow из библиотеки .NET:

```
BigInteger.Pow(Основание, ПоказательСтепени)
```

В этой библиотеке есть немало полезных функций, поэтому при необходимости серьезной работы с «длинной арифметикой», имеет смысл их изучить. В частности, обратите внимание на:

- `BigInteger.GreatestCommonDivisor(a, b)` – НОД чисел a и b;
- `BigInteger.ModPow(p, q, k)` – остаток от целочисленного деления на k значения p в степени q;

9.3.1. Инициализация данных

Казалось бы, какие тут могут быть проблемы: в операторе присваивания слева указываем имя переменной типа **BigInteger**, справа – литерал, изображающий нужное значение. Пока значение не превышает **int64**.MaxValue, т.е. в нем не больше 19 цифр – действительно, никаких проблем. А вот если цифр больше, то компилятор такое

значение забракует, поскольку не сможет создать целочисленную константу необходимого размера. Выход – использовать строковое представление числа. Но тогда придется строку приводить к типу **BigInteger**.

```
var a := BigInteger.Parse('12345678901234567890123456789O');
```

Метод .Parse предполагает, что строка содержит корректное изображение целого числа, возможно со знаком. В случае, если это не так, будет сгенерировано исключение и вы получите сообщение об ошибке, причем произойдет это не при компиляции, а во время выполнения программы: «Ошибка времени выполнения: Не удалось выполнить синтаксический анализ значения».

Конечно, в реальной ситуации написать ерунду в литерале можно разве что себе назло. А вот получить некорректное значение при клавиатурном вводе вполне возможно.

```
var a := BigInteger.Parse(ReadlnString);
```

Для обработки подобной ситуации следует использовать другой метод:

```
var a: BigInteger;
if not BigInteger.TryParse(ReadlnString, a) then
begin
  Print('Неверный ввод');
  exit
end;
```

Преобразовать целочисленное значение или выражение к типу **BigInteger** можно посредством явного приведения типа, например **BigInteger**(26).

9.3.2. Приведение BigInteger к другому типу

Попытка явно привести значение типа **BigInteger** к другому целочисленному типу может привести к возникновению исключения на этапе выполнения программы. Действительно, количество цифр может оказаться чрезмерным и такое преобразование приведет к потере точности, что очень нежелательно в целочисленной арифме-

тике. В то же время, преобразование к вещественному типу исключения не вызовет, поскольку сам факт использования данных такого типа свидетельствует о том, что программист заранее готов в какой-то степени жертвовать точностью.

```
##
var a := BigInteger.MinusOne; // это -1
var b := integer(a);
a := BigInteger.Parse('123456789012345678901234 5');
Print(b, real(a)) // -1 1.23456789012346E+24
var s := a.ToString; // к строке приводим обычным методом
Println(s) // 123456789012345678901234 5
```

Задача 9.1

Вычислить значение 123! (факториал). Подсчитать, сколько раз в нем встречается цифра «3»

```
##
var p := BigInteger.One; // Число 1 типа BigInteger
for var i := 2 to 123 do
  p *= i;
p.Println;
p.ToString.Count(t -> t = '3').Print
```

```
1214630436702532967576624324188129585545421708848338231532891816182923589236216766883115696061264020217073583522129404778259109157041165147218602951990626164673073390741981495296000000000000000000000000000000000
16
```

Глава 10

Множества

В этой главе…

Создание множеств
Операции над множествами
Примеры решения задач

Множество — это большое количество, которое позволяет воспринимать себя как одно».

Георг Кантор

Формат этой книги предполагает, что вы знакомы с понятием множества из курса школьной математики. Но тут мы его несколько расширим.

Множество – редактируемая неупорядоченная совокупность уникальных элементов, обладающих каким-либо общим свойством, позволяющим отнести все элементы к определенному типу T (базовому типу множества).

В программах множества всегда имеют определенное количество элементов, именуемое ***мощностью множества***. Множество похоже на последовательность, но в отличие от нее, под все элементы множества выделяется память. Во множестве не выделяют первого, следующего и последнего элемента – оно неупорядоченно. Элементы множества можно перебрать по одному, но при этом нельзя указать, с какого элемента начать перебор. Элементы множества уникальны. Например, во множестве чисел не могут находиться два числа с одинаковыми значениями. В отличие от последовательности, множество можно редактировать: включать дополнительные элементы и исключать существующие.

10.1. Создание множества Set of T

Множество описывается с использованием ключевого слова **set of**, за которым указывается тип элементов множества.

```
var s1 : set of integer;
var s2 : set of real;
```

Описание, подобное приведенному выше, приводит к созданию пустого множества, т.е. множества с мощностью ноль. Описание можно объединить с заполнением множества некоторыми элементами, перечислив их в квадратных скобках (такая запись называется ***конструктором множества***).

```
##
var s1: set of integer := [1, 2, 3];
var s2: set of real := [5, 0.12];
Writeln(s1, NewLine, s2)
```

```
{3,2,1}
{5,0.12}
```

Обратите внимание, что содержимое последовательности оператор Write выводит в фигурных скобках. Можно также пользоваться оператором Print и методом .Print.

Конструктор множества содержит список элементов, разделенных запятыми и заключенный в квадратные скобки. Список может состоять из единственного элемента и даже быть пустым. В качестве элемента множества могут быть использованы литералы, выражения соответствующего типа, а также диапазонные значения вида *m..n*, соответствующие перечислению значений от *m* до *n*. При этом значения *m* и *n* должны принадлежать одному порядковому типу. Конструктор указывается в качестве правой части в операторе присваивания. Если тип множества не указан, а в конструкторе записаны элементы различных типов, определяется наиболее общий тип, который и объявляется базовым типом множества.

```
##
var s1: set of real;
var s2 := [-10, 0, -32, 5..12, 28..34, 19];
s1 := [8.5, 11, 13.7, -5.192];
Println(s1);
s2.Print
```

```
{-5.192,11,13.7,8.5}
34 33 32 -10 31 29 28 -32 19 30 12 11 10 9 8 7 6 5 0
```

Основная цель поддержки множеств **set of T** в PascalABC.NET – совместимость с базовым Паскалем. Эти множества плохо укладываются в концепции современного программирования, поскольку для операций с ними предусмотрен скудный набор средств. Полноценной заменой им может служить HashSet – ***коллекция*** Microsoft .NET Framework.

10.2. Создание множества HashSet

Современные средства для организации множеств поддерживают ряд дополнительных операций, в частности, удаление всех элементов, удовлетворяющих условию, копирование элементов множества в массив, объединение множеств, нахождения их пересечения, разности (в том числе, симметрической) и т.д. Тип элементов множества при этом может быть произвольным (но, конечно же, единым), а на их количество накладываются очень слабые по сравнению с множествами Set of T ограничения.

В PascalABC.NET множество HashSet реализовано на основе стандартной коллекции System.Collections.Generic.HashSet. Множество является стандартным типом языка, поэтому его можно создать с использованием оператора присваивания вида

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

В качестве типа данных, помещаемых во множество, можно указать любой тип, в том числе, предварительно определенный пользователем. Типом данных, помещаемых во множество, может быть и множество.

```
type
  r1 = record
    f1: integer;
    f2: real;
    f3: array of boolean;
    f4: string
  end;

begin
  var a := new HashSet<boolean>;
  var b := new HashSet<array[,] of real>;
  var c := new HashSet<r1>;
end.
```

Создание множества можно совместить с его заполнением элементами последовательности (массива, списка и т.п.), указанной в качестве параметра. Множество HashSet можно привести в строке, массиву, списку и т.д. посредством расширений .ToString, .ToArray, .ToList и т.п.

```
##
var h1 := new HashSet<integer>(SeqRandom(4, 10, 99));
h1.Println; // 54 92 42 33
var h2 := new HashSet<integer>(h1.Sorted);
h2.Println // 33 42 54 92
```

В приведенном примере множество h1 заполняется данными от генератора случайных чисел. Множество h2 заполняется данными из множества h1, которые предварительно сортируются в порядке возрастания.

Множество можно заполнить непосредственно перечисленными данными, воспользовавшись «короткой функцией» HSet:

```
var h := HSet(25, -23, 47, 100, 0, 14);
```

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

Множество также можно создать при помощи расширения .ToHashSet из данных, приводящихся к последовательности.

```
##
var h := SeqRandom(8, -999, 999).ToHashSet;
h.Println // -471 289 -499 611 -835 -568 212 -869
```

Аргументом функции HSet может также быть объект любого типа, в том числе, члены последовательности. В множество можно поместить как сложный объект, так и последовательность отдельных значений базового типа. Изучите приведенный ниже пример!

```
##
var h1 := HSet(Seq(14, 0, 172, -5, 0, 39, 14));
h1.Println; // 14 172 -5 0 39
var h3 := HSet(ArrRandom(5, 0, 1));
h3.Println; // 0 1 – элементы множества уникальны
var h4 := HSet(h3); // один элемент – множество !
h4.Println; // (0,1)
var h5 := HSet('Приветик!'); // параметр - строка
h5.Println(' '); // Приветик!
var h6 := HSet('Приветик!'.ToCharArray); // один массив !
h6.Println(' '); // [П,р,и,в,е,т,и,к,!]
h6.Count.Println; // 1 – действительно один элемент
var h7 := HSet('Приветик!'.Select(t -> t));
h7.Println(' ') // П р и в е т к ! – второе «и» пропадает
```

Множества HashSet и Set of T совместимы по присваиванию.

```
##
var s1 := [5, 8, 11, 4, -2, 0, 7];
s1.Println; // -2 11 8 7 5 4 0
var s2: HashSet<integer>;
s2 := s1;
s2.Println; // -2 11 8 7 5 4 0
var s3: set of integer;
s3 := s2;
s3.Println // -2 11 8 7 5 4 0
```

10.3. Операции над множествами

В PascalABC.NET реализовано большинство операций, которые определены в математике для множеств; остальные операции могут быть легко смоделированы.

10.3.1. Перебор элементов в цикле foreach

Ничего нового тут нет, все так же, как для последовательностей. Но это единственный способ получить доступ к элементам множества с тем, чтобы произвести с ними какое-то действие. Ведь элементы множества не упорядочены и не нумерованы, поэтому нельзя напрямую обратиться к какому-то конкретному элементу. С точки зрения современных концепций программирования это архаизм.

```
##
var s := [5, 8, 11, 6, -3, 18, 40, 26];
Println(s);
var n := 0;
foreach var c in s do
  if c.IsOdd then n += 1;
Println('Число нечетных элементов в множестве равно', n)
```

```
{11,-3,26,8,40,5,6,18}
Число нечетных элементов в множестве равно 3
```

Вот столько приходится писать для того, чтобы узнать, сколько нечетных элементов содержит множество Set of T. Параметр цикла *c* поочередно предоставляет доступ к каждому элементу множества.

Посмотрим, как может улучшить ситуацию использование множества HashSet.

```
##
var s := HSet(5, 8, 11, 6, -3, 18, 40, 26);
s.Println;
Println('Число нечетных элементов в множестве равно',
    s.Count(t -> t.IsOdd))
```

10.3.2. Добавление элемента

Классический способ, описанный Н.Виртом – использовать процедуру Include(ИмяМножества, ДобавляемыйЭлемент). Второй способ – использовать операцию +=; при этом добавляемый элемент заключается в квадратные скобки, что превращает его в одноэлементное множество. Для множеств HashSet вместо Include надо пользоваться расширением .Add, а при использовании операции += квадратные скобки не указываются.

В приведенном ниже примере обратите внимание, что порядок следования элементов в множестве Set of T меняется хаотично, а в HashSet элементы хранятся в порядке поступления.

```
##
var s := [5, 8, 11, 6, -3, 18, 40, 26];
Println(s);
s += [7]; // в квадратных скобках для Set of T
s += [12];
Include(s, 0); // только для Set of T
Include(s, 13); // только для Set of T
Println(s);
var s1 := HSet(5, 8, 11, 6, -3, 18, 40, 26);
s1.Println;
s1 += 7; // без квадратных скобок для HashSet
s1.Add(13); // только для HashSet
s1.Println;
```

```
{11,-3,26,8,40,5,6,18}
{40,13,12,11,26,8,7,-3,5,6,18,0}
5 8 11 6 -3 18 40 26
5 8 11 6 -3 18 40 26 7 13
```

10.3.3. Удаление элемента из множества

Н.Вирт использовал для этой цели процедуру Exclude(ИмяМножества, УдаляемыйЭлемент). PascalABC.NET предлагает второй способ – использовать операцию -=. Если запрашиваемый для удаления эле-

мент во множестве отсутствует, попытка удаления ошибкой не считается. Увлекшись удалением, можно даже получить пустое множество и бесконечно продолжать процесс, что иногда и делают начинающие программисты вместо того, чтобы просто написать s:=[ ]. Для множеств HashSet вместо Exclude надо пользоваться расширениями .Remove или .RemoveWhere, а при использовании операции -= квадратные скобки не указываются. Очистить множество HashSet позволяет расширение .Clear.

```
##
var s := [5, 8, 11, 6, -3, 18, 40, 26];
Println(s);
var s1 := new HashSet<integer>;
s1 := s;
s -= [26];
Exclude(s, 18);
s -= [9]; // не ошибка
Exclude(s, 10); // не ошибка
Println(s);
s1.Println;
s1 -= 26;
s1.Remove(11);
s1.RemoveWhere(t -> t.IsEven); // удаление по условию
s1.Println;
```

```
{11,-3,26,8,40,5,6,18}
{40,11,-3,8,6,5}
11 -3 26 8 40 5 6 18
-3 5
```

10.3.4. Проверка наличия элемента

Проверить, имеется ли во множестве элемент с заданным значением позволяет операция **in**. Она используется в логическом выражении вида

```
Элемент in Множество
```

Ирония в том, что всю эту возню со множествами часто затевают именно ради возможности осуществлять такие проверки. К счастью, множество можно не объявлять, а задавать конструктором или формировать «на лету». А иногда возможно обойтись и без множеств. Так, в приведенном ниже примере вместо [1..9] достаточно указать 1..9.

```
##
var n := ReadInteger;
if n in [1..9] then
  Println('Введено натуральное однозначное число')
```

Здесь множество [1..9] используется непосредственно, без объявления – тот самый случай, когда множество весьма полезно.

Задача 10.1
Дана последовательность из n случайных чисел, принадлежащих отрезку [-182; 317]. Вывести числа, в которых встречается цифра 1, 3 или 6.

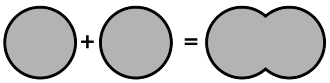
```
##
var n := ReadInteger;
var a := ArrRandom(n, -182, 317);
a.Println;
foreach var m in a do
begin
  foreach var c in m.ToString do
    if c in ['1', '3', '6'] then
    begin
      m.Print;
      break
    end
end
```

```
15
313 195 -151 -8 -157 245 -96 317 -136 205 229 20 234 179 -174
313 195 -151 -157 -96 317 -136 234 179 -174
```

Те, кто не испытывает проблем с использованием возможностей функционального стиля, могут записать решение короче:

```
##
var n := ReadInteger;
var a := ArrRandom(n, -182, 317);
a.Println;
a.Where(m -> m.ToString.Any(c -> c in ['1','3','6'])).Print
```

10.3.5. Объединение множеств (+)



Два или более множеств можно объединить, соединив их имена знаком операции «плюс» (+). При объединении полу-

чается множество, содержащее без повторов все элементы, имеющиеся в исходных множествах.

```
##
([1..3] + [7..12] + [-3, 0, 2, 5, 9]).Println;
(HSet(1..3) + HSet(7..12) + HSet(-3,0,2,6,9)).Print
```

```
12 11 10 9 8 7 -3 5 3 2 1 0
1 2 3 7 8 9 10 11 12 -3 0 6
```

Для HashSet дополнительно имеется метод UnionWith(p), который позволяет добавить к имеющемуся множеству элементы, заданные параметром *p*, приводящимся к последовательности.

```
##
var a := HSet('A'..'Z');
a.Println;
a.UnionWith('абракадабра');
a.Print
```

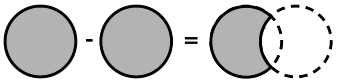
```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZабркд
```

А вот так можно сформировать **упорядоченное** множество букв русского алфавита:

```
HSet(('А'..'Е')+'Ё'+('Ж'..'Я'));
```

В начале главы говорилось, что множество неупорядочено. Но HashSet – это не совсем множество. Это коллекция, обладающая в том числе, свойствами множества.

10.3.6. Разность множеств (-)



Для получения разности множеств используется операция «минус» (-). Разностью множеств p и q является множество, в которое войдут лишь те элементы из p, которых нет в q.

```
##
([1..15] - [-3, 0, 2, 5, 9]).Println;
var a := HSet(1..15);
var b := HSet(-3, 0, 2, 5, 9);
(a - b).Println;
a.ExceptWith(b); // Только для HashSet
a.Print
```

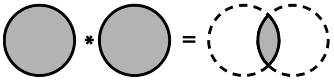
```
15 14 13 12 11 10 8 7 6 4 3 1
1 3 4 6 7 8 10 11 12 13 14 15
1 3 4 6 7 8 10 11 12 13 14 15
```

В методе ExceptWith(p) параметр *p* может быть любого типа, приводящегося к последовательности.

```
##
var a := HSet('а'..'я');
a.ExceptWith('информатика в школе');
Print(a)
```

```
{б,г,д,ж,з,й,п,с,у,х,ц,ч,щ,ъ,ы,ь,э,ю,я}
```

10.3.7. Пересечение множеств (*)



Пересечение множеств выполняется при помощи операции «звездочка» (*). Результат операции пересечения множеств – новое множество, состоящее только из тех элементов, которые присутствуют в обоих исходных множествах.

```
##
([1..15] * [-3, 0, 2, 5, 9]).Println;
(HSet(1..15) * HSet(-3, 0, 2, 5, 9)).Println;
var a := HSet(1..15);
a.IntersectWith(HSet(-3, 0, 2, 5, 9)); // Только для HashSet
a.Print
```

```
5 9 2
2 5 9
2 5 9
```

В методе IntersectWith(p) параметр *p* может быть любого типа, приводящегося к последовательности. Сравните результат с полученным в предыдущем подразделе.

```
##
var a := HSet('а'..'я');
a.IntersectWith('информатика в школе');
Print(a)
```

```
{а,в,е,и,к,л,м,н,о,р,т,ф,ш}
```

10.3.8. Симметрическая разность

Метод s.SymmetricExceptWith(p) исключает из множества *s* элементы, которые одновременно присутствуют в этом множестве и последовательности *p*. Определен только для последовательности HashSet.

```
##
var a := HSet('а'..'я');
a.SymmetricExceptWith('информатика в школе');
Print(a)
```

```
{б,г,д,ж,з,й,п,с,у,х,ц,ч,щ,ъ,ы,ь,э,ю,я, }
```

10.3.9. Сравнение множеств

Над множествами определены шесть операций сравнения подобно тому, как над парой значений определены шесть операций отношения. И даже знаки операций сравнения совпадают со знаками операций отношения. Операции сравнения возвращают логическое значение.

§1. Равенство множеств (=)

Два множества равны, если они содержат одни и те же элементы.

```
##
Print([3, 1, 2] = [1, 2, 3]); // равны
Print([1, 2, 3] = [1, 2, 4]); // не равны
Print(HSet(3, 1, 2) = HSet(1, 2, 3)); // равны
Print(HSet(1, 2, 3).SetEquals(HSet(1, 2, 4))); // не равны
```

```
True False True False
```

Метод s.SetEquals(p) устанавливает равенство (эквивалентность) элементов множества *s* и элементов последовательности *p*.

```
##
var a := ' авеиклмнортфш'.ToHashSet;
Println(a);
a.SetEquals('информатика в школе').Print
```

```
{ ,а,в,е,и,к,л,м,н,о,р,т,ф,ш}
True
```

§2. Неравенство множеств (<>)

Два множества не равны, если существует хотя бы один элемент, который содержится в одном множестве и отсутствует в другом. Это операция, обратная равенству множеств, поэтому если выполняется равенство, не выполняется неравенство и наоборот.

```
##
Print([3, 1, 2] <> [1, 2, 3]); // неравенство не выполнено
Print([1, 2, 3] <> [1, 2, 4]) // неравенство выполнено
```

```
False True
```

§3. Строгое вложение (<)

Для множеств p и q отношение p < q истинно, когда все элементы p содержатся в q, но не все элементы q содержатся в p.

```
##
([3, 1, 2] < [1, 2, 3, 4]).Print; // верно
([1, 2, 3] < [2, 3, 1]).Print; // неверно
([1, 2, 3] < [2, 3, 4]).Println; // неверно
HSet(3, 1, 2).IsProperSubsetOf(HSet(1, 2, 3, 4)).Print;
HSet(1, 2, 3).IsProperSubsetOf(HSet(2, 3, 1)).Print;
HSet(1, 2, 3).IsProperSubsetOf(HSet(2, 3, 4)).Print
```

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

Метод s.IsProperSubsetOf(p) возвращает True, если множество *s* является строгим подмножеством последовательности *p* (все элементы *s* присутствуют в *p* и при этом *p* содержит дополнительные элементы) и False в противном случае. Метод определен только для HashSet, но с HashSet можно также использовать операцию <.

§4. Нестрогое вложение (<=)

Для множеств p и q отношение p < q истинно, когда все элементы p содержатся в q. Для HashSet дополнительно определен метод s.IsSubsetOf(p), возвращающий True, если множество *s* является подмножеством последовательности *p* (все элементы множества *s* присутствуют в последовательности *p*) и False в противном случае.

```
##
([3, 1, 2] <= [1, 2, 3, 4]).Print; // верно
([1, 2, 3] <= [2, 3, 1]).Print; // верно
([1, 2, 3] <= [2, 3, 4]).Println; // неверно
HSet(3, 1, 2).IsSubsetOf(HSet(1, 2, 3, 4)).Print;
HSet(1, 2, 3).IsSubsetOf(HSet(2, 3, 1)).Print;
HSet(1, 2, 3).IsSubsetOf(HSet(2, 3, 4)).Print
```

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

§5. Строго содержит (>)

Для множеств p и q отношение p > q истинно, когда все элементы q содержатся в p, но не все элементы p содержатся в q. Это операция, обратная строгому вложению. Для HashSet дополнительно определен метод s.IsProperSupersetOf(p), возвращающий True, если множество *s* является строгим надмножеством над последовательностью *p* (все элементы последовательности *p* присутствуют во множестве *s* и при этом оно содержит дополнительные элементы) и False в противном случае.

```
##
([3, 1, 2, 4] > [1, 2, 4]).Print; // верно
([1, 2, 3] > [2, 3, 1]).Print; // неверно
([1, 2, 3] > [2, 3, 4]).Println; // неверно
HSet(3, 1, 2, 4).IsProperSupersetOf(HSet(1, 2, 4)).Print;
HSet(1, 2, 3).IsProperSupersetOf(HSet(2, 3, 1)).Print;
HSet(1, 2, 3).IsProperSupersetOf(HSet(2, 3, 4)).Print
```

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

§6. Нестрого содержит (>=)

Для множеств p и q отношение p >= q истинно, когда все элементы q содержатся в p. Это операция, обратная нестрогому вложению. Для HashSet дополнительно определен метод s.IsSupersetOf(p), возвращающий True, если множество *s* является надмножеством над последовательностью *p* (все элементы последовательности *p* присутствуют во множестве *s*) и False в противном случае.

```
##
([3, 1, 2, 4] >= [1, 2, 4]).Print; // верно
([1, 2, 3] >= [2, 3, 1]).Print; // верно
([4, 2, 3] >= [1, 3, 4]).Println; // неверно
HSet(3, 1, 2, 4).IsSupersetOf(HSet(1, 2, 4)).Print;
HSet(1, 2, 3).IsSupersetOf(HSet(2, 3, 1)).Print;
HSet(4, 2, 3).IsSupersetOf(HSet(1, 3, 4)).Print
```

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

10.3.10. Дополнительные методы HashSet

Метод s.Contains(r) возвращает True, если элемент со значением *r* присутствует во множестве *s* и False в противном случае (проверка на наличие элемента во множестве).

Метод s.CopyTo(a, i) копирует элементы множества *s* в существующий одномерный динамический массив *a*, начиная с элемента, указанного индексом *i*. Если данные не помещаются в массиве, возникает исключение.

Метод s.Overlaps(p) возвращает True, если и во множестве *s*, и в последовательности *p* есть хотя бы один совпадающий элемент и False в противном случае.

10.4. Примеры решения задач

Множества удобно использовать в случаях, когда требуется получать и хранить наборы данных с уникальными значениями. Другое назначение множеств – выполнение над ними теоретико-множественных операций, таких как пересечение, объединение, разность и т.д.

Задача 10.2
Получить массив, содержащий 20 случайных натуральных неповторяющихся двухзначных чисел.

Можно воспользоваться множеством, помещая в него случайные числа до тех пор, пока в нем не окажется 20 элементов. Затем преобразовать это множество в массив.

```
##
var s := SeqRandom(20, 10, 99).ToHashSet;
while s.Count < 20 do
  s += Random(10, 99);
var a := s.ToArray;
a.Print
```

```
52 25 56 23 63 92 93 10 71 42 44 68 15 88 64 26 19 97 30 98
```

Здесь сначала делается попытка создать множество из последовательности, содержащей 20 натуральных случайных чисел, заданных на отрезке [10;99]. Если в результате мощность множества окажется меньше 20, остальные элементы будут добавлены в цикле.

Задача 10.3
Дана строка, содержащая слова, разделенные одним или более пробелом. Верно ли, что в строке имеется слово, содержащее все буквы, использованные для составления остальных слов?

```
begin
  var s := '  апельсин  лепнина спаниель   писанина';
  var L := new List<HashSet<char>>;
  foreach var w in s.ToWords do
    L.Add(w.ToHashSet);
  var imax := L.Select(t -> t.Count).ToArray.IndexMax;
  var Smax := L[imax];
  L.RemoveAt(imax);
  var good := True;
  foreach var ss in L do
    if not (ss <= Smax) then
    begin
      good := False;
      break
    end;
  if good then Println('Верно')
  else Println('Неверно')
end.
```

```
Верно
```

Строим список, элементами которого являются множества HashSet, содержащие набор символов, из которых составлены соответствующие слова. Определяем индекс элемента с множеством, содержащим максимальное количество символов. Этот элемент помещаем в множество Smax, а затем удаляем из списка. Остается лишь проверить

выполнение нестрогого вложения между каждым оставшимся в списке множеством и Smax.

Задача 10.4
Определить количество гласных и согласных русских букв во введенной строке. Учитывать букву «ё».

```
##
var Строка := ReadString('Введите текст:').ToLower;
var Гласные := 'аеёиоуыэюя'.ToHashSet; // запомните этот прием
var Согласные := HSet('а'..'я') - Гласные;  // и этот тоже
var (Гласных, Согласных) := (0, 0);
foreach var Символ in Строка do
  if Символ in Гласные then
    Гласных += 1
  else if Символ in Согласные then
    Согласных += 1;
$'Гласных:{Гласных}, согласных:{Согласных}'.Print
```

```
Введите текст: В PascalABC.NET удобно работать с множествами HashSet
Гласных:10, согласных:17
```

Обратите внимание на приемы, позволяющие быстро формировать подмножества символов и не писать множество одинарных кавычек. Последний оператор в коде программ использует интерполированную строку. Привыкайте, это компактно и очень удобно.

Вам могут встретиться задачи, в которых предлагается делать сложный анализ текста – находить частоту использования каждой буквы, находить символы, встретившиеся не менее указанного числа раз, выводить все использованные символы в порядке уменьшения частоты их использования и т.п. Такие задачи лучше решать при помощи словаря Dictionary, о котором мы поговорим позднее.

Глава 11

Стеки, очереди, словари

В этой главе…

Основные понятия
Стек
Очередь
Словарь
Пара «ключ – значение»
Примеры решения задач

Если вы наслаждаетесь используемыми инструментами, то работа будет выполнена успешно.

Дональд Кнут,
американский ученый

В программировании существует 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.

PascalABC.NET позволяет напрямую обращаться к ***обобщенным*** стандартным коллекциям Microsoft .NET Framework, реализованным в виде набора классов. Термин «обобщенная» означает, что тип элементов коллекции не фиксирован. Возможно, вы не подозревали, что уже работали с коллекциями, но пришло время сказать правду: списки List и множества HashSet представляют собой обобщенные коллекции. А теперь познакомьтесь еще с тремя – стеком, очередью и словарем.

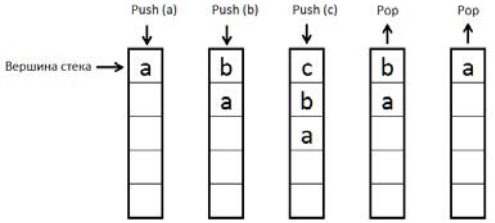
11.1. Стек (Stack)

Стек легко представить себе в виде стопки книг или колоды карт. Над стеком определено три операции. Книгу или карту можно добавить в стопку только сверху (операция push – добавление). Название книги и значение карты видно только для одной книги или карты, лежащей наверху (значение top – доступ к элементу на вершине стека). Наконец, забрать можно лишь то, что лежит сверху (операция pop – извлечение).

Программный стек (англ. Stack – стопка) – список, в котором добавление и исключение элементов производится на одном его конце, называемом ***вершиной***. Особенность стека в том, что он изменяет порядок следования помещенных объектов на противоположный.

Алгоритм работы стека схематически описывается фразой «Последним зашел – первым вышел» (англ. Last In – First Out, сокращенно LIFO).

Рассмотрим принцип работы стека на примере помещения в него объектов «a», «b» и «c» и последующего извлечения объектов «c и «b». После выполнения операции Push(a), объект «a» появляется на вершине стека. Операция Push(b) «проталкивает» объект «a» в глубину стека и теперь на его вершине будет доступен объект «b». Для объекта «c» происходит аналогичное действие. Далее, операция Pop удаляет с вершины стека объект «c», а из глубины стека «поднимается» на вершину объект «b». Затем операция Pop удаляет с вершины стека объект «b» и на вершине стека оказывается объект «a».



Безусловно, физически никакого «проталкивания» данных в стек не происходит, но это уже особенности программной реализации стека.

В PascalABC.NET стек является стандартным типом и его можно создать как обычный объект при помощи конструктора:

```
var имя := new Stack<ТипДанных>;
```

В качестве типа данных, помещаемых в стек, можно указать любой тип, в том числе, предварительно определенный пользователем.

```
type
  r1 = record
    f1: integer;
    f2: real;
    f3: array of boolean;
    f4: string
  end;

begin
  var a := new Stack<Stack<real>>;
  var b := new Stack<array[,] of real>;
  var c := new Stack<r1>;
end.
```

Конструктор стека дает возможность совместить создание стека с его заполнением элементами последовательности (или данными с типом, приводящимся к последовательности), указанной в качестве параметра. При этом порядок следования элементов в новом стеке будет обратным по отношению к принимаемым данным.

```
##
var st1 := new Stack<integer>(SeqGen(5, 1, t -> t + 2));
st1.Println; // 9 7 5 3 1
```

В приведенном примере в стек st1 были помещены элементы последовательности 1, 3, 5, 7 и 9, заданной генератором нечетных чисел. Вывод содержимого стека производится, начиная с его вершины, поэтому на мониторе отображаются значения 9 7 5 3 1. Это еще раз иллюстрирует особенность стека: он изменяет порядок следования данных на противоположный.

11.1.1. Операции для работы со стеком

В работе со стеком *st* поддерживаются следующие средства:

- `st.Count` – свойство, возвращающее количество элементов в стеке;
- `st.Push(p)` – метод, помещающий объект *p* на вершину стека;
- `st.Peek` – метод, возвращающий объект, находящийся на вершине стека, при этом сам объект остается на месте. Может вызываться в виде функции;
- `st.Pop` – метод, удаляющий объект с вершины стека. Возвращает удаляемый объект. Может вызываться в виде функции;
- `st.Clear` – метод, очищающий стек;

- `st.Contains(p)` – метод, возвращающий True, если объект *p* находится в стеке и False в противном случае (проверка на наличие объекта в стеке);
- `st.ToArray` – метод, копирующий содержимое стека в одномерный динамический массив и возвращающий этот массив в качестве результата. Не требует предварительного описания массива;
- `st.CopyTo(a, i)` – метод, копирующий содержимое стека в существующий одномерный динамический массив *a*, начиная с элемента, указанного индексом *i*. Если данные не помещаются в массиве, возникает исключение;
- `st.Print` – стандартный метод PascalABC.NET. Осуществляет вывод содержимого стека, начиная с его вершины. В качестве параметра можно указать символ или строку–разделитель, по умолчанию используется пробел;
- `st.Println` – то же самое; в конце вывода осуществляется переход к новой строке.

Кроме того, к содержимому стека можно применять все методы для работы с последовательностями.

11.1.2. Пример использования стека

Задача 11.1
Проверить корректность расстановки круглых скобок во введенном с клавиатуры выражении.

Предлагается следующий несложный алгоритм:
1. Стек пуст.
2. Выполняем последовательный просмотр всех символов строки.
3. Если очередной символ является круглой открывающей скобкой, заносим его на вершину стека.
4. Если очередной символ является круглой закрывающей скобкой, проверяем стек. Если он не пуст, удаляем символ с вершины, если пуст – фиксируем ошибку.
5. Если после окончания просмотра строки стек не пуст, фиксируем ошибку.

```
##
var St := new Stack<char>;
foreach var c in ReadlnString('Введите выражение:') do
  case c of
    '(': St.Push(c);
    ')':
      if St.Count > 0 then St.Pop
         else
      begin
        Println('Скобки расставлены неверно');
        Exit
      end
  end;
if St.Count = 0 then
  Println('Скобки расставлены верно')
else
  Println('Скобки расставлены неверно')
```

```
Введите выражение: (a+3*b)*((c+d)*a+1)/(a-1)
Скобки расставлены верно
```

11.2. Очередь (Queue)

Понятие очереди взято из реальной жизни. Обслуживание всегда производится в начале очереди, а постановка в очередь происходит в ее конце. В информатике начало очереди называют её ***головой***, а конец – ***хвостом***. Включение в очередь происходит на ее конце (в хвосте), а исключение и доступ – в начале очереди (в голове). В соответствии с дисциплиной обслуживания, очередь часто называют списком FIFO (англ. First In – First Out – первым вошел, первым вышел).

Очередь (англ. Queue) – список, в котором добавление элементов производится на одном его конце, а исключение элементов (и, как правило, доступ к ним) производится на другом его конце.

В PascalABC.NET очередь является стандартным типом и ее можно создать как обычный объект при помощи конструктора:

```
var имя := new Queue<ТипДанных>;
```

В качестве типа данных, помещаемых в очередь, можно указать любой тип, в том числе, предварительно определенный пользователем.

```
type
  r1 = record
    f1: integer;
    f2: real;
    f3: array of boolean;
    f4: string
  end;

begin
  var a := new Queue<Queue<boolean>>;
  var b := new Queue<array[,] of real>;
  var c := new Queue<r1>;
end.
```

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

```
##
var q1 := new Queue<integer>(SeqRandom(4, 10, 99));
q1.Println; // 54 92 42 33
var q2 := new Queue<integer>(q1.Sorted);
q2.Print // 33 42 54 92
```

В приведенном примере очередь q1 заполняется данными от генератора случайных чисел. Очередь q2 заполняется данными из очереди q1, которые предварительно сортируются в порядке возрастания.

11.2.1. Операции для работы с очередью

В работе с очередью *q* поддерживаются следующие средства:

- `q.Count` – свойство, возвращающее количество элементов;
- `q.Enqueue(p)` – метод, помещающий объект *p* в конец очереди *q*;
- `q.Peek` – метод, возвращающий объект, находящийся в начале очереди *q*. Может вызываться в виде функции;
- `q.Dequeue` – метод, удаляющий объект из начала очереди *q*. Возвращает удаляемый объект. Может вызываться в виде функции.

Аналогично стеку, очередь имеет методы .Clear, .Contains, .ToArray, .CopyTo, .Print и .Println. К содержимому очереди можно применять все методы для работы с последовательностями.

11.2.2. Пример использования очереди

Массив, даже динамический, требует указания размера. Очередь этого не требует. Отсюда вытекает очевидное преимущество использования очереди в задачах, где количество данных неизвестно, например, для приема данных заранее неизвестного объема. Впрочем, чаще для этой цели используют список List.

Задача 11.2
С клавиатуры вводится последовательность натуральных чисел, ограниченная нулем. Значение чисел не превышают 999. Вывести значения поступивших чисел по группам: сначала одноразрядные, затем двухразрядные и в конце трехразрядные. Порядок следования чисел в каждой группе не должен изменяться по сравнению с порядком ввода чисел.

Для решения задачи можно создать три очереди, содержимое которых затем вывести.

```
##
var q1 := new Queue<integer>;
var q2 := new Queue<integer>;
var q3 := new Queue<integer>;
foreach var d in ReadSeqIntegerWhile(t -> t <> 0) do
  if d < 10 then
    q1.Enqueue(d)
  else if d < 100 then
    q2.Enqueue(d)
  else
    q3.Enqueue(d);
  q1.Println;
  q2.Println;
  q3.Println
```

```
3 18 42 15 9 14 621 1 42 311 9 0
3 9 1 9
18 42 15 14 42
621 311
```

11.3. Словари

Словарь – это структура данных, в которой единицей хранения является пара «ключ – значение». Словарь позволяет быстро находить

искомое значение, указывая его ключ. Заданный ключ преобразуется в индекс элемента хранения данных (это называется ***ассоциированием***). Добавляемые в словарь данные должны иметь уникальный ключ.

11.3.1. Структура «ключ – значение»

Структура KeyValuePair определяет пару «ключ – значение», которую можно задавать или получать. Типы данных ключа TKey и значения TValue могут быть произвольными.

Для создания объекта KeyValuePair<TKey, TValue> используется конструктор. Поддерживаются свойства .Key и .Value, доступные только для чтения. Объекты класса KeyValuePair можно описывать и создавать следующим образом:

```
##
var kv1: KeyValuePair<integer, string>;
var kv2: KeyValuePair<string, List<integer>>;
var kv3 := new KeyValuePair<integer,
    array of integer>(28, Arr(1, 8, 3, 2));
Writeln(kv3); // (28,[1,8,3,2])
var kv4 := new KeyValuePair<string, real>('j12k29', 1.39);
Writeln(kv4); // (j12k29,1.39)
```

Рассмотрим примеры работы со свойствами .Key и .Value, предоставляющие доступ к ключу и значению соответственно.

```
##
var kv1 := new KeyValuePair<integer,
    array of integer>(28, Arr(1, 3, 4, 2));
Print('kv1 с ключом', kv1.Key, 'хранит массив значений');
kv1.Value.Print;
Print(' с суммой, равной', kv1.Value.Sum)
```

```
kv1 с ключом 28 хранит массив значений 1 3 4 2 с суммой, равной 10
```

Создать и/или инициализировать структуру KeyValuePair можно при помощи короткой функции KV(ключ, значение).

```
##
var kv1 := KV(-18, 'Тест');
Println(kv1); // (-18,Тест)
var kv2: KeyValuePair<char, List<integer>>;
kv2 := KV('*', Lst(1, 3, 5, 7, 2, 4, 6, 0));
Println(kv2) // (*,[1,3,5,7,2,4,6,0])
```

11.3.2. Создание словаря

В PascalABC.NET словарь Dictionary<Tkey, TValue> создается как коллекция объектов KeyValuePair<TKey, TValue> и является стандартным типом языка, поэтому его можно создать с использованием оператора присваивания вида

```
var имя := new Dictionary<ТипКлюча, ТипЗначения>;
```

Типы ключа и значения могут быть произвольными, в том числе предварительно определенными пользователем. При этом не нужно проявлять особого фанатизма в выборе экстравагантного типа для ключей.

```
type
  r1 = record
    f1: integer;
    f2: real;
    f3: array of boolean;
    f4: string
  end;

begin
  var a := new Dictionary<integer, boolean>;
  var b := new Dictionary<integer, array[,] of real>;
  var c := new Dictionary<string, r1>;
end.
```

Конструктор словаря не дает возможности совместить создание словаря с его заполнением данными, но для этой цели можно использовать «короткую функцию» Dict. В качестве источника данных для заполнения словаря можно задавать последовательность пар «ключ – значение» (в виде массива кортежей) или коллекцию объектов KeyValuePair.

```
##
var d1 := Dict(('a1', 142), ('b2', 216), ('d4', 18));
d1.Println;
var d2 := new Dictionary<integer, char>;
var kvp := new KeyValuePair<integer, char>[4];
for var i := 0 to kvp.High do
  kvp[i] := KV(i + 1, Chr(Ord('a') + i));
d2 := Dict(kvp);
d2.Print
```

```
(a1,142) (b2,216) (d4,18)
(1,a) (2,b) (3,c) (4,d)
```

Словарь также можно создать при помощи расширения .ToDictionary из данных, приводящихся к последовательности. При этом необходимо указать хотя бы один параметр в виде лямбда-функции, задающей преобразование элементов последовательности в уникальный ключ. Второй, необязательный параметр, также задается в виде «лямбды» и определяет преобразование элементов последовательности в хранимые значения.

```
##
var a := ArrRandom(4, -99, 99);
var d := a.Distinct
    .ToDictionary(t -> t, t -> Sign(t) * Abs(t) ** (1 / 3));
d.Println
```

В данном случае ключом словаря будет случайное целое число в диапазоне от -99 до 99, а значением – кубический корень из этого числа. Расширение Distinct не позволяет ключам дублироваться.

```
(60,3.91486764116886)
(96,4.57885697021333)
(-67,-4.06154810044568)
(5,1.7099759466767)
```

Элементы словаря Dictionary хранятся неупорядоченными относительно значений своих ключей. Обращение к элементу словаря в формате D[k] подразумевает обращение к данным, хранящимся в элементе, имеющем ключ k. Эти **данные доступны для изменения**. Если указанный ключ не найден, во время исполнения программы возникает исключение, приводящее к выводу сообщения «Ошибка времени выполнения: Данный ключ отсутствует в словаре». Сам **ключ элемента менять недопустимо**.

11.3.3. Операции для работы со словарем

В работе со словарем поддерживаются следующие средства:

- `.Count` – свойство, возвращающее количество элементов в словаре;
- `.Keys` – свойство, возвращающее последовательность ключей, имеющихся в словаре;
- `.Values` – свойство, возвращающее последовательность значений, имеющихся в словаре;
- `.Add(ключ, значение)` – метод, добавляющий в словарь элемент с указанными ключом и значением. При попытке добавления элемента с ключом, уже имеющимся в словаре, возникает исключение;
- `.Clear` – метод, удаляющий из словаря все элементы;
- `.ContainsKey(ключ)` – метод, возвращающий True, если в словаре имеется элемент с указанным ключом и False в противном случае (проверка на наличие ключа в словаре);
- `.ContainsValue(значение)` – метод, возвращающий True, если в словаре имеется элемент с указанным значением и False в противном случае (проверка на наличие значения в словаре);
- `.Get(ключ)` – расширение PascalABC.NET, возвращающее значение элемента, связанного с указанным ключом или значение по умолчанию, если ключ не найден;
- `.Remove(ключ)` – метод, удаляющий из словаря элемент с указанным ключом. Если такого ключа нет, ничего не происходит;
- `.TryGetValue(ключ, имя_переменной)` – метод, выполняющий поиск элемента словаря по указанному ключу. Если поиск успешен, переменная заполняется значением из найденного элемента и возвращается True. В противном случае возвращается False;
- `.Print` – стандартный метод PascalABC.NET. Осуществляет вывод содержимого множества. В качестве параметра можно указать символ или строку - разделитель, по умолчанию используется пробел;
- `.Println` – то же самое; в конце вывода осуществляется переход к новой строке.

К словарям Dictionary можно применять все методы для работы с массивами и последовательностями, но при этом следует помнить, что каждый элемент такой последовательности имеет тип KeyValuePair и в «лямбдах» указывать свойства по типу t -> t.Key, t -> t.Value.

11.3.4. Пример работы со словарем

Задача 11.3
Определить, сколько раз в заданном тексте встречается каждый символ, не учитывая пробелов. Результат расположить в порядке невозрастания частоты, с которой встречаются символы.

```
##
var s := Concat('Язык Паскаль был разработан в 1970 г. ',
    'Никлаусом Виртом как язык, обеспечивающий строгую ',
    'типизацию и интуитивно понятный синтаксис.',
    ' Он был назван в честь французского математика,',
    ' физика и философа Блеза Паскаля.');
var s1 := s.ToLower.Where(c -> c <> ' ');
var d := new Dictionary<char, integer>;
foreach var c in s1 do
  if d.ContainsKey(c) then
    d[c] += 1
  else
    d.Add(c, 1);
d.OrderByDescending(t -> t.Value).ThenBy(t -> t.Key).Print
```

```
(а,21) (и,18) (о,12) (к,11) (н,11) (с,11) (т,11) (з,8) (л,7) (в,6)
(б,5) (е,5) (п,5) (р,5) (ы,5) (м,4) (у,4) (ф,4) (я,4) (.,3) (г,3)
(ю,3) (,,2) (й,2) (ц,2) (ч,2) (ь,2) (0,1) (1,1) (7,1) (9,1) (щ,1)
```

Сначала получаем последовательность символов *s1*, переведя алфавитные символы строки *s* на нижний регистр, а затем исключив пробелы. Для каждого символа производим проверку его наличия в словаре, где символ является ключом, а число повторений – данными. Если ключ найден, число повторений увеличиваем на единицу. В противном случае добавляем в словарь новый элемент, устанавливая число повторений равным единице.

Глава 12

Модули

В этой главе…

Понятие модуля
Стандартные модули
Учебные модули
Примеры решения задачи

Есть только одно действительно неистощимое сокровище — это большая библиотека.

Пьер Буаст,
французский лексикограф

В языке PascalABC.NET программы могут строиться из ***модулей*** – программных компонент особого вида. Модуль позволяет разбить текст программы на несколько физических файлов, компилируемых раздельно. Это дает возможность автономно отлаживать и тестировать модули, а также во многом снимает проблему внесения в готовый код непреднамеренных и несанкционированных изменений.

Еще задолго до появления концепции модулей, на заре становления программирования, при написании программ начали применять ***библиотеки стандартных программ***. Идея использования библиотек состоит во включении в программный код обращений к заранее написанным фрагментам, реализующим некоторые стандартные операции. Математические библиотеки используются для вычисления различных математических функций (таких, как квадратный корень, синус, логарифм). С помощью библиотек ввода-вывода преобразуют вводимые данные в формат, необходимый для вычислений и обеспечивают оформление вывода результатов. Библиотеки математических методов позволяют, к примеру, оперировать матрицами, решать уравнения, находить значения специальных функций. Существуют также библиотеки, позволяющие работать с компьютерной графикой и многие другие.

И модули, и библиотеки стандартных программ, оформляются в виде модулей. А собственно библиотеками в PascalANC.NET называют самостоятельные программные единицы, динамически загружаемые в память из файлов с расширением .dll (Dynamic-Link Libraries). Из модуля выбираются только нужные вызывающей программе компоненты, а библиотека всегда загружается в память компьютера в полном объеме.

Чтобы воспользоваться модулем, его нужно ***подключить***, для чего в первой строке программы используется оператор **uses**

```
uses ИмяМодуля1, ИмяМодуля2, …;
```

Изучение принципов написания собственных модулей не входит в вопросы, которые рассматриваются в этой книге. Вы лишь познакомитесь с некоторыми стандартными модулями и научитесь ими пользоваться.

PascalABC.NET использует как стандартные библиотечные модули Microsoft .NET Framework (их имена начинаются с System.), так и собственные. Исходные тексты всех собственных модулей открыты и доступны для просмотра.

12.1. Стандартные модули

Система программирования на базе языка PascalABC.NET поставляется с набором стандартных модулей. Часть этих модулей описана только в Справке, другие имеют отдельные описания и руководства. Объем и формат данной книги не предоставляют возможности подробно останавливаться на стандартных модулях, поэтому предлагаемые сведения неполны, носят общий описательный характер и в своем большинстве взяты из Справки PascalABC.NET.

12.1.1. Модули для работы с графикой

В Справке дается материал по следующим шести стандартным модулям: GraphABC, ABCObjects, GraphWPF, WPFObjects, Graph3D и ABCSprites.

§1. Модуль GraphABC

Модуль GraphABC – простая графическая библиотека, предназначенная для создания несобытийных графических и анимационных программ в процедурном и частично в объектном стиле. Рисование осуществляется в специальном графическом окне, возможность рисования в нескольких окнах отсутствует. Кроме этого, в модуле GraphABC определены простейшие события мыши и клавиатуры, позволяющие создавать элементарные событийные приложения. Основная сфера использования модуля GraphABC - **обучение**.

Начиная с версии 3.3.0.1531 от 30.08.2017 модуль GraphABC объявлен устаревшим; его обновление и поддержка прекращены. Тем не менее,

в последующие версии модуль GraphABC по-прежнему будет входить. Вместо GraphABC рекомендуется использовать модуль растровой графики GraphWPF.

§2. Модуль GraphWPF

Модуль GraphWPF основан на графической библиотеке WPF и является современной и усовершенствованной версией устаревшего модуля GraphABC. Введен в PascalABC.NET начиная с версии 3.3.0.1531 от 30.08.2017.

§3. Модуль ABCObjects

Модуль ABCObjects создан на основе модуля GraphABC. Он реализует векторные графические объекты с возможностью масштабирования, наложения друг на друга, создания составных графических объектов и многократного их вложения друг в друга. Каждый векторный графический объект корректно себя перерисовывает при перемещении, изменении размеров и частичном перекрытии другими объектами. Модуль предназначен для раннего обучения основам объектно-ориентированного программирования, а также для реализации графических и анимационных проектов средней сложности.

§4. Модуль WPFObjects

Модуль WPFObjects основан на графической библиотеке WPF и является современной и усовершенствованной версией устаревшего модуля ABCObjects. Введен в PascalABC.NET начиная с версии 3.4.2.1782 от 01.09.2018.

§5. Модуль Graph3D

Модуль Graph3D основан на библиотеке Helix Toolkit WPF. Он реализует трехмерные графические объекты с возможностью масштабирования, группировки, поворотов, перемещения, создания дочерних объектов. Предназначен для обучения основам объектно-ориентированного программирования и основам 3D-графики. Позволяет создавать простейшие трехмерные анимации и простейшие интерактивные трехмерные программы и игры.

Модуль Graph3D введен в PascalABC.NET начиная с версии 3.3.0.1531 от 30.08.2017. Для работы требует библиотеки Microsoft .NET 4.5. При работе под операционной системой Windows XP (SP2) возможно возникновение проблем, которые не могут быть устранены из-за прекращения поддержки Windows XP фирмой Microsoft.

§6. Модуль ABCSprites

Модуль ABCSprites реализует спрайты - анимационные объекты с автоматически меняющимися кадрами. Спрайты автоматически анимируются в цикле, что управляется специальным таймером из модуля Timers (см. Справку и поставляемые примеры кода).

12.1.2. Учебные модули

Учебные модули, как следует из их названия, предназначены для решения задач, связанных с обучением. Так, модули Робот и Чертежник, представляют собой реализацию Исполнителей, использующихся на уроках школьной информатики, а также в электронном задачнике РТ4 (автор – М. Э. Абрамян), входящем в комплект поставки PascalABC.NET. Модуль NumLibABC реализует достаточно мощную библиотеку численных методов, которая может быть использована не только для обучения, но и для выполнения инженерных и научных расчетов. Модуль SF, позволяющий существенно ускорять написание кода с помощью двух и трехбуквенных сокращений, может использоваться на олимпиадах. Модуль School, содержащий реализацию часто используемых алгоритмов, будет весьма полезен на стадии обучения программированию.

§1. Модуль Robot

Исполнитель Робот, описанный в учебнике А. Г. Кушниренко, Г. В. Лебедева и Я. Н. Зайдельмана «Информатика 7–9 классы», М., 2001, реализован в виде модуля Robot. Исполнитель действует на прямоугольном клеточном поле. Между некоторыми клетками, а также по периметру поля, находятся стены. Основная цель Робота – закрасить указанные клетки и переместиться в конечную клетку.

§2. Модуль Drawman

Исполнитель Чертежник, описанный в учебнике А. Г. Кушниренко, Г. В. Лебедева и Я. Н. Зайдельмана «Информатика 7–9 классы», М., 2001, реализован в виде модуля Drawman. Исполнитель предназначен для построения рисунков и чертежей на плоскости с координатами. Чертежник имеет перо, которое он может поднимать, опускать и перемещать. При перемещении опущенного пера за ним остается след.

§3. Модуль NumLibABC

Библиотека численных методов NumLibABC в настоящее время в Справке не представлена, но имеет самостоятельное справочное пособие в виде файла с именем NumLibABC.pdf, который находится в папке \PascalABC.NET\Doc. В справочном пособии кроме назначения и форматов вызова дается необходимая для понимания вопроса теория, а также приводятся примеры. Библиотека реализована в виде набора классов, поэтому ее исходный текст может использоваться при изучении основ объектно–ориентированного программирования. Модуль NumLibABC был введен в PascalABC.NET начиная с версии 3.3.0.1552 от 20.10.2017.

NumLibABC – свободно распространяемая библиотека, реализованная в системе программирования PascalABC.NET 3.5 и поставляющаяся вместе с ней, в том числе, в исходном коде. В пакете находятся программы, реализующие различные численные методы посредством классов, а также вспомогательные программы и сопутствующие типы данных.

С помощью пакета NumLibABC можно решать задачи из следующих областей:

- нахождение корней нелинейных уравнений;
- операции с простыми дробями;
- операции с полиномами;
- интерполяция, дифференцирование и аппроксимация данных, заданных в табличном виде;
- операции с векторами и матрицами, решение систем линейных уравнений;

- решение систем дифференциальных уравнений;
- вычисление определенных интегралов;
- задачи оптимизации.

Часть программ переведена в Паскаль на уровне исходного текста из существующих пакетов прикладных программ, таких как SSPLIB (на языке Фортран) или опубликованных в литературе. Другая часть написана на основе алгоритмов, приведенных в различных источниках. В любом случае, для поддержания чистоты открытой лицензии, приводятся ссылки на источник.

§4. Модуль SF

Модуль SF (Short Functions – короткие функции) появился, чтобы дать возможность продвинутым пользователям быстро писать программы в условиях лимита времени – на олимпиадах, экзаменах, контрольных и проверочных работах.

«Короткое» подключение библиотеки происходит, если первой строкой программы вместо begin указать ### - в этом случае конечный end. (с точкой) не указывается.

Модуль SF в основном содержит синонимы имеющихся функций – ввода, вывода и преобразования типа. Нашлось в ней место и недолюбливаемой за длину операции mod: выражение a mod b = 0 можно записать в виде a.D(b).

Также, можно оперативно организовать себе «калькулятор», воспользовавшись шаблоном ### Pr( ). Подлежащее вычислению выражение записывается внутри круглых скобок.

§5. Модуль School

Модуль School (англ. «школа») содержит реализацию алгоритмов, часто встречающихся в школьных задачах. Каждая реализация в основном имеет два формата – для вызова в виде функции и для записи в точечной нотации.

Алгоритмы можно разбить на несколько групп.

1. Перевод десятичного числа *n* в другую систему счисления
– функция Bin(n) преобразует *n* в двоичную строку;
– функция Oct(n) преобразует *n* в восьмеричную строку;
– функция Hex(n) преобразует *n* в шестнадцатеричную строку;
– функция ToBase(sn, k) преобразует значение *n*, заданное строкой sn, в строку, записанную в системе счисления с основанием *k* (k = 2..36);
– расширение sn.ToBase(k) делает то же, что ToBase.

2. Перевод числа, заданного символьной строкой *sn*, представляющей значение в системе счисления по основанию *k* (k = 2..36), в десятичную систему счисления
– функция Dec(sn, k) возвращает значение типа int64;
– функция DecBig(sn, k) возвращает значение типа BigInteger.

3. Нахождение минимума и максимума последовательности целых чисел *s* за один проход
– функция MinMax(*s*) возвращает кортеж (Min, Max). Тип данных **integer** или **int64** – в зависимости от типа последовательности;
– расширение s.MinMax делает то же самое.

4. Нахождение НОД и НОК пары чисел *a* и *b*
– функция НОД(a, b) возвращает НОД типа **integer** или **int64**;
– расширение t.НОД возвращает НОД для кортежа t = (a, b) с данными типа **integer** или **int64**;
– функция НОК(a, b) возвращает НОК типа **int64**;
– функция НОДНОК(a, b) возвращает кортеж вида (НОД, НОК) для пары чисел *a* и *b* типа **int64**.

5. Разложение числа n на простые множители. Результат помещается в список List
– функция Factorize(n) выполняет разложение числа n типа **integer** или **int64**;
– расширение n.Factorize делает то же самое.

6. Простые числа
– функция Primes(n) возвращает список List, содержащий простые числа на отрезке [2;n];
– функция FirstPrimes(n) возвращает список List, содержащий первые *n* простых чисел;

– расширение n.IsPrime возвращает **True**, если *n* – простое и **False** в противном случае. Переменная *n* должна быть типа **integer** или **int64**.

7. Получение списка List, содержащего все цифры числа *n* в порядке их следования слева направо
– функция Digits(n) возвращает список типа List<**int64**>;
– расширение n.Digits делает то же, возвращая список типа List<**integer**> или List<**int64**>.

8. Получение списка List, содержащего все натуральные делители числа *n*, включая 1 и *n*
– функция Divisors(n), в зависимости от типа *n*, возвращает значение типа List<**integer**> или List<**int64**>;
– расширение n.Divisors делает то же самое.

9. Тригонометрия. Функции SinDegrees(n), CosDegrees(n) и TanDegrees(n) возвращают значения синуса, косинуса и тангенса для вещественного аргумента *n*, заданного в градусной мере.

10. Генерация случайных вещественных чисел
– функция ArrRandomReal(n, a, b, t) возвращает массив длины n, заполненный случайными вещественными числами из промежутка [*a*;*b*) с *t* знаками в дробной части;
– функция SeqRandomReal(n, a, b, t) возвращает последовательность длины n, заполненную случайными вещественными числами из промежутка [*a*;*b*) с *t* знаками в дробной части;
– функция MatrRandomReal(m, n, a, b, t) возвращает массив размером *m*×*n*, заполненный случайными вещественными числами из промежутка [*a*;*b*) с *t* знаками в дробной части.

11. Построение таблиц истинности
– расширение a.Imp(b) возвращает результат операции импликации a → b;
– функция TrueTable((a, b) -> f(a, b)) возвращает матрицу типа boolean, содержашую таблицу истинности для заданной функции двух аргументов;
– функция TrueTable((a, b, c) -> f(a, b, c)) возвращает матрицу типа boolean, содержашую таблицу истинности для заданной функции трех аргументов;

– функция TrueTable((a, b, c, d) -> f(a, b, c, d)) возвращает матрицу типа boolean, содержашую таблицу истинности для заданной функции четырех аргументов;
– функция TrueTable((a, b, c, d, e) -> f(a, b, c, d, e)) возвращает матрицу типа boolean, содержащую таблицу истинности для заданной функции пяти аргументов;
– процедура TrueTablePrint(a) выводит таблицу истинности, полученную посредством функции TrueTable;
– процедура TrueTablePrint(a, f) выводит таблицу истинности, полученную посредством функции TrueTable. При f = 0 выводятся только строки, в которых значение функции равно False, при f = 1 – только строки, в которых оно равно True.

12. Процедура ReplaceLast(s, a, b) в строке *s* заменяет последнее вхождение подстроки *a* подстрокой *b*.

12.2. Примеры решения задач

Задача 12.1
Вычислить значение выражения $(3A5_{16}+73_9)\times 23_7-211_9$ и записать его в системе счисления по основанию 13.

```
uses School;

begin
  ((Dec('3A5', 16) + Dec('73', 9)) * Dec('23', 7) - Dec('211', 9))
      .ToString
      .ToBase(13)
      .Print
end.
```

```
7862
```

Алгоритм решения: переводим все числа в десятичную систему счисления, вычисляем значение выражения, переводим его в строку, переходим к системе счисления по указанному основанию и выводим результат.

Задача 12.2
Построить таблицу истинности для функции трех переменных $F(a,b,c) = a \rightarrow (b \vee \neg b \wedge \neg c)$

```
uses School;

begin
  var a := TrueTable((a, b, c) -> a.Imp(b or not b and not c));
  TrueTablePrint(a)
end.
```

```
a b c F
-------
0 0 0 1
0 0 1 1
0 1 0 1
0 1 1 1
1 0 0 1
1 0 1 0
1 1 0 1
1 1 1 1
```

Задача 12.3

Получить все четырехзначные натуральные числа, которые являются произведением пяти различных простых делителей. Числа упорядочить по возрастанию суммы их цифр, а при равной сумме – по возрастанию самих чисел.

```
uses School;

begin
  (1000..9999)
      .Where(n -> n.Factorize.Distinct.Count = 5)
      .OrderBy(n -> n.Digits.Sum)
      .ThenBy(n -> n)
      .Print
end.
```

```
2310 2730 4620 5610 6006 6510 7140 7410 9030 3570 4290 4830 5460
6090 6270 6630 8610 9240 6930 8190 3990 7590 7770 8580 9282 9570
9660 7854 7980 8970 9690 9870 8778
```

Приложение

В приложении…

Алгоритмы к ЕГЭ
Перенаправление ввода/вывода на файлы

П1. Алгоритмы к ЕГЭ

Приводятся возможные алгоритмические задачи для перечня требований к уровню подготовки выпускников, достижение которых проверяется на едином государственном экзамене по информатике и ИКТ.

§1. Минимум и максимум

Алгоритмы: нахождение минимума и максимума двух, трех, четырех данных чисел без использования массивов и циклов.

Задача П1-1
Найти максимум четырех целых чисел и минимум трех вещественных чисел без использования массивов и циклов.

```
##
var (a, b, c, d) := ReadInteger4;
var max := a;
if b > max then max := b;
if c > max then max := c;
if d > max then max := d;
Println('max =', max);
var(x, y, z) := ReadReal3;
var min := x;
if y < min then min := y;
if z < min then min := z;
Print('min =', min)
```

```
35 -24 47 0
max = 47
-5.23 2.65 0
min = -5.23
```

Можно также воспользоваться функциями Max и Min:

```
##
var (a, b, c, d) := ReadInteger4;
Println('max =', Max(a, b, c, d));
var(x, y, z) := ReadReal3;
Print('min =', Min(x, y, z))
```

Подобные задачи приведены также в I - 3.5.1 (здесь и далее римские цифры означают часть книги).

§2. Корни квадратного уравнения

Рассматриваем уравнение вида $ax^2+bx+c=0$ при $a\neq0$. Оно всегда имеет два или ноль действительных корней. Теория вопроса дается в подразделе I-3.5.5.

Задача П1-2
Найти корни квадратного уравнения $ax^2+bx+c=0$, если $a\neq0$. Коэффициенты a, b и c ввести с клавиатуры. Вывод осуществить с точностью 5 знаков после запятой.

```
##
var (a, b, c) := ReadReal3('Введите коэффициенты a, b, c:');
var D := b * b - 4 * a * c;
if D > 0 then
begin
  D := Sqrt(D);
  var (x1, x2) := ((-b - D) / (2 * a), (-b + D) / (2 * a));
  Write('x1 = ', x1:0:5, ',  x2 = ', x2:0:5)
end
else if D = 0 then
  Write('x = ', -b / (2 * a):0:5) // корня два, совпадающих
else
  Print('Действительных корней нет')
```

§3. Позиционные системы счисления

Алгоритмы: запись натурального числа в позиционной системе с основанием, меньшим или равным 10. Обработка и преобразование такой записи числа.

Чтобы перевести натуральное десятичное число *n* в систему счисления по основанию *k*, нужно последовательно делить его нацело на *k* и выписывать остатки в обратном порядке, пока на месте n не получим ноль. Для представления результата можно использовать строку, приписывая полученные остатки в символьном виде слева, либо стек, выгрузив затем его содержимое в качестве ответа. Не совсем понятно, почему речь об основаниях, не превышающих 10, ведь в информатике широко рассматривается и шестнадцатеричная система счисления. Возможно, эта проблема связана с тем, что в прошлых КИМ ЕГЭ не было задач на обработку символьных строк программным путем.

Потребуется знание материала раздела I-1.4, а также раздела III-8.2 для работы со строкой или раздела III-11.1, если захочется использовать стек.

Задача П1-3
Перевести десятичное число n (0 < n < 2 000 000 000) в систему счисления по основанию 8.

Решение без стека:

```
##
var (n, k, s) := (ReadInteger, 8, '');
while n > 0 do
begin
  s := n mod k + s;
  n := n div k
end;
s.Print
```

```
463453246
3347736076
```

Решение со стеком:

```
##
var (n, k, s) := (ReadInteger, 8, new Stack<integer>);
while n > 0 do
begin
  s.Push(n mod k);
  n := n div k
end;
s.Print('')
```

Не знаю, что выберете вы, но решение со строкой явно проще. Второе решение – явная демонстрация «А я умею работать со стеком!».

Есть и третье решение – использовать учебную библиотеку School (III-12.1.2, §5). В ней имеется готовая функция, делающая все необходимое.

```
uses School;

begin
  var (n, k) := (ReadInteger, 8);
  n.ToString.ToBase(k).Print
end.
```

Здесь надо только помнить, что метод ToBase принимает не число, а строку, поскольку он рассчитан на работу с числами любой длины.

§4. Сумма и произведение для последовательности

Алгоритмы: нахождение сумм и произведений элементов данной конечной числовой последовательности (или массива).

Пока что встречались задания, в которых сумма и произведение не превышали диапазона, отведенного для данных типа **integer**. Если есть опасение, что происходит переполнение, поменяйте тип переменных, отведенных для накопления суммы (произведения), на **int64** и сравните результаты. А еще лучше – используйте тип BigInteger, о котором сказано в разделе III-9.2.

Задача П1-4
Найти сумму и произведение n целочисленных элементов последовательности, каждый из которых по абсолютной величине не превышает 30000.

1. Нахождение суммы и произведения элементов последовательности в цикле. Вводимые элементы не храним. Начнем с типа **integer**.

```
##
var n := ReadInteger; // длина последовательности
var s := 0; // Сумма. Варианты: int64(0), BigInteger.Zero
var p := 1; // Произведение. int64(1), BigInteger.One
for var i := 0 to a.High do
begin
  var n := ReadInteger; // очередной элемент
  s += n;
  p *= n
end;
Print(s, p)
```

Рассмотрим, как будет работать программа при вводе различных данных.

```
12
223 -287 298 -20 79 336 -118 39 -6 276 -210 -185
425 2144768000
```

Произведение достаточно большое. Но верно ли оно? Используем тип BigInteger для s и p:

```
12
223 -287 298 -20 79 336 -118 39 -6 276 -210 -185
425 2997763435679314917888000
```

Оказывается, не зря усомнились. Значение произведения такое, что в типе **int64** не разместится.

2. Теперь рассмотрим решение задачи в функциональном стиле, несколько ее упростив: будем искать только произведение. Почему? Работая с последовательностью, элементы которой вводятся с клавиатуры, мы должны использовать однопроходный алгоритм (раздел II-6.1), а условие требует найти суммы и произведение, для чего потребуются два прохода. Полное решение с массивом рассмотрим позднее.

```
##
var n := ReadInteger; // длина последовательности
ReadSeqInteger(n).Product(m -> BigInteger(m)).Print
```

3. Если требуется работа с массивом, использование цикла может выглядеть так:

```
##
var n := ReadInteger; // длина последовательности
var a := ReadArrInteger(n); // элементы массива
var s := BigInteger.Zero; // Сумма. Варианты: 0, int64(0)
var p := BigInteger.One; // Произведение. 1, int64(1)
foreach var k in a do
begin
  s += k;
  p *= k
end;
Print(s, p)
```

4. Функциональный стиль позволяет сократить и такую запись.

```
##
var n := ReadInteger; // длина последовательности
var a := ReadArrInteger(n);
a.Sum.Print;
a.Product(m -> BigInteger(m)).Print
```

§5. Простые переборные задачи

Алгоритмы: использование цикла для решения простых переборных задач (поиск наименьшего простого делителя данного натурального числа, проверка числа на простоту и т.д.).

Задача П1-5
Найти наименьший простой делитель заданного натурального числа n. Значение n не превышает 2 миллиарда.

В соответствии с условием заключаем, что достаточно использовать тип **integer**. Ряд наименьших простых делителей – простые числа 2, 3, 5, … Алгоритм решения состоит в последовательном нахождении простых чисел либо до получения нулевого остатка от деления *n* на очередное простое число *k*, либо до выполнения условия $k^2>n$. В первом случае значение *k* является ответом, во втором – считается, что такой делитель отсутствует (само число *n* является простым). Для ускорения решения после проверки для *k = 3* можно перебирать только нечетные числа. Разновидность цикла – лучше **while** или **repeat** … **until**.

Наличие библиотеки School может сильно упростить решение задачи. В нее включен ряд средств для работы с простыми числами, использование которых сильно сократит программный код. Посмотрите одно из возможных решений. Расширение *n.Factorize* возвращает упорядоченный по возрастанию список всех простых делителей числа *n*, исключая 1 и само *n*. Расширение .FirstOrDefault вернет в качестве *k* первый элемент этого списка или ноль, если список окажется пустым.

```
uses School;

begin
  var n := ReadInteger;
  var k := n.Factorize.FirstOrDefault;
  if (k > 1) and (k <> n) then
    Print('Наименьший простой делитель равен', k)
  else
    Print('Нет простых делителей')
end.
```

А теперь рассмотрите решение на основе цикла, которое не использует возможностей библиотеки School и является типичной переборной задачей:

```
begin
  var n := ReadInteger;
  if n > 1 then
    if (n mod 2 = 0) and (n > 2) then
    begin
      Print('Наименьший простой делитель равен 2');
      exit
    end
    else
    begin
      var k := 3;
      while k * k <= n do
        if n mod k = 0 then
        begin
          Print('Наименьший простой делитель равен', k);
          exit
        end
        else
          k += 2
    end;
  Print('Нет простых делителей')
end.
```

Что дает нам право считать найденный делитель простым числом? Он простой, ведь в противном случае его собственным делителем было хотя бы одно из ранее просмотренных чисел.

Проверка числа на простоту выполняется аналогично, но в этом случае не должен быть найден ни один из делителей. Если вдруг нашли – число не является простым и можно прерывать цикл.

§6. Заполнение элементов массивов

Алгоритмы: заполнение элементов одномерного и двумерного массивов по заданным правилам.

PascalABC.NET предлагает различные способы заполнения массивов. Можно воспользоваться генератором массива и присвоить результат имеющемуся массиву. Можно использовать традиционные циклы. Двумерный массив можно сформировать из одномерных массивов.

Генераторы одномерных массивов рассмотрены в разделе II-6.10, двумерных – в разделе II-7.2. О преобразовании матрицы в массив массивов и обратном преобразовании написано в параграфе II-7.4 §2 и подразделе II-7.6.2.

Ниже приводится пример решения двух простейших задач на заполнение массивов при помощи цикла **for**.

Задача П1-6
Заполнить целочисленный одномерный массив длины n ($n \leqslant 100$) кубами порядковых номеров его элементов.

```
##
var n := ReadInteger;
var a := new integer[n];
for var i := 0 to a.High do
  a[i] := (i + 1) * Sqr(i + 1);
a.Print
```

Задача П1-7
Заполнить целочисленный двумерный массив размером m×n ($m, n \leqslant 100$), вычислив значения элементов по формуле $a_{ij} = 100i+j$, $i = 1..m$, $j = 1..n$.

```
##
var (m, n) := ReadInteger2;
var a := new integer[m, n];
for var i := 1 to m do
  for var j := 1 to n do
    a[i - 1, j - 1] := 100 * i + j;
a.Print(6)
```

§7. Операции с элементами массива

Алгоритмы: Линейный поиск элемента. Вставка и удаление элементов в массиве. Перестановка элементов данного массива в обратном порядке. Суммирование элементов массива. Проверка соответствия элементов массива некоторому условию.

Задача П1-8
Дан целочисленный массив размера n ($n \leqslant 10000$), заполненный случайными числами из отрезка [-100;100]. Найти в нем номер первого от начала элемента со значением, равным k.

Поскольку в условии не сказано, что массив упорядочен, нужно вести последовательный перебор элементов до тех пор, пока не будет найден элемент с искомым значением или не будут просмотрены все элементы.

```
##
var (n, k) := ReadInteger2;
var a := ArrRandom(n, -100, 100);
a.Println;
for var i := 0 to a.High do
  if a[i] = k then
  begin
    Print('Номер элемента равен', i + 1);
    exit
  end;
Print('Элемент с таким значением отсутствует')
```

Эта задача может быть решена и без явного перебора.

```
##
var (n, k) := ReadInteger2;
var a := ArrRandom(n, -100, 100);
a.Println;
var i := a.IndexOf(k);
if i >= 0 then
  Print('Номер элемента равен', i + 1)
else
  Print('Элемент с таким значением отсутствует')
```

Задача П1-9

Дан целочисленный массив размера n ($n \leqslant 10000$), заполненный случайными числами из отрезка [-100;100]. Вставить после элемента с минимальным значением элементы со значением -1, 0 и 1. Затем удалить все элементы, следующие после элемента с максимальным значением.

```
begin
  var n := ReadInteger;
  var a := ArrRandom(n, -100, 100);
  a.Println;
  var imin := 0;
  for var i := 1 to a.High do
    if a[i] < a[imin] then
      imin := i;
  SetLength(a, n + 3);
  for var i := a.High downto imin + 3 do
    a[i] := a[i - 3];
  (a[imin + 1], a[imin + 2], a[imin + 3]) := (-1, 0, 1);
  var imax := 0;
  for var i := 1 to a.High do
    if a[i] > a[imax] then
      imax := i;
  SetLength(a, imax + 1);
  a.Print
end.
```

Срезы дают короткое и наглядное решение.

```
##
var n := ReadInteger;
var a := ArrRandom(n, -100, 100);
a.Println;
var imin := a.IndexMin;
a := a[:imin + 1] + |-1, 0, 1| + a?[imin + 1:];
a := a[:a.IndexMax + 1];
a.Print
```

Задача П1-10

Дан целочисленный массив размера n ($n \leqslant 10000$, четное), заполненный случайными числами из отрезка [-100;100]. Переставить в обратном порядке элементы второй половины массива.

Можно сделать перестановку «на месте», обменивая местами пары элементов, можно создать новый массив и переписать туда элементы в нужном порядке. Рассмотрим вариант с перестановкой «на месте».

Пусть n = 10, тогда перестановке подлежат элементы с индексами от 5 до 9 (в динамическом массиве индексы отсчитываются от нуля). Менять местами нужно пары 5 – 9 и 6 – 8. Элемент с индексом 7 останется на месте. Для n = 20 перестановка ведется на интервале индексов от 10 до 19, а пары будут в этом случае такими: 10 – 19, 11 – 18, 12 – 17, 13 – 16, 14 – 15. На первом шаге определяем диапазон индексов для перестановки пар: [n **div** 2; n – 1]. Количество пар, участвующих в перестановке, равно n **div** 4.

```
##
var n := ReadInteger;
var a := ArrRandom(n, -100, 100);
a.Println;
var k := n div 2;
for var i := 0 to n div 4 - 1 do
  Swap(a[k + i], a[n - 1 - i]);
a.Print
```

Традиционно, более короткое решение без циклов. Срезы и тут решают проблему.

```
##
var n := ReadInteger;
var a := ArrRandom(n, -100, 100);
a.Println;
a[n div 2:] := a[n - 1:n div 2 - 1:-1];
a.Print
```

Задача П1-11

Дан целочисленный массив размера n ($n \leqslant 10000$), заполненный случайными числами из отрезка [-100;100]. Найти сумму его элементов, имеющих четные значения.

Алгоритмы вычисления сумм и произведений приведены в Приложении I-П4. При работе с массивами элементы выбираются посредством их индексов. В суммировании участвуют лишь те элементы, для которых выполняется заданное условие (четность).

```
##
var n := ReadInteger;
var a := ArrRandom(n, -100, 100);
a.Println;
var s := 0;
for var i := 0 to a.High do
  if a[i] mod 2 = 0 then
    s += a[i];
Print(s)
```

Поскольку индекс массива нужен только для выборки элемента, удобнее использовать цикл **foreach**. Также можно воспользоваться расширением .IsEven для проверки на четность.

```
##
var n := ReadInteger;
var a := ArrRandom(n, -100, 100);
a.Println;
var s := 0;
foreach var b in a do
  if b.IsEven then
    s += b;
Print(s)
```

Решение в функциональном стиле намного лаконичнее:

```
##
var n := ReadInteger;
var a := ArrRandom(n, -100, 100);
a.Println;
a.Where(b -> b.IsEven).Sum.Print
```

§8. Второй экстремум в массиве

Алгоритмы: нахождение второго по величине (второго максимального или второго минимального) значения в данном массиве за однократный просмотр массива.

Здесь нужно запоминать и поддерживать два экстремума (минимума или максимума) – первого и второго по величине. При проверке очередного элемента возможны несколько случаев.

Пусть требуется найти второе максимальное по величине значение в массиве. Введем переменную для максимального значения с именем *max1* и переменную для второго по величине максимального значе-

ния с именем *max2*. Положим значения *max1* и *max2* равными значению первого элемента массива. Для любого последующего значения элемента массива a_i выполняем следующий алгоритм:

Если $a_i > max1$, запоминаем *max1* в *max2* и a_i в *max1*.
В противном случае, если $a_i > max2$, запоминаем a_i в *max2*.
После просмотра всего массива искомое значение будет находиться в *max2*.

Задача П1-12
Дан целочисленный массив размера n ($n \leqslant 10000$), заполненный случайными числами из отрезка [-100;100]. Найти второе по величине максимальное значение в массиве за его однократный просмотр.

```
##
var n := ReadInteger;
var a := ArrRandom(n, -100, 100);
a.Println;
var (max1, max2) := (a[0], a[1]);
if max2 > max1 then
  Swap(max1, max2);
for var i := 2 to a.High do
  if a[i] > max1 then
    (max1, max2) := (a[i], max1)
  else if a[i] > max2 then
    max2 := a[i];
Print(max2)
```

Решение в функциональном стиле здесь использовать не следует, поскольку в нем количество просмотров массива в явном виде не присутствует. При отсутствии контроля за выполнением условия однократного просмотра проще всего массив отсортировать по убыванию в последовательность и взять второй от начала элемент.

```
##
var n := ReadInteger;
var a := ArrRandom(n, -100, 100);
a.Println;
a.OrderDescending.Skip(1).Take(1).Print
```

§9. Подсчет числа экстремумов в массиве

Алгоритмы: нахождение минимального (максимального) значения в данном массиве и количества элементов, равных ему, за однократный просмотр массива.

Потребуется завести две переменные – одну для хранения экстремума и вторую для счетчика количества значений, равных экстремуму. Если значение экстремума переопределяется, в счетчик помещается единица. В случае, когда очередное значение равно экстремуму, в счетчик добавляется единица.

Задача П1-13
Дан целочисленный массив размера n ($n \leqslant 10000$), заполненный случайными числами из отрезка [-20;20]. Найти значение минимального элемента в массиве и количество элементов, равных ему за однократный просмотр массива.

```
##
var n := ReadInteger;
var a := ArrRandom(n, -20, 20);
a.Println;
var (min, k) := (a[0], 1);
for var i := 1 to a.High do
  if a[i] < min then
    (min, k) := (a[i], 1)
  else if a[i] = min then
    k += 1;
Print(min, k)
```

Решение в функциональном стиле здесь использовать не следует, поскольку в нем количество просмотров массива в явном виде не присутствует. При отсутствии контроля за выполнением условия однократного просмотра можно отсортировать массив по возрастанию в последовательность, сгруппировать элементы по значениям, сформировать минимум и счетчик, а затем взять первый элемент полученной последовательности. В целом код получается не очень простым для начинающих.

```
##
var n := ReadInteger;
var a := ArrRandom(n, -20, 20);
a.Println;
var (min, k) := a.Order.GroupBy(b -> b)
   .Select(t -> (t.Key, t.Count)).First;
Print(min, k)
```

§10. Операции с элементами, отобранными по условию

Алгоритмы: операции с элементами массива, отобранными по некоторому условию (например, нахождение минимального четного элемента в массиве, нахождение количества и суммы всех четных элементов в массиве).

Такие задачи решаются аналогично рассмотренным выше (например, П1-11). В теле цикла добавляется условие для отбора.

§11. Сортировка массива

Обычно в школе рассматривается алгоритм пузырьковой (обменной) сортировки, как наиболее простой в реализации. Действительно, реализация этого алгоритма занимает три строки кода и легко запоминается. Этот алгоритм школьник знать обязан. Но знать – не означает всегда и везде писать такой код. В PascalABC.NET имеются процедуры и функции, реализующие сортировку и проще пользоваться ими.

Задача П1-14
Дан целочисленный массив размера n ($n \leqslant 10000$), заполненный случайными числами из отрезка [-100;100]. Отсортировать массив по возрастанию.

```
##
var n := ReadInteger;
var a := ArrRandom(n, -100, 100);
a.Println;
for var i := n - 2 downto 0 do
  for var j := 0 to i do
    if a[j] > a[j + 1] then
      Swap(a[j], a[j + 1]);
a.Print
```

Для получения сортировки по убыванию достаточно заменить операцию отношения > на <.

Используя средства PascalABC.NET можно написать и короче:

```
##
var n := ReadInteger;
var a := ArrRandom(n, -100, 100);
a.Println;
a.Sort;
a.Print
```

Сортировка массива по убыванию обеспечивается использованием параметра, например, Sort((u, v) -> v – u).

§12. Слияние массивов

Алгоритм: слияние двух упорядоченных массивов в один без использования сортировки.

Существует достаточное количество алгоритмов слияния упорядоченных массивов. Используем самый короткий из них – слияние с барьером. Он рассмотрен в подразделе II-6.26.4.

Задача П1-15
Даны два целочисленных массива размера n и m соответственно (n, m $\leqslant$ 10000), заполненные упорядоченными по неубыванию случайными числами из отрезка [-100;100]. Выполнить слияние упорядоченных массивов не используя сортировку.

```
begin
  // подготовка исходных данных и их вывод
  var (n, m) := ReadInteger2;
  var a := SeqRandom(n, -100, 100).Order.ToArray;
  a.Println;
  var b := SeqRandom(m, -100, 100).Order.ToArray;
  b.Println;
  // добавляем барьеры
  SetLength(a, n + 1);
  a[n] := 200;
  SetLength(b, m + 1);
  b[m] := 200;
```

```
    // алгоритм слияния
    var c := new integer[n + m]; // выходной массив
    var (ia, ib) := (0, 0); // индексы для прохода по массивам
    var ic := 0; // индекс в результирующем массиве
    while (a[ia] <> 200) or (b[ib] <> 200) do
      if a[ia] < b[ib] then
      begin
        c[ic] := a[ia];
        ia += 1;
        ic += 1
      end
      else
      begin
        c[ic] := b[ib];
        ib += 1;
        ic += 1
      end;
    c.Print
end.
```

Конечно, на практике работать со списками List предпочтительнее, но данный пример иллюстрирует работу именно с массивами в соответствии с требованиями к уровню подготовки выпускников. Работа со списками List показана в примерах подраздела II-6.26.4.

§13. Обработка отдельных символов в строке

Алгоритмы: Обработка отдельных символов данной строки. Подсчет частоты появления символа в строке.

Работая с символами в PascalABC.NET нужно определиться, в какой кодовой таблице планируется работать. Это важно для символов кириллицы, а также других национальных алфавитов, отличных от латиницы, поскольку большая часть прочих символов всегда имеет коды от 0 до 127_{10}.

Получить код *n* некоторого символа, хранящегося в переменной *c* типа **char**, можно посредством вызова функции *Ord(c)* или эквивалентной ей *OrdUnicode(c)*. Если нужен код символа в национальной таблице Windows (cp1251 для России), используется вызов *OrdAnsi(c)*. Обратное преобразование выполняют соответственно функции Chr(n), ChrUnicode(n), ChrAnsi(n).

Код символа, возвращаемый функциями *Ord( )* и *OrdUnicode( )*, имеет тип **word** (двухбайтное целое, принадлежащее отрезку 0..65535). Функция *OrdAnsi( )* возвращает значение типа **byte** (однобайтное целое, принадлежащее отрезку 0..255). Над этим можно особенно не задумываться, поскольку значения типа **byte** и **word** можно свободно присваивать переменным типа **integer**, а также объединять их в выражениях с другими целочисленными типами.

Обработка символов часто связана с выделением из символьных строк отдельного символа. Также, она появляется при работе с последовательностями (массивами, списками и т.д.) символов. Как для строки *s*, так и для массива *a* обращение к символу с порядковым номером *k* (отсчет от единицы) записывается в виде *s[k]* или *a[k]*.

Задача П1-16
Во введенной с клавиатуры строке подсчитать суммарное количество прописных и строчных букв «ш».

Для тех, кто «забыл»: прописные (заглавные, большие) буквы находятся на верхнем регистре клавиатуры, а строчные (маленькие) – на нижнем. Можно отдельно проверять символ на принадлежность обоим регистрам, либо использовать функцию преобразования регистра символа и проверять его на одном регистре.

```
##
var s := ReadString;
var k := 0;
for var i := 1 to s.Length do
  if (s[i] = 'Ш') or (s[i] = 'ш') then
    k += 1;
Print(k)
```

```
Шелковый тревожный шорох в пурпурных портьерах, шторах
3
```

Короткое решение в функциональном стиле:

```
##
var s := ReadString;
s.Count(c -> c.ToLower = 'ш').Print
```

Суперкороткое решение в одну строку:

```
## ReadString.Count(c -> c.ToLower = 'ш').Print
```

Задача П1-17
В заданной строке заменить каждое вхождение буквы «ы» на букву «и».

```
##
var s := 'Жы-Шы пишы с буквой ы';
s.Println;
for var i := 1 to s.Length do
  if s[i] = 'ы' then
    s[i] := 'и';
s.Print
```

Решение с использованием функции замены символов:

```
##
var s := 'Жы-Шы пишы с буквой ы';
s.Println;
s := s.Replace('ы', 'и');
s.Print
```

Задача П1-18
В заданной строке найти и вывести символы, у которых код превышает 127 (национальные символы).

В любой кодировке символ с кодом, меньшим 128 (интернациональный символ), имеет постоянный код. Исходя из соображений краткости записи, код символа будем получать посредством функции *Ord( )*.

```
##
var s := 'My flowers are beautiful!';
for var i := 1 to s.Length do
  if Ord(s[i]) > 127 then
    Print(s[i])
```

```
o a a
```

Решение в функциональном стиле:

```
##
var s := 'My flowers are beautiful!';
s.Where(c -> Ord(c) > 127).Print(' ')
```

Кто не понял, почему в расширении .Print в качестве параметра указан пробел, отправляется читать подраздел III-8.2.4.

§14. Обработка подстрок в строке

Алгоритмы: Работа с подстроками данной строки с разбиением на слова по пробельным символам. Поиск подстроки внутри данной строки, замена найденной подстроки на другую строку.

При разбиении строки на слова нужно отличать случай, когда слова разделены ровно одним пробелом и когда пробелов между словами может быть больше, чем один. Кроме того, пробелы могут быть перед первым словом в строке и после последнего слова. В PascalABC.NET имеются мощные и удобные средства для разбиения строки по словам (подраздел III-8.3.10), поэтому будут рассмотрены только базовые алгоритмы для подобного разбиения.

Задача П1-19
Дана строка, содержащая слова, разделенные ровно одним пробелом. Под словом понимается последовательность любых символов, отличных от пробела. Вывести все слова, начиная каждое с новой строки.

Разбиение на слова связано с поиском пробельного символа. Пока он не найден – мы находимся внутри очередного слова. Как нашли – выводим это слово и переходим к обработке очередного слова, следующего за пробелом. Код программы будет короче, если в конце строки обеспечить наличие пробела.

```
##
var Строка := 'Наша тестовая строка содержит шесть слов';
Строка += ' ';
var Слово := '';
for var Позиция := 1 to Строка.Length do
  if Строка[Позиция] = ' ' then
  begin
    Println(Слово);
    Слово := ''
  end
  else
    Слово += Строка[Позиция]
```

Можно также написать вариант программы с циклом **foreach**, он будет немного нагляднее.

```
##
var Строка := 'Наша тестовая строка содержит шесть слов';
Строка += ' ';
var Слово := '';
foreach var Символ in Строка do
  if Символ = ' ' then
  begin
    Println(Слово);
    Слово := ''
  end
  else
    Слово += Символ
```

Задача П1-20
Дана строка, содержащая слова, разделенные ровно одним пробелом. Под словом понимается последовательность любых символов, отличных от пробела. Найти количество слов в строке. Гарантируется, что строка не пустая.

Эта задача существенно проще предыдущей. Если слова разделены ровно одним пробелом, то количество слов в такой строке превышает количество пробелов на единицу. Следовательно, достаточно подсчитать лишь количество пробелов.

```
##
var Строка := 'Наша тестовая строка содержит шесть слов';
var Счетчик := 0;
foreach var Символ in Строка do
  if Символ = ' ' then
    Счетчик += 1;
if Счетчик = 0 then
  Print(1)
else if Строка[1] = ' ' then
  Print(0)
else
  Print(Счетчик + 1)
```

Проверка условия в конце связана с тем, что строка может содержать только одно слово, только один пробел, а также более одного слова.

Задача П1-21
Дана строка, содержащая слова, разделенные ровно одним пробелом. Под словом понимается последовательность любых символов, отличных от пробела. Получить список слов, вывести их, начиная каждое с новой строки, а также найти первое из слов с максимальной длиной и его номер по порядку.

Нам подойдет алгоритм решения задачи П1-19, но вместо вывода будем помещать слова в список List.

```
begin
  var Строка := 'Наша тестовая строка содержит шесть слов';
  var Слово := '';
  var Список := new List<string>;
  foreach var Символ in Строка do
    if Символ = ' ' then
    begin
      Список.Add(Слово);
      Слово := ''
    end
    else
    Слово += Символ;
  if Слово <> '' then
    Список.Add(Слово);
  Список.PrintLines;
  var ИндексДлинного := 0;
  for var Индекс := 0 to Список.Count - 1 do
    if Список[Индекс].Length > Список[ИндексДлинного].Length then
      ИндексДлинного := Индекс;
  Слово := Список[ИндексДлинного];
  $'Найдено слово "{Слово}" номер {ИндексДлинного + 1}'.Print
end.
```

Поиск и замена подстрок средствами PascalABC.NET подробно рассматривались в главе III-8. Самый примитивный алгоритм поиска подстроки *ss* в строке *s* состоит в поиске до выполнения условия *s[i] = ss[1]* с последующими проверками *s[i+1] = ss[2]*, *s[i+2] = ss[2]*, ... *s[i+k] = ss[k]*. Если для очередной пары символов *s[i+j] ≠ ss[j]*, алгоритм поиска повторяется с позиции *i+j+1*. Поиск останавливается, если непросмотренная часть строки короче длины искомой подстроки.

Для замены подстрок требуется знать позиции, в которых начинаются подстроки в исходной строке. Если выполняется более одной

замены, нужно проводить их от конца, поскольку в противном случае после замены возможна сдвижка позиций символов.

В любом случае, базовые алгоритмы замены достаточно длинны и требуют анализа многих ситуаций, поэтому использовать их в повседневных задачах (а также на экзаменах или олимпиадах) не имеет смысла из-за потерь времени на кодирование и отладку.

П2. ЕГЭ: перенаправление ввода/вывода

В демоверсии «компьютерного ЕГЭ», представленной ФИПИ в конце лета 2020 года, имеются задачи в которых исходные данные нужно брать из текстового файла. Для решения таких задач можно использовать средства PascalABC.NET, предназначенные именно для работы с файлами, но эта тема в книге не рассмотрена. Причина проста: среди «возможных алгоритмических задач для перечня требований к уровню подготовки выпускников, достижение которых проверяется на едином государственном экзамене по информатике и ИКТ» нет задач, действительно требующих серьезно работать с файлами.

В PascalABC.NET ввод производится через файл, определенный переменной *input*, по умолчанию связывающей его с клавиатурой. Аналогично, вывод через переменную *output* связан с монитором. Для того, чтобы вместо клавиатуры принимать данные из файла с именем *МойФайл.txt*, достаточно указать оператор

```
Assign(input, 'МойФайл.txt');
```

Если файл не находится в той же папке, что и программа, нужно указывать полный путь, например 'C:\Экзамен\ЕГЭ\МойФайл.txt'.

Вывод вместо монитора можно также направить в файл, указав

```
Assign(output, 'МойВывод.txt');
```

Вызов процедуры Assign нужно выполнять до первого обращения ко вводу или выводу.