МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

ОТЧЁТ

по Курсовой работе по дисциплине

Функциональное и логическое программирование

Тема: «Создание библиотеки для работы со строками»

РУКОВОДИТЕЛЬ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Суркова А.С.

СТУДЕНТ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Уткин А.О

21-ПО

Работа защищена «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_

С оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Нижний Новгород 2024

Оглавление

[Введение 2](#_Toc167319937)

[Постановка задачи 3](#_Toc167319938)

[Обзор языка Scala 3](#_Toc167319939)

[Структура проекта и описание файлов 6](#_Toc167319940)

[Особенности работы с массивами в функциональных языках программирования 7](#_Toc167319941)

[Сортировка 9](#_Toc167319942)

[Лексикографическое сравнение 10](#_Toc167319943)

[Разделение строк на слова 12](#_Toc167319944)

[Sliding 12](#_Toc167319945)

[Sublist 13](#_Toc167319946)

[Проблемы возникшие при разработке программы 14](#_Toc167319947)

[Тестовые примеры 17](#_Toc167319948)

[Вывод 18](#_Toc167319949)

# Введение

Строковый тип данных представляет собой последовательность символов, которые могут содержать текст, числа или другие данные, записанные в текстовом формате. Строки являются одним из основных типов данных в большинстве языков программирования и используются для хранения и обработки текстовой информации, такой как имена, адреса, текстовые сообщения и многое другое. Строки могут быть созданы с помощью различных символов обозначения, таких как одинарные или двойные кавычки, и могут быть преобразованы в другие типы данных, такие как числа или булевы значения, в зависимости от контекста использования. Обработка строк включает в себя такие операции, как поиск подстроки, разбиение на подстроки, преобразование регистра и замена подстрок.

Языки Erlang, Haskell, Пролог используют для строкового типа список символов. Этот метод делает язык более «теоретически элегантным» за счёт соблюдения ортогональности в системе типов, но приносит существенные потери быстродействия.

В языке программирования Scala строковый тип данных представлен классом String, который представляет собой последовательность символов Unicode. Строки в Scala являются неизменяемыми (immutable), что означает, что после создания строки ее содержимое нельзя изменить.

Scala предоставляет различные операции для работы со строками, такие как конкатенация (с помощью оператора +), сравнение строк, извлечение подстроки, форматирование и многое другое. Также в Scala доступны многострочные строки (raw strings), которые удобно использовать для работы с многострочными текстовыми данными.

Строки в Scala являются важным и широко используемым типом данных, который играет значительную роль при работе с текстовой информацией в приложениях на этом языке программирования.

# Постановка задачи

Целью курсовой работы было создание библиотеки для работы с строками. В библиотеке должны содержаться функции, соответствующие принципам парадигмы функционального программирования то есть функции должны быть чистыми. Функции должны являться аналогами стандартных функций для работы со строками scala, таких как slider,substring,count и тд. Также можно реализовать функции для работы с массивами типа map,sum,drop итд, на базе которых и построить функции для работ со строками.

# Обзор языка Scala



Рис 2.1 «Логотип Scala»

**Scala** — мультипарадигмальный язык программирования, спроектированный кратким и типобезопасным для простого и быстрого создания компонентного программного обеспечения, сочетающий возможности функционального и объектно-ориентированного программирования.

Первые версии языка созданы в 2003 году коллективом лаборатории методов программирования Федеральной политехнической школы Лозанны под руководством Мартина Одерски, язык реализован для платформ Java и JavaScript. По мнению Джеймса Стрэчена, создателя языка программирования Groovy, Scala может стать преемником языка Java.

Язык был создан в 2001—2004 годах в Лаборатории методов программирования EPFL. Он стал результатом исследований, направленных на разработку улучшенной языковой поддержки компонентного программного обеспечения. За основу при разработке языка были взяты две идеи:

1. Язык программирования компонентного ПО должен быть масштабируемым в том смысле, что должна быть возможность с помощью одних и тех же концепций описать как маленькие, так и большие части. Поэтому внимание было сконцентрировано на механизмах абстракции, композиции и декомпозиции вместо введения большого количества примитивов, которые могут быть полезными только на каком-то одном уровне масштабирования.
2. Масштабируемая поддержка компонентов может быть предоставлена языком программирования, унифицирующим и обобщающим объектно-ориентированное и функциональное программирование. Некоторые из основных технических новшеств Scala — это концепции, представляющие собой сплав этих парадигм программирования. В статически типизированных языках, к которым относится Scala, эти парадигмы до сих пор были почти полностью разделены.

Язык был выпущен для общего пользования на платформе JVM в январе 2004 года и на платформе .NET в июне 2004 года, в 2016 году создан LLVM-компилятор (Scala Native).

Преимущества Scala:

* Функциональное и объектно-ориентированное программирование: Scala позволяет писать как функциональный, так и объектно-ориентированный код, что дает широкие возможности при проектировании приложений.
* Статическая типизация: Scala использует статическую типизацию, что позволяет выявлять ошибки на ранних стадиях разработки и повышает надежность программ.
* Мощная система типов: Scala обладает продвинутой системой типов, включая поддержку генериков, высокоуровневые типы и расширенную инференцию типов.
* Интероперабельность с Java: Scala обладает отличной совместимостью с Java, что позволяет использовать библиотеки Java и взаимодействовать с Java-кодом без проблем.
* Расширяемость и масштабируемость: Scala поддерживает функциональность для создания расширяемого и масштабируемого программного обеспечения.

Недостатки Scala:

* Крутой порог вхождения: Из-за сложности языка и продвинутых концепций, Scala может быть трудным для новичков в программировании.
* Долгое время компиляции: Из-за сложности типов и функциональных конструкций, компиляция программ на Scala может занимать больше времени, чем на других языках.
* Менее распространенный язык: Scala не так широко распространен как Java, Python или JavaScript, что может затруднить поиск ресурсов и специалистов для работы.
* Сложность отладки и профилирования: Некоторые разработчики могут столкнуться с трудностями в отладке и профилировании Scala-программ из-за сложных аспектов языка.

Хотя Scala имеет ряд преимуществ, включая мощную систему типов и поддержку функционального программирования, ее сложность и некоторые недостатки могут создать некоторые вызовы для разработчиков.

# Структура проекта и описание файлов

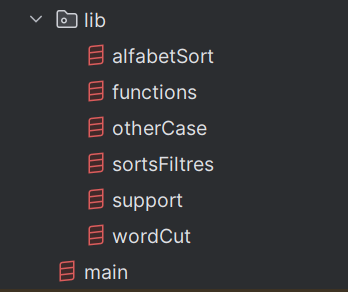


Рис 3.1 «Файлы проекта»

Проект содержит папку lib с 6 scala файлами и файл main в котором содержатся тестовые примеры.

Файл **support** – содержит вспомогательные функции для работы со списками, на базе функций из этого файла строятся многие другие функции

**sortsFiltres**- файл содержит реализацию метода mySort- аналога стандартной функции sort, quickSort- реализация популярного сортирующего алгоритма, sortBy-функция сортирующая массив элементов к которым применен некоторая функция

**alfabetSort**- содержит функции реализующие алфавитную сортировку слов

**otherCase**-содержит функции меняющие строчные буквы на заглавные и наоборот

**wordCut**- содержит функции делящие строку на слова и обратно

**functions**- непосредственно функции для работ со строками

Взаимосвязь функций:

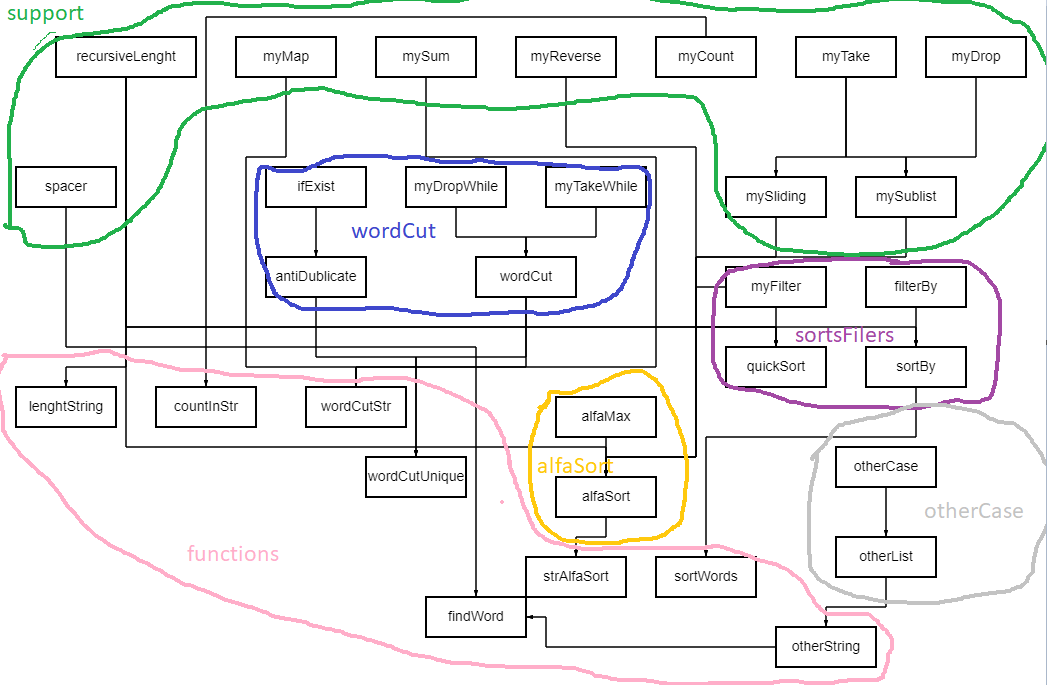


Рис 3.2 «Связь функций проекта»

# Особенности работы с массивами в функциональных языках программирования

Список — это структура данных, представляющая собой упорядоченный набор элементов. Каждый элемент списка имеет свой индекс, который указывает на его позицию в списке. Список может содержать элементы любого типа данных, например целые числа, строки, другие списки и т. д.

Основные операции, которые можно выполнять с списком, включают добавление элемента в список, удаление элемента из списка, доступ к элементу по индексу, изменение элемента по индексу, сортировку элементов списка и т. д.

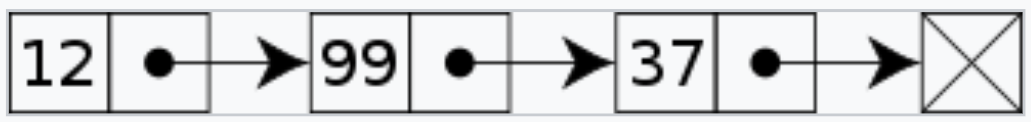


Рис 4.1 «Список наглядно»

В функциональных языках программирования, таких как Haskell или Scala, функции `head` и `tail` используются для работы с списками. Функция `head` возвращает первый элемент списка, а функция `tail` возвращает все элементы списка, кроме первого. Эти функции часто используются для разделения списка на голову и хвост, что удобно при рекурсивной обработке данных.

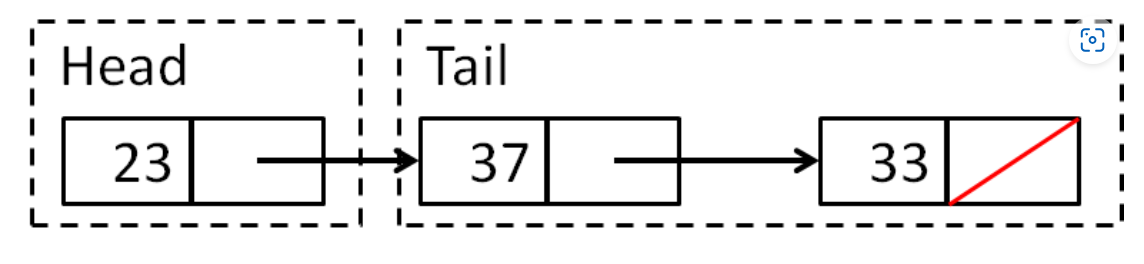


Рис 4.2 «Head Tail наглядно»

В Scala для конкатенации списков можно использовать оператор `++` или метод `:::`

Рекурсия в Scala - это процесс, при котором функция вызывает саму себя. Использование рекурсии в функциональных языках очень удобно ведь : функция f является чистой если выражение f(x) является ссылочно прозрачным для всех ссылочно прозрачных x, а ссылочная прозрачность — свойство, при котором замена выражения на вычисленный результат этого выражения не изменяет желаемых свойств программы. То есть применение функций или даже рекурсивных вызовов в выражениях поддерживается даже на уровне идеологии парадигмы.

Пример использования:

def recursiveLength(lst: List[\_]):Int= {  
 lst match {  
 case Nil => 0  
 case head :: tail => 1+recursiveLength(lst.tail)  
 }  
}

Визуализация

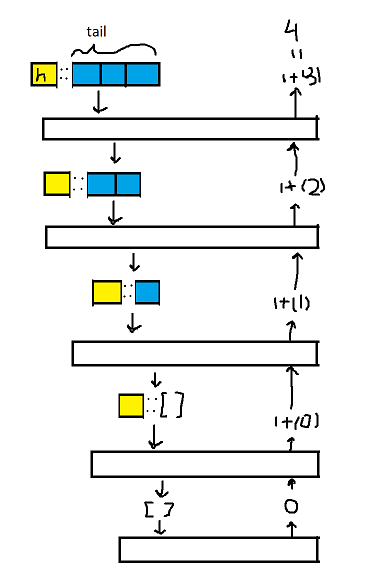


Рис 4.3 «Рекурсивная длинна строки наглядно»

# Сортировка

**Быстрая сортировка**, **сортировка Хоара** (англ. *quicksort*), часто называемая ***qsort*** (по имени в стандартной библиотеке языка Си) — алгоритм сортировки, разработанный английским информатиком Тони Хоаром во время его работы в МГУ в 1960 году.

Один из самых быстрых известных универсальных алгоритмов сортировки массивов: в среднем обменов при упорядочении элементов; из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

Quicksort - это один из самых быстрых и эффективных алгоритмов сортировки, который основывается на принципе "разделяй и властвуй". Он разбивает массив на подмассивы, сортирует их отдельно и объединяет в конечный отсортированный массив. Quicksort обладает сложностью O(n log n) в лучшем случае и O(n^2) в худшем случае, но зачастую работает быстрее других алгоритмов сортировки, таких как сортировка пузырьком или вставками.

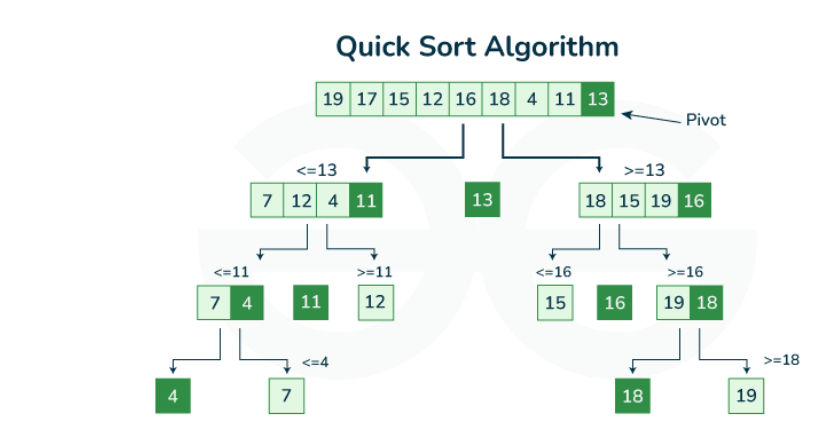


Рис 5.1 «quicksort наглядно»

Программа на c

void quickSort(int arr[], int low, int high)

{

    if (low < high) {

        // call Partition function to find Partition Index

        int partitionIndex = partition(arr, low, high);

        // Recursively call quickSort() for left and right

        // half based on partition Index

        quickSort(arr, low, partitionIndex - 1);

        quickSort(arr, partitionIndex + 1, high);

    }

}

Моя реализация

def myQuickSort(lst: List[Int]): List[Int] = {  
 if (lst==List.*empty*) lst  
 else {  
 val pivot = lst(recursiveLength(lst) / 2)  
 myQuickSort(myFilter(lst,(\_ < pivot)) ++  
 myFilter(lst,(\_ == pivot)) ++  
 myQuickSort(myFilter(lst,(\_ > pivot))))  
 }  
}

# Лексикографическое сравнение

Лексикографическое сравнение - это способ сравнения строк, чисел или других объектов на основе лексикографического порядка, который определяет порядок следования элементов в словаре.

При лексикографическом сравнении два объекта сначала сравниваются по первому символу или цифре: если они не равны, то возвращается результат этого сравнения; если они равны, то сравниваются следующие символы или цифры и так далее до тех пор, пока получится результат.

void compareStrings(char\* s1, char\* s2)

{

    int i;

    // comparing each character

    for (i = 0; s1[i] != '\0' || s2[i] != '\0'; i++) {

        if (s1[i] > s2[i]) {

            printf("String 1 is lexicographically greater "

                   "than string 2");

            return;

        }

        else if (s2[i] > s1[i]) {

            printf("String 2 is lexicographically greater "

                   "than string 1");

            return;

        }

    }

    // comparing length of two strings

    if (s1[i] != '\0') {

        printf("String 1 is lexicographically greater than "

               "string 2");

    }

    else if (s2[i] != '\0') {

        printf("String 2 is lexicographically greater than "

               "string 1");

    }

    else {

        printf("Both strings are lexicographically equal");

    }

}

Моё решение

def alfaMax(left: List[Char], right: List[Char]): List[Char] = {  
 (left, right) match {  
 case (Nil, Nil) => List.*empty* case (Nil,h::t)=>left  
 case(h::t,Nil)=>right  
 case (lh::lt, rh::rt) if (lh.toInt < rh.toInt) => left  
 case (lh :: lt, rh :: rt) if (lh.toInt > rh.toInt) => right  
 case (lh :: lt, rh :: rt) => lh::alfaMax(lt, rt)  
 }  
}

Моя программа сравнивает два списка Char. Если оба списка= Nil значит оба списка закончились и они равны, если один закончился а второй нет, возвращаем тот который короче. Иначе сравниваем ASCII код первых букв. Если и они равны, то рекурсивно отправляем в функцию tail.

# Разделение строк на слова

**Метод split() используется для разбиения строки на список подстрок на основе указанного разделителя.**

Реализация на c#

**public** **static** List<**object**> SplitEx(**this** **string** text, **char** \_split)

       {

**var** result = new List<**object**>();

**var** startIndex = 0;

**for** (**int** i = 0; i < text.Length; i++)

           {

**if** (text[i] == \_split)

               {

                   result.**Add**(text.Substring(startIndex, i - startIndex));

                   startIndex = ++i;

               }

           }

**return** result;

       }

Моя реализация

def wordCut(lst:List[Char]):List[List[Char]]={  
 lst match {  
 case Nil => List.*empty* case head :: tail => List(myTakeWhile(lst,(\_!=' ')))++wordCut(myDropWhile(lst,(\_!=' ')))  
 }  
}

# Sliding

Метод скользящего окна — **локальное усреднение показателя, позволяющее снять влияние случайных явлений с эмпирических кривых и вскрыть закономерные пространственные изменения изучаемого признака.** Наблюденные в ближайших точках (в пределах окна) данные суммируются и делятся на число точек в окне; полученное значение присваивается средней точке. Затем в заданном направлении окно перемещается на 1 точку и операция повторяется.

Код для Python

**def** **maxSum**(arr, k):

# length of the array

n = len(arr)

# length of array must be greater

# window size

**if** n < k:

**print**("Invalid")

**return** -**1**

# sum of first k elements

window\_sum = sum(arr[:k])

max\_sum = window\_sum

# remove the first element of previous

# window and add the last element of

# the current window to calculate the

# the sums of remaining windows by

**for** i **in** range(n - k):

window\_sum = window\_sum - arr[i] + arr[i + k]

max\_sum = max(window\_sum, max\_sum)

**return** max\_sum

Моя реализация

def mySlidingList[T](lst:List[T],num:Int):List[List[T]]={  
 lst match {  
 case Nil => List.*empty* case head :: tail => if(recursiveLength(lst.tail)>=num) List(lst.head::mySubList(lst.tail,0,num-1))++mySlidingList(lst.tail,num) else List.*empty* }  
}

# Sublist

**subList — это метод в интерфейсе List, который позволяет создать новый список из части существующего списка.**

**Код на c++**

public List<E> subList(int arg0, int arg1) {

ArrayList<E> ar = new ArrayList<E>();

ListIterator myiter=listIterator(arg0);

int k = arg1 - arg0 + 1;

int i;

for(i = 0; i < k; ++i) {

ar.add((E) myiter.next());

}

List <E> sublist=new GITLinkedList(ar);

return sublist;

}

Мой код

def mySubList[T](lst:List[T],start:Int,stop:Int):List[T]={  
 val lenght=recursiveLength(lst)  
 if(start<0 || stop<0 || start> lenght||stop> lenght|| stop<start) throw new ArithmeticException("Illegal")  
 myTake(myDrop(lst,start),stop-start)  
}

# Проблемы возникшие при разработке программы

После создания первой версии программы были выявлены следующие проблемы:

* Использование множественных конструкций else if - не функционально и не эстетично. можно заменить, не меняя суть алгоритма, но я бы использовал иной подход. (строки можно сцепить и проверять посимвольно). Скажем шире. Оправданное использование конструкции if-else в фп - редкость.

Для функции:

def which\_alfabetter(left:List[Char],right:List[Char]):Char= { if((left==List.empty)&&(right!=List.empty)) 'l'  
  else if((left!=List.empty)&&(right==List.empty)) 'r'   
else if((left==List.empty)&&(right==List.empty)) 'e'  
  else if(left.head.toInt < right.head.toInt)'l'//чем раньше буква в ASCII тем меньше её номер   
else if(left.head.toInt > right.head.toInt)'r'  
  else if(left.head.toInt == right.head.toInt) which\_alfabetter(left.tail,right.tail) else '?'  
}

Проведя работу по облагораживанию кода моя функция преобрела вид:

def alfaMax(left: List[Char], right: List[Char]): List[Char] = {  
 (left, right) match {  
 case (Nil, Nil) => List.*empty* case (Nil,h::t)=>left  
 case(h::t,Nil)=>right  
 case (lh::lt, rh::rt) if (lh.toInt < rh.toInt) => left  
 case (lh :: lt, rh :: rt) if (lh.toInt > rh.toInt) => right  
 case (lh :: lt, rh :: rt) => lh::alfaMax(lt, rt)  
 }  
}

* which\_alfabetter - названа в нотации snake. в scala для названия функций используется lowerCamelCase. т.е. whichAlfabetter

В программировании существует несколько стилей написания составных слов, таких как:

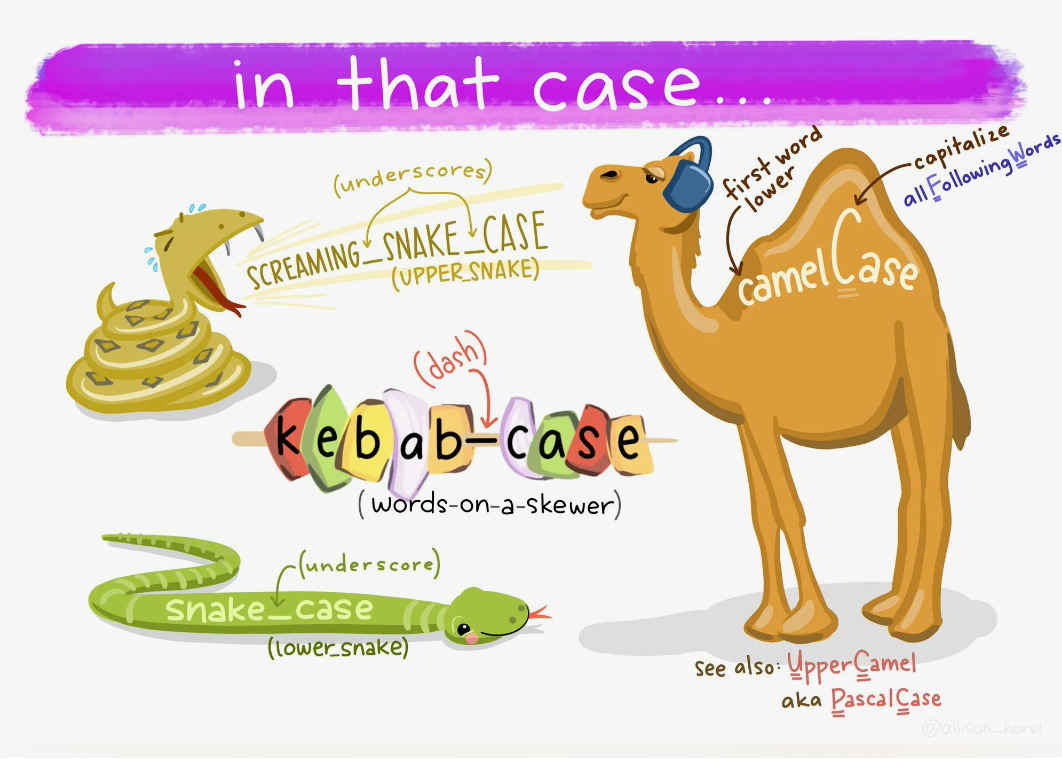


Рис 9.1 «Стили написания переменных»

1. CamelCase: Стиль написания, при котором слова в составном слове пишутся без пробелов, а каждое следующее слово начинается с заглавной буквы. Например, "camelCaseExample".

2. PascalCase: Похож на CamelCase, однако первая буква каждого слова также является заглавной. Например, "PascalCaseExample".

3. snake\_case: Стиль, при котором слова разделяются символом подчеркивания "\_". Например, "snake\_case\_example".

4. kebab-case: Стиль, где слова разделяются символом дефиса "-". Например, "kebab-case-example".

Выбор стиля написания составных слов зависит от предпочтений и соглашений в рамках конкретного проекта или команды разработчиков. Важно придерживаться одного стиля для повышения читаемости кода и обеспечения консистентности в проекте.

Для функции:

def wordcut(list: List[Char], list2:List[String], list3: List[Char] ):List[String] = { list match {  
    case Nil => list2 case head :: tail => { if (list.head == ' ')  
    {  
      wordcut(list.tail,list2++List(list3.mkString),List.empty) }  
    else {  
      wordcut(list.tail,list2,list3++List(list.head)) }  
    } }  
}

Эта функция оказалась достаточно грамоздской и сложной для восприятия человеку видяшему её впервые. Кроме того она требует слишком много параметров.

Функция была преобразована :

def wordCut(lst:List[Char]):List[List[Char]]={  
 lst match {  
 case Nil => List.*empty* case head :: tail => List(myTakeWhile(lst,(\_!=' ')))++wordCut(myDropWhile(lst,(\_!=' ')))  
 }  
}

# Тестовые примеры

* Дана строка. Если она начинается на 'abc', то заменить их на 'www', иначе добавить в конец строки 'zzz'.

Решение:

if(mySubString(str,0,3)=="abc")"www"++mySubString(str,3,lenghtString(str)) else str++"zzz"

* Замените в строке все вхождения 'word' на 'letter'.

Решение

spacer(myMap(wordCutStr(word),(x=>if(x=="word")"letter" else x))).mkString

* Дана строка. Определить, содержит ли строка только символы 'a', 'b', 'c' или нет.

if(myCount(abaList,(x=>if(x=='a'|| x=='b' || x=='c') true else false))==lenghtString(abac)) true else false

* Найдите количество вхождения 'aba' в строку.

mySum(myMap(mySliding(aba,lenghtString(subAba)), (x=>if(x=="aba")1 else 0)))

* Напечатать все слова, которые состоят из тех же букв что и последнее слово текста

val txtWords=wordCutStr(txt)  
*println*(myFilter(txtWords,x=>if(sortBy(x.toList,x=>x.toInt)==sortBy(txtWords.last.toList,x=>x.toInt))true else false))

* Напечатать самое длинное слово на букву 'a'

val longA="allocate ale cucumbers"  
val wordsLongA=wordCutStr(longA)  
val filtredLonga=myFilter(wordsLongA,x=>if(x.toList.head=='a')true else false)  
val resultLonga=sortBy(filtredLonga,lenghtString).last  
*println*(resultLonga)

# Вывод

Мне удалось разработать библиотеку для работы со строками с помощью которой можно решать все существующие на сегодняшний день задачи

# Список литературы

1. Geekforgeeks.org
2. Scala-lang.org
3. Habr.com
4. Learnscala.org

# Приложение – код программы

Файл main

import lib.{\*}  
@main  
def main(): Unit = {  
 *println*(mySliding("ababa",2))  
 //Дана строка. Если она начинается на 'abc', то заменить их на 'www',  
 // иначе добавить в конец строки 'zzz'.  
 val str="abcdef"  
 *println*(if(mySubString(str,0,3)=="abc")"www"++mySubString(str,3,lenghtString(str)) else str++"zzz")  
 //Замените в строке все вхождения 'word' на 'letter'.  
 val word="listen to my word"  
 *println*(spacer(myMap(wordCutStr(word),(x=>if(x=="word")"letter" else x))).mkString)  
 //Дана строка. Определить, содержит ли строка только символы 'a', 'b', 'c' или нет.  
 val abac="abac"  
 val abaList=abac.toList  
 *println*(if(myCount(abaList,(x=>if(x=='a'|| x=='b' || x=='c') true else false))==lenghtString(abac)) true else false)  
 //Найдите количество вхождения 'aba' в строку.  
 val aba="ababaka"  
 val subAba="aba"  
 *println*(mySum(myMap(mySliding(aba,lenghtString(subAba)), (x=>if(x=="aba")1 else 0))))  
 //Напечатать все слова, которые состоят из тех же букв что и последнее слово текста  
 val txt="cab bac lol abc"  
 val txtWords=wordCutStr(txt)  
 *println*(myFilter(txtWords,x=>if(sortBy(x.toList,x=>x.toInt)==sortBy(txtWords.last.toList,x=>x.toInt))true else false))  
 //Напечатать самое длинное слово на букву 'a'  
 val longA="allocate ale cucumbers"  
 val wordsLongA=wordCutStr(longA)  
 val filtredLonga=myFilter(wordsLongA,x=>if(x.toList.head=='a')true else false)  
 val resultLonga=sortBy(filtredLonga,lenghtString).last  
 *println*(resultLonga)  
  
}

Файл alfabetSort

package lib  
//сравнивает два списка символов кто из них алфавитнее  
def alfaMax(left: List[Char], right: List[Char]): List[Char] = {  
 (left, right) match {  
 case (Nil, Nil) => List.*empty* case (Nil,h::t)=>left  
 case(h::t,Nil)=>right  
 case (lh::lt, rh::rt) if (lh.toInt < rh.toInt) => left  
 case (lh :: lt, rh :: rt) if (lh.toInt > rh.toInt) => right  
 case (lh :: lt, rh :: rt) => lh::alfaMax(lt, rt)  
 }  
}  
  
//Сортировка по алфавиту  
def alfaSort(lst: List[List[Char]]): List[List[Char]] = {  
 if (lst == List.*empty*) lst  
 else {  
 val pivot = lst(recursiveLength(lst) / 2)  
 alfaSort(myFilter(lst,(x=>if(alfaMax(x,pivot)==x && x!=pivot)true else false))) ++  
 myFilter(lst,(x=>if(x==pivot)true else false)) ++  
 alfaSort( myFilter(lst, (x => if (alfaMax(x, pivot) == pivot && x!=pivot) true else false)))  
 }  
}

файл functions

package lib  
//своя длинна строки см lib.support  
def lenghtString(str:String):Int=recursiveLength(str.toList)  
//посчитать символ в строке см li.support  
def countInStr(str:String,ch:Char):Int={  
 myCount(str.toList,(\_==ch))  
}  
//делим строку на слова см lib.word\_cut  
def wordCutStr(str:String): List[String] = {  
 myMap(wordCut(str.toList),(x=>x.mkString))  
}  
//функция находящая уникальные слова в строке см lib.word\_cut  
def wordCutUnigue(str:String):List[String]={  
 antiDublicate(wordCutStr(str))  
}  
//самое короткое слово см lib.SorsFiltres  
def shortestWord(str:String):String={  
 sortBy(wordCutStr(str),lenghtString).head  
}  
//самое длинное слово см lib.SorsFiltres  
def longestWord(str:String):String={  
 myReverse(sortBy(wordCutStr(str),lenghtString)).head  
}  
//см lib.SorsFiltres  
def sortWords(str:String):List[String]={  
 sortBy(wordCutStr(str),lenghtString)  
}  
//Делает строку капса см lib.UpperCase  
def otherString(str:String):String={  
 otherList(str.toList).mkString  
}  
//Выделяет капсом нужное слово см lib.UpperCase  
def findWord(str:String,charector:String):String={  
 spacer(myMap(wordCutStr(str),(x=>if(x==charector)otherString(x) else x))).mkString  
}  
//Сортирует строку по алфавиту см lib.Alfabet\_sort  
def strAlfabetSort(str:String):List[String]={  
 myMap(alfaSort(myMap(wordCutStr(str), (x: String) => x.toList)),(x: List[Char]) => x.mkString)  
}  
def strMap(list: String,lambda: (Char=>Any) ):List[Any]={  
 myMap(list.toList,lambda)  
}  
def mySubString(str:String,start:Int,stop:Int):String={  
 mySubList(str.toList,start,stop).mkString  
}  
def mySliding(str:String,num:Int):List[String]={  
 myMap(mySlidingList(str.toList,num),(x=>x.mkString))  
}

файл otherCase

package lib  
def ifNumber(ch:Char):Boolean={  
 if (ch.toInt > 47 && ch.toInt < 58) true else false  
}  
def ifUpperEng(ch:Char):Boolean={  
 if(ch.toInt>64 && ch.toInt<91) true else false  
}  
def ifLowerEng(ch: Char): Boolean = {  
 if (ch.toInt > 96 && ch.toInt < 123) true else false  
}  
def ifUpperRus(ch: Char): Boolean = {  
 if (ch.toInt > 1039 && ch.toInt < 1072) true else false  
}  
def ifLowerRus(ch: Char): Boolean = {  
 if (ch.toInt > 1071 && ch.toInt < 1104) true else false  
}  
def otherCase(ch: Char):Char={  
 if(ifUpperEng(ch)||ifUpperRus(ch)) (ch.toInt+32).toChar else   
 if (ifLowerEng(ch) || ifLowerRus(ch)) (ch.toInt - 32).toChar else ch  
   
}  
  
//функция преобразовывающая список строчных букв в список Заглавных букв  
def otherList(lst:List[Char]):List[Char] ={  
 lst match {  
 case Nil => List.*empty* case head :: tail => List(otherCase(lst.head))++otherList(lst.tail)  
 }  
}

файл sortsFilter

package lib  
//quicksort для списка Int  
def myQuickSort(lst: List[Int]): List[Int] = {  
 if (lst==List.*empty*) lst  
 else {  
 val pivot = lst(recursiveLength(lst) / 2)  
 myQuickSort(myFilter(lst,(\_ < pivot)) ++  
 myFilter(lst,(\_ == pivot)) ++  
 myQuickSort(myFilter(lst,(\_ > pivot))))  
 }  
}  
//моя реализация фильтра  
def myFilter[T](list: List[T],option: ((T)=>Boolean) ):List[T]={  
 list match {  
 case Nil => List.*empty* case head :: tail => if(option(head)) List(list.head) ++ myFilter(list.tail,option) else myFilter(list.tail,option)  
 }  
}  
//фильтр которые фильтрует элементы списка в зависимости от результата возвращаемой функции  
def filterBy[T](list: List[T],option: ((Int)=>Boolean),lambda: (T=>Int) ):List[T]={  
 list match {  
 case Nil => List.*empty* case head :: tail => if(option(lambda(head))) List(list.head) ++ filterBy(list.tail,option,lambda) else filterBy(list.tail,option,lambda)  
 }  
}  
//сортировка в зависимости от результатов возвращаемой функции  
def sortBy[T](lst: List[T],lambda:(T=>Int)): List[T] = {  
 if (lst==List.*empty*) lst  
 else {  
 val pivot = lambda(lst(recursiveLength(lst) / 2))  
 sortBy(filterBy(lst,(\_ < pivot),lambda),lambda) ++  
 filterBy(lst,(\_ == pivot),lambda) ++  
 sortBy(filterBy(lst,(\_ > pivot),lambda),lambda)  
 }  
}

файл support

package lib  
//длинна массива  
def recursiveLength(lst: List[\_]):Int= {  
 lst match {  
 case Nil => 0  
 case head :: tail => 1+recursiveLength(lst.tail)  
 }  
}  
//сумма в массиве Int  
def mySum(lst: List[Int]):Int={  
 lst match {  
 case Nil => 0  
 case head :: tail => lst.head + mySum(lst.tail)  
 }  
}  
//сделаем свой рекурсивный map  
def myMap[T,U](list: List[T],lambda: (T=>U) ):List[U]={  
 list match {  
 case Nil => List.*empty* case head :: tail => List(lambda(list.head)) ++ myMap(list.tail,lambda)  
 }  
}  
//reverse инвертируем список  
def myReverse[T](list: List[T]):List[T]={  
 list match {  
 case Nil => List.*empty* case head :: tail => myReverse(list.tail)++List(list.head)  
 }  
}  
//свой рекурсивный count  
def myCount[T,U](list: List[T],option: ((T)=>Boolean) ):Int={  
 list match {  
 case Nil => 0  
 case head :: tail => if(option(head)) 1 + myCount(list.tail,option) else myCount(list.tail,option)  
 }  
}  
def mySubList[T](lst:List[T],start:Int,stop:Int):List[T]={  
 val lenght=recursiveLength(lst)  
 (start, stop) match{  
 case (start, stop) if(start<0)=> throw new IllegalArgumentException("Illegal")  
 case (start, stop) if(stop<0)=> throw new IllegalArgumentException("Illegal")  
 case (start, stop) if(start>stop)=> throw new IllegalArgumentException("Illegal")  
 case (start, stop) if(start<0)=> throw new IllegalArgumentException("Illegal")  
 }  
 if(start<0 || stop<0 || start> lenght||stop> lenght|| stop<start) throw new IllegalArgumentException("Illegal")  
 myTake(myDrop(lst,start),stop-start)  
}  
def myDrop[T](lst:List[T],it:Int):List[T]={  
 lst match {  
 case Nil => List.*empty* case head :: tail => if(it==0) lst else myDrop(lst.tail,it-1)  
 }  
}  
def myTake[T](lst:List[T],it:Int):List[T]={  
 lst match {  
 case Nil => List.*empty* case head :: tail => if(it==0) List.*empty* else lst.head::myTake(lst.tail,it-1)  
 }  
}  
def mySlidingList[T](lst:List[T],num:Int):List[List[T]]={  
 lst match {  
 case Nil => List.*empty* case head :: tail => if(recursiveLength(lst.tail)>=num-1) List(lst.head::myTake(lst.tail,num-1))++mySlidingList(lst.tail,num) else List.*empty* }  
}

файл wordCut

package lib  
//функция сохраняет список пока option==true  
def myTakeWhile[T](lst:List[T],option: T=>Boolean):List[T]={  
 lst match {  
 case Nil => List.*empty* case head :: tail => if(option(lst.head)) List(lst.head)++myTakeWhile(lst.tail,option) else List.*empty* }  
}  
//функция обратная предыдущей  
def myDropWhile[T](lst: List[T], option: T => Boolean): List[T] = {  
 lst match {  
 case Nil => List.*empty* case head :: tail => if (option(lst.head)) myDropWhile(lst.tail, option) else lst.tail  
 }  
}  
//делим строку на слова  
def wordCut(lst:List[Char]):List[List[Char]]={  
 lst match {  
 case Nil => List.*empty* case head :: tail => List(myTakeWhile(lst,(\_!=' ')))++wordCut(myDropWhile(lst,(\_!=' ')))  
 }  
}  
// функция находящая элемент в массиве  
def ifExist(list:List[\_],character: Any):Boolean= {  
 list match {  
 case Nil => false  
 case head :: tail => {  
 if (list.head == character) true else ifExist(list.tail, character)  
 }  
 }  
}  
  
//функция удаляющая дубликаты из массива  
def antiDublicate[T](list:List[T]):List[T]={  
 list match {  
 case Nil => List.*empty* case head :: tail => if (ifExist(list.tail,list.head)) list.tail else List(list.head) ++ antiDublicate(list.tail)  
 }  
}  
//вставляет пробелы  
def spacer(lst:List[String]):List[String]= {  
 lst match {  
 case Nil => lst  
 case head :: tail => {  
 List(lst.head, " ") ++ spacer(lst.tail)  
 }  
 }  
}