ЛАБОРАТОРНАЯ РАБОТА №1
ИМПЕРАТИВНОЕ (ПРОЦЕДУРНОЕ) ПРОГРАММИРОВАНИЕ

Цель: познакомиться с особенностями процедурного программирования. Решить задания в процедурном стиле. Составить отчет.

Теоретические сведения

Выполнение программы в процедурном стиле сводится к последовательному выполнению операторов с целью преобразования исходного состояния памяти, т.е. значений исходных данных, в заключительное, т.е. в результаты. Таким образом, с точки зрения программиста имеются программа и память, причем первая последовательно обновляет содержимое последней.

Процедурный стиль предоставляет возможность программисту определять каждый шаг в процессе решения задачи. Особенность таких языков программирования состоит в том, что задачи разбиваются на шаги и решаются шаг за шагом. Используя процедурный язык, программист определяет языковые конструкции для выполнения последовательности алгоритмических шагов.

Переменная состоит из имени и выделенной области памяти, которая соответствует ей. Для объявления или, другими словами, создания переменной используются директивы (ключевые слова, конструкции). В разных языках программирования создание переменных отличается.

Функция – это подпрограмма специального вида, которая может принимать на вход параметры, выполнять различные действий и передавать результаты работы. Вызов функции является, с точки зрения языка программирования, выражением, он может использоваться в других выражениях или в качестве правой части присваивания.

Процедура – это независимая именованная часть программы, которую после однократного описания можно многократно вызвать по имени из последующих частей программы для выполнения определенных действий.

```r
chr_var <- "Привет, Мир!" #Создание переменной строкового типа (character)
num_var <- 24.3 #Создание переменной вещественного типа (numeric)
int_var <- 124L #Создание переменной целочисленного типа (integer)
cplx_var <- 56 + 4i #Создание переменной комплексного типа (complex)
logi_var <- TRUE #Создание переменной логического типа (logical)
assign("name_var", "Какое-то значение") #Создание переменной с помощью assign()
```

Рисунок 1 – Создание переменных в языке R

Функции. Для создания функций используются служебные слова: func, def, function и др. Синтаксис создания функций отличается в разных языках программирования.

```r
#Создание функции, возвращающей большее значение
bigger <- function(a, b) {
  if (a > b) {
    return(a)
  } else {
    return(b)
  }
}
bigger(5, 8) #Корректное использование функции
bigger(5) #Не корректное использование функции
bigger(7, 6, 9) #Не корректное использование функции
```

Рисунок 2 – Создание и использование функций в языке R

Написание программ

С развитием языков программирования развивались и технологии, используемые при написании программного кода. Первые программы писались сплошным текстом. Это была простая последовательность команд. Все это выглядело следующим образом:

```
Начало программы
Команда 1
Команда 2
...
Команда N
Конец программы
```

Рисунок 3 – Структура линейной программы

Используя такой подход к программированию, можно было сделать очень мало. Единственное, что было доступно программисту для создания логики в данном случае – это переходы. Под переходом понимается переход на какую-то команду при определенных условиях, которые сложились в процессе обработки данных на процессоре.

```
Начало программы
Команда 1
Команда 2
Если выполнено условие, то вернуться к команде 1
Команда 3
...
Команда N
Конец программы
```

Рисунок 4 – Образец части программы, построенной с использованием безусловного перехода

Безусловный переход (goto, jmp).

Как правило, оператор безусловного перехода состоит из двух частей: собственно оператора и метки, указывающей целевую точку перехода в программе (например: goto метка). Метка, в зависимости от правил языка, может быть либо числом (как, например, в классическом BASIC), либо идентификатором используемого языка программирования. Для меток-идентификаторов метка, как правило, ставится перед оператором, на который должен осуществляться переход, и отделяется от него двоеточием (метка:).

Действие оператора перехода состоит в том, что после его исполнения следующими будут исполняться операторы программы, идущие в тексте непосредственно после метки (до следующего оператора перехода, ветвления или цикла). Для машинных языков инструкция перехода копирует в регистр процессора, содержащий адрес следующей выполняемой команды, адрес команды, помеченной меткой.

```
1 i=0
2 i+=1
3 PRINT i; "squared="; i*i
4 IF i>100 THEN GOTO 6
5 GOTO 2
6 PRINT "Program Completed."
7 END
```

а – процедурный стиль,

```
1 FOR i=1 TO 100
2     PRINT i; "squared="; i*i
3 NEXT i
4 PRINT "Program Completed."
5 END
```

б – структурное программирование

Рисунок 5 – Пример программ, печатающих числа от 1 до 100 и квадраты этих чисел, выполненные в разных стилях на языке программирования BASIC

В процедурном подходе какой-то код программы мог объединяться в отдельные блоки (процедуры). После этого такой блок команд можно вызывать из любой части программы.

```
Начало процедуры 1
Команда 1
Команда 2
Конец процедуры 1
Начало программы
Команда 1
Команда 2
Если выполнено условие, то выполнить код процедуры 1.
Команда 3
Конец программы.
```

Рисунок 6 – Структура программы, выполненной в процедурном стиле

Практическая часть

Задание 1 – Написать программу, выполненную в процедурном стиле. Программа должна быть выполнена в виде псевдокода, в виде блок-схемы и на языке высокого уровня (ЯВУ) (здесь и далее, если не оговорено иное, при отсылке к ЯВУ необходимо выполнять код на языке R). Для построения блок-схемы рекомендуется использовать ресурс draw.io или аналогичную программу. Построение блок схемы делается с учетом правил, содержащихся в презентации Императивное (процедурное) программирование.

Вариант 1

Напишите программу, рассчитывающую площадь трех фигур: квадрат, прямоугольник и круг. На входе программа запрашивает введение данных о фигурах (для квадрата – сторона, круг – радиус, прямоугольник – две стороны). На выходе программа указывает площади трех фигур и общую площадь.

Вариант 2

Напишите программу, рассчитывающую сумму расходов за месяц. На входе программа запрашивает сведения о расходах по нескольким пунктам (минимум 3 статьи расходов). На выходе программа указывает суммарные расходы и максимальную статью расходов.

Вариант 3

Напишите программу, подсчитывающую среднее количество занятий в неделю. Программа запрашивает информацию о количестве занятий в день (по дням недели). На выходе программа указывает среднее количество занятий в неделю с округлением до ближайшего целого числа.

Задание 2 – Опишите, представленный код в виде псевдокода и ответьте на вопрос, что будет получено при передаче функции числа 7? Также реализуйте данный алгоритм на ЯВУ.

Функция на ассемблере с синтаксисом AT&T:

```
foo:
  cmp $0, %edi
  jg calc
  mov $1, %eax
  jmp exit
calc:
  push %edi
  sub $1, %edi
  call foo
  pop %edi
  imul %edi, %eax
exit:
  ret
```

Это функция с одним входным параметром, для которой ABI (двоичный интерфейс приложений) предписывает передачу одного параметра через регистр %edi, а передачу возвращаемого значения через регистр %eax.

Замечания:

1) Команда 'imul src, dest' умножает src на dest и кладет результат в dest.

2) Не нужно думать о переполнении. Его здесь не будет.

Контрольный вопросы:

1. Особенности процедурного программирования
2. Линейная программа
3. Понятия: переменная, процедура, функция,
4. Безусловный оператор

Вопросы для поиска и письменного ответа:

1. Хронология процедурных языков
2. Спагетти-код (особенность и причины)
3. Процедурный стиль и архитектура фон Неймана (взаимосвязь)

Лабораторная работа №2

СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

Цель: познакомиться с особенностями структурного программирования. Решить задания в структурном стиле. Составить отчет.

Теоретические сведения

Структурное программирование – методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков.

В соответствии с данной методологией любая программа строится без использования оператора goto из трех базовых управляющих структур: последовательность, ветвление, цикл; кроме того, используются подпрограммы. При этом разработка программы ведется пошагово, методом «сверху вниз».

Цель структурного программирования – повысить производительность труда программистов, в том числе при разработке больших и сложных программных комплексов, сократить число ошибок, упростить отладку, модификацию и сопровождение программного обеспечения.

Теорема Бёма – Якопини – положение структурного программирования, согласно которому любой исполняемый алгоритм может быть преобразован к структурированному виду, то есть такому виду, когда ход его выполнения определяется только при помощи трех структур управления: последовательной, ветвлений и повторов или циклов.

Цикл – разновидность управляющей конструкции в высокоуровневых языках программирования, предназначенная для организации многократного исполнения набора инструкций.

Бесконечный цикл – цикл, написанный таким образом, что условие выхода из него никогда не выполняется.

```
repeat {
  # Выполняем действие
}
```

Рисунок 1 – Бесконечный цикл в R

```
while(TRUE) {
  # Выполняем действие
}
```

Рисунок 2 – Бесконечный цикл в R (на основе цикла while)

Цикл с предусловием – цикл, который выполняется, пока истинно некоторое условие, указанное перед его началом. Это условие проверяется **до** выполнения тела цикла, поэтому тело может быть не выполнено ни разу (если условие с самого начала ложно). В большинстве процедурных языков программирования реализуется оператором **while**, отсюда его второе название – while-цикл.

```
i <- 0
while(i <- 5) {
  i <- i + 1
  print(i)
}
```

Рисунок 3 – Цикл с предусловием в R (while)

Цикл со счетчиком – цикл, в котором некоторая переменная изменяет свое значение от заданного начального значения до конечного значения с некоторым шагом, и для каждого значения этой переменной тело цикла выполняется один раз. В большинстве языков программирования реализуется оператором for, в котором указывается счетчик (так

называемая «переменная цикла»), требуемое количество проходов (или граничное значение счетчика) и, возможно, шаг, с которым изменяется счетчик

```
for (i in 1:5) {
  print(i)
}
```

Рисунок 4 – Цикл со счетчиком в R (for)

Во многих языках программирования вместо стандартного счетчика, имеющего индексное значение, можно пройтись по итерируемой последовательности.

```
cities <- c("New York",
            "Paris",
            "Saint-Denis",
            "Oslo",
            "Montreal",
            "Helsinki")
for (city in cities) {
  print(city)
}
```

Рисунок 5 – Цикл со счетчиком в R (for), перебирающий итерируемый вектор

Досрочный выход из цикла. Команда досрочного выхода применяется, когда необходимо прервать выполнение цикла, в котором условие выхода еще не достигнуто. Такое бывает, например, когда при выполнении тела цикла обнаруживается ошибка, после которой дальнейшая работа цикла не имеет смысла.

Команда досрочного выхода обычно называется break или exit, а ее действие аналогично действию команды безусловного перехода (goto) на команду, непосредственно следующую за циклом, внутри которого эта команда находится.

```
for(i in 1:5) {
  if (i == 3) {
    break
    }
  print(i)
}
```

Рисунок 6 – Досрочный выход из цикла for в R

Пропуск итерации. Данный оператор применяется, когда в текущей итерации цикла необходимо пропустить все команды до конца тела цикла. При этом сам цикл прерываться не должен, условия продолжения или выхода должны вычисляться обычным образом.

В языке C, его языках-потомках и во многих других языках программирования в качестве команды пропуска итерации используется оператор continue в конструкции цикла. Действие этого оператора аналогично безусловному переходу на строку внутри тела цикла, следующую за последней его командой. В языке R специальным оператор для пропуска итерации является next.

```
for(i in 1:5) {
  if (i == 3) {
    next
    }
  print(i)
}
```

Рисунок 7 – Пропуск итерации в цикле for в R

Практическая часть

Задание 1 – Написать программу, выполненную в структурном стиле. Программа должна рассчитывать площадь фигур (программа должна корректно отрабатывать данные

согласно варианту в приложении А). На вход программа запрашивает строку, если в нее введено название фигуры, то программа запрашивает необходимые параметры фигуры, если введено значение отличное от названия фигуры, то программа повторно предлагает ввести название фигуры, если пользователь не справляется с этой задачей более 3 раз подряд, то программа сообщает о некорректности действий пользователя и завершается. В случае введения корректных данных программа должна выдать ответ, а также описание хода решения.

Программа должна быть выполнена в виде блок-схемы и на ЯВУ.

Задание 2 – Написать программу вычисляющую площадь неправильного многоугольника. Многоугольник на плоскости задается целочисленными координатами своих N вершин в декартовой системе. Стороны многоугольника не соприкасаются (за исключением соседних - в вершинах) и не пересекаются. Программа в первой строке должна принимать число N – количество вершин многоугольника, в последующих N строках – координаты соответствующих вершин (вершины задаются в последовательности против часовой стрелки). На выход программа должна выдавать площадь фигуры.

Программа должна быть выполнена в виде блок-схемы и на ЯВУ.

Вопросы для контроля из материалов лабораторного занятия

1. Особенности структурного программирования
2. Теорема Бёма – Якопини
3. Пропуск итерации и досрочный выход из цикла

Вопросы для поиска и письменного ответа

1. Цикл с постусловием
2. Совместный цикл
3. Вложенные циклы
4. Принцип проектирования программ «сверху-вниз».

ПРИЛОЖЕНИЕ А – ВАРИАНТЫ К ЗАДАНИЮ 1

Вариант	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Треугольник														
Квадрат														
Прямоугольник											+	+	+	+
Параллелограмм					+	+	+	+	+	+				
Ромб		+	+	+				+	+	+				+
Трапеция	+		+	+		+	+			+		+	+	
Круг	+	+		+	+		+		+		+		+	
Эллипс	+	+	+		+	+		+			+	+		+
Вариант	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**
Треугольник														
Квадрат							+	+	+	+	+	+	+	+
Прямоугольник	+	+	+	+	+	+								
Параллелограмм			+	+	+	+							+	+
Ромб	+	+				+				+	+	+		
Трапеция		+			+			+	+			+		
Круг	+			+			+		+		+			+
Эллипс			+				+	+		+			+	
Вариант	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**
Треугольник								+	+	+	+	+	+	+
Квадрат	+	+	+	+	+	+	+							
Прямоугольник			+	+	+	+	+							
Параллелограмм	+	+					+							+
Ромб		+				+					+	+	+	
Трапеция	+				+				+	+			+	
Круг				+				+		+		+		
Эллипс			+					+	+		+			+
Вариант	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**
Треугольник	+	+	+	+	+	+	+	+	+	+	+	+	+	+
Квадрат									+	+	+	+	+	+
Прямоугольник				+	+	+	+	+						+
Параллелограмм	+	+	+					+					+	
Ромб			+				+					+		
Трапеция		+				+					+			
Круг	+				+					+				
Эллипс				+					+					

Лабораторная работа №3

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Цель: познакомиться с особенностями объектно-ориентированного программирования. Научиться создавать собственные классы с использованием R6. Решить задания в соответствующем стиле программирования. Составить отчет.

Теоретические сведения

Объектно-ориентированное программирование (ООП) – методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Принципы ООП по Алану Кею

- все является объектом;
- вычисления осуществляются путем взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некоторое действие;
- объекты взаимодействуют, посылая и получая сообщения;
- сообщение – это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия;
- каждый объект имеет независимую память, которая состоит из других объектов;
- каждый объект является представителем (экземпляром) класса, который выражает общие свойства объектов.
- в классе задается поведение (функциональность) объекта.
- все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия;
- классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования
- память и поведение, связанное с экземплярами определенного класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.
- программа представляет собой набор объектов, имеющих состояние и поведение;
- устойчивость и управляемость системы обеспечивается за счет четкого разделения ответственности объектов (за каждое действие отвечает определенный объект), однозначного определения интерфейсов межобъектного взаимодействия и полной изолированности внутренней структуры объекта от внешней среды (инкапсуляции).

Механизмы ООП

Абстракция – придание объекту характеристик, которые отличают его от всех объектов, четко определяя его концептуальные границы;

Инкапсуляция – можно скрыть ненужные внутренние подробности работы объекта от окружающего мира (алгоритмы работы хранятся вместе с данными);

Наследование – можно создавать специализированные классы на основе базовых (позволяет избегать написания повторного кода);

Полиморфизм – в разных объектах одна и та же операция может выполнять различные функции;

Композиция – объект может быть составным и включать другие объекты.

Некоторые понятия

Объект – абстракция данных;

Объект – это отдельный представитель класса, имеющий конкретное состояние и поведение, полностью определяемое классом;

Объект: тип, методы,

Данные – объекты и отношения между ними;

Класс – это способ описания сущности, определяющий состояние и поведение, зависящее от этого состояния, а также правила для взаимодействия с данной сущностью (контракт)

С точки зрения программирования класс можно рассматривать как набор данных (полей, атрибутов, членов класса) и функций для работы с ними (методов).

Атрибут класса – содержательная характеристика класса, описывающая множество значений, которые могут принимать отдельные объекты этого класса

Методы класса – функция, которая может выполнять какие-либо действия над данными (свойствами) класса.

Дженерик (обобщенная функция) – функция, способная принимать разные структуры данных (разные классы), и работающая по-разному с данными структурами.

Синтаксис создания дженериков (на примере языка R)

```
get_n_elements <- function(x, ...){
  UseMethod("get_n_elements")
}
```

Рисунок 1 – Создание дженерика (подсчет элементов)

```
get_n_elements.data.frame <- function(x, ...){
  return(nrow(x) * ncol(x))
}
```

Рисунок 2 – Описание работы с классом data.frame для дженерика (подсчет элементов)

```
get_n_elements.default <- function(x, ...){
  return(length(unlist(x)))
}
```

Рисунок 3 – Описание работы по умолчанию для дженерика (подсчет элементов)

```
vec_numbers <- rnorm(10, mean = 0, sd = 1)
class(vec_numbers) <- "norm_distrib"
class(vec_numbers)
```

Рисунок 4 – Определение собственного класса

Генератор классов – шаблон для создания объектов.

Для создания классов в виде сложных структур в языке программирования R рекомендуется использовать пакет R6. Для работы с пакетом его сперва необходимо установить, а затем подключить, используя стандартные функции install.packages() и library(). Следующим этапом является написание генератора класса с использованием функции R6Class() пакета R6. При написании генератора используется стиль, похожий на создание функций, за исключением того, что вместо function используется ключевое слово R6Class, в примере на рисунке 5 представлен генератор класса Thing.

```
ThingFactory <- R6Class(
  "Thing",
  private = list(
    a_field = "a value",
    another_field = 123
  )
)
```

Рисунок 5 – Создание генератора класса Thing

Генератор класса в качестве первого аргумента получает имя класса, далее в генератор передается до трех списков: private, active и public. Список active целесообразно рассматривать с точки зрения ограничения доступа, и использования данных исключительно на чтение (в рамках данного занятия не рассматривается). Имя класса рекомендуется писать в стиле UpperCamelCase.

Для создания нового объекта класса необходимо вызвать генератор с добавлением $new().

```
thing_factory_object <- ThingFactory$new()
```

Рисунок 6 – Создание объекта класса Thing

В ходе дальнейшего рассмотрения мы остановимся на двух списках: private и public, логика конструктора предполагает, что содержимое списка private является данными, а содержимое списка public – методами. На рисунке 7 показан пример создания метода в списке public который выводит на печать содержимое переменных, находящихся в списке private.

```
ThingFactory <- R6Class(
  "Thing",
  private = list(
    a_field = "a value",
    another_field = 123
  ),
  public = list(
    do_something = function(x, y, z) {
      # Доступ к приватным полям
      paste(
        private$a_field,
        private$another_field
      )
    }
  )
)
```

Рисунок 7 – Создание генератора класса Thing с двумя списками

При создании объектов бывает необходимо создавать их с некоторыми значениями (передавать параметры в список private). Для этого необходимо использовать специальный метод initialize() в списке public. Данный метод автоматически вызывается при вызове конструктора, но может быть переопределен пользователем (рисунок 8).

```
ThingFactory <- R6Class(
  "Thing",
  private = list(
    a_field = "a value",
    another_field = 123
  ),
  public = list(
    initialize = function(a_field, another_field) {
      if(!missing(a_field)) {
        private$a_field <- a_field
      }
      if(!missing(another_field)) {
        private$another_field <- another_field
      }
    },
    print = function() {
      print(paste0(private$a_field, " = ", private$another_field))
    }
  )
)
```

Рисунок 8 – Создание генератора класса Thing с определением метода initialize()

Для создания объекта с другими значениями необходимо передать эти значения в генератор как представлено на рисунке 9.

```
a_thing <- ThingFactory$new(
  a_field = "a different value",
  another_field = 456
)
```

Рисунок 9 – Создание объекта класса Thing с новыми значениями

Практическая часть

Задание 1. Создайте дженерик, принимающий вектор, содержащий параметры фигуры и вычисляющий ее площадь. Для разных фигур создайте разные классы. В качестве метода по умолчанию дженерик должен выводить сообщение о невозможности обработки данных.

Задание 2. Создайте генератор класса Микроволновая печь. В качестве данных класс должен содержать сведения о мощности печи (Вт) и о состоянии дверцы (открыта или закрыта). Данный класс должен обладать методами открыть и закрыть дверь микроволновки, а также методом, отвечающим за приготовление пищи. Метод, отвечающий за приготовление пищи, должен вводить систему в бездействие (используется Sys.sleep) на определенное количество времени (которое зависит от мощности печи) и после выводить сообщение о готовности пищи.

Выполните создание двух объектов этого класса со значением по умолчанию и с передаваемыми значениями. Продемонстрируйте работу этих объектов по приготовлению пищи.

Задание 3. Создайте класс копилка. Описание структуры классы выполните из своего понимания копилки.

Вопросы для контроля из материалов лабораторного занятия

1. Принципы ООП по Алану Кею
2. Механизмы ООП
3. Основные понятия ООП
4. Создание и назначение дженериков
5. Создание класса в R6
6. Структура класса в R6

Вопросы для поиска и письменного ответа

1. История появления ООП. Основные этапы
2. Связь ООП с другими парадигмами программирования
3. Чистые языки, реализующие концепцию ООП. История появления
4. Мультипарадигмальные языки, реализующие концепцию ООП. История появления

Лабораторная работа №4

ВЕКТОРНОЕ ПРОГРАММИРОВАНИЕ

Цель: познакомиться с особенностями векторного программирования в R. Решить задания в соответствующем стиле программирования. Составить отчет.

Теоретические сведения

R – язык программирования для научных вычислений и анализа данных с упором на визуализацию и воспроизводимость;

R – свободное кроссплатформенное программное обеспечение с открытым исходным кодом;

R – интерпретируемый язык с интерфейсом командной строки;

R – мультипарадигмальный, векторный язык, сочетающий в себе:

функциональное программирование;

процедурное программирование;

объектно-ориентированное программирование;

рефлексивное программирование.

Векторизация – поэлементное одновременное выполнение действий над всеми элементами.

Основные объекты языка:

Вектор – основной объект языка R. Вектор может содержать более одного значения, все объекты в векторе имеют одну природу.

Лист – элемент языка R который может содержать разные по размеру и типу данных векторы.

Дата фрейм – элемент языка R который может содержать одинаковые по размеру, но разные по типу данных векторы. Дата фрейм является разновидностью листа.

Примеры синтаксиса языка R

```
my_var1 <- 42
my_var2 <- 35.25
my_var1 + 100
my_var1 + my_var2 - 12
my_var3 <- my_var1^2 + my_var2^2
my_var3
```

Рисунок 1 – Создание объектов и работа с ними (выполните код)

```
my_var3 > 200
my_var3 > 5000
my_var1 == my_var2
a <- 34
a = 2
my_var1 != my_var2
my_new_var <- my_var1 == my_var2
my_new_var
```

Рисунок 2 – Получение логических переменных (выполните код)

```
1:67
my_vector1 <- 1:67
my_vector2 <- c(-32, 45, 67, 12.78, 129, 0, -65)
my_vector1[1]
my_vector1[3]
my_vector2[2]
my_vector2[c(1, 2, 3)]
my_vector2[1:3]
my_vector2[c(1, 5, 6, 7, 10)]
```

Рисунок 3 – Создание векторов и обращение к элементам вектора (выполните код)

```
my_vector1 + 10
my_vector2 + 56
my_vector2 == 0
my_vector1 > 30
x <- 23
my_vector1 > 23
my_vector1 > x
x == 23
```

Рисунок 4 – Арифметические и логические операции (выполните код)

```
my_vector2 > 0
my_vector2[my_vector2 > 0]
my_vector2[my_vector2 < 0]
my_vector2[my_vector2 == 0]
my_vector1[my_vector1 > 20 & my_vector1 < 30]
my_numbers <- my_vector1[my_vector1 > 20 & my_vector1 < 30]
positeve_numbers <- my_vector2[my_vector2 > 0]
```

Рисунок 5 – Логические выборки (выполните код)

```
age <- c(16, 18, 22, 27)
is_married <- c(F, F, T, T)
data <- list(age, is_married)
data
data[[1]][1]
data[[2]][3]
name <- c('Olga', 'Maria', 'Nastya', 'Polina')
data <- list(age, is_married, name)
data
```

Рисунок 6 – Создание листа (выполните код)

```
df <- data.frame(Name = name, Age = age, Status = is_married)
df
```

Рисунок 7 – Создание дата фрейма (выполните код)

Создание собственных функций в R:

Для создания собственных функций в R используется следующая конструкций:

```
function_name <- function(argument_1, argument_2){
  function_body
  ...
  return(value)
}
```

Рисунок 8 – Конструкция создания функций в R

```
area_circle <- function(r){
  area <- r^2*pi
  return(area)
}
# Вызов функции
area_circle(5)
```

Рисунок 9 – Пример создания и вызова функции «Площадь круга»

Некоторые объекты языка R:

Матрица (matrix) представляет собой двумерную совокупность числовых, логических или текстовых величин.

Массив (array) – это совокупность некоторых однотипных элементов, обладающая размерностью больше двух.

Векторизованные функции:

Характерной особенностью R является векторизация вычислений. Векторизация представляет собой один из способов выполнения параллельных вычислений, при котором

программа определенным образом модифицируется для выполнения нескольких однотипных операций одновременно.

Принцип векторизованных вычислений применим не только к векторам как таковым, но и к более сложным объектам R - матрицам, спискам и таблицам данных (для R разницы между последними двумя типами объектов не существует – фактически таблица данных является списком из нескольких компонентов-векторов одинакового размера). В базовой комплектации R имеется целое семейство функций, предназначенных для организации векторизованных вычислений над такими объектами. В названии всех этих функций имеется слово apply (англ. применить), которому предшествует буква, указывающая на принцип работы той или иной функции (см. подробнее в справочном файле - ?apply).

Функция apply() - используется в случаях, когда необходимо применить какую-либо функцию ко всем строкам или столбцам матрицы (или массивов большей размерности):

```
# Создадим обычную двумерную матрицу
M <- matrix(seq(1, 16), 4, 4)
# Найдем минимальные значения в каждой строке матрицы
apply(M, 1, min)
# Найдем минимальные значения в каждом столбце матрицы
apply(M, 2, max)
```

Рисунок 10 – Пример работы с функцией apply

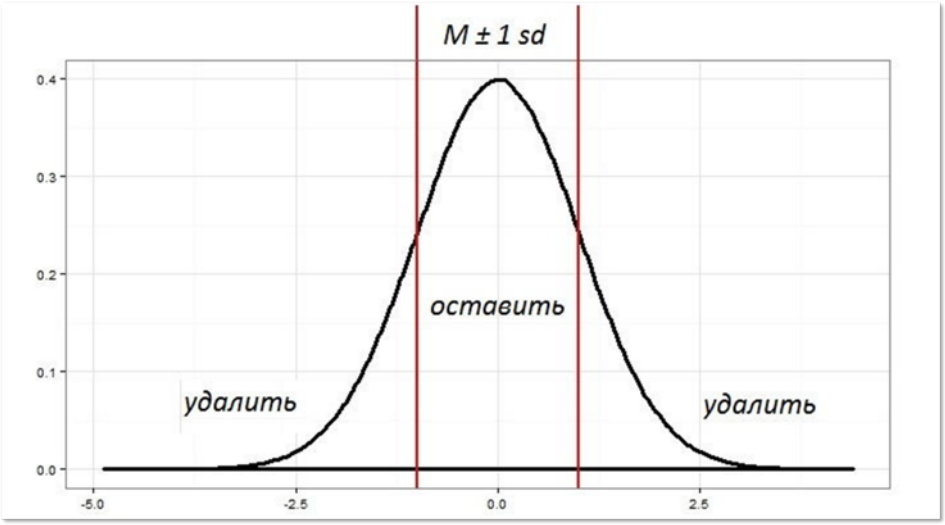
Практическая часть

Задание 1 – Предобработка данных. Создайте новый вектор my_vector, следующей строчкой:

```
my_vector <- c(21, 18, 21, 19, 25, 20, 17, 17, 18, 22, 17, 18, 18, 19, 19, 27, 21, 20,
24, 17, 15, 24, 24, 29, 19, 14, 21, 17, 19, 18, 18, 20, 21, 21, 19, 17, 21, 13, 17, 13,
23, 15, 23, 24, 16, 17, 25, 24, 22)
```

В векторе my_vector отберите только те наблюдения, которые отклоняются от среднего меньше, чем на одно стандартное отклонение. Сохраните эти наблюдения в новую переменную my_vector2. При этом исходный вектор оставьте без изменений.

Полезные функции: mean(x) – среднее значение вектора x; sd(x) – стандартное отклонение вектора x; abs(x) – абсолютное значение числа n



Задание 2. Напишите функцию get_negative_values, которая получает на вход dataframe произвольного размера. Функция должна для каждой переменной в данных проверять, есть ли в ней отрицательные значения. Если в переменной отрицательных значений нет, то эта

переменная нас не интересует, для всех переменных, в которых есть отрицательные значения мы сохраним их в виде списка или матрицы, если число элементов будет одинаковым в каждой переменной (смотри пример работы функции).

```
test_data <- as.data.frame(list(V1 = c(-9.7, -10, -10.5, -7.8, -8.9), V2 = c(NA, -10.2,
-10.1, -9.3, -12.2), V3 = c(NA, NA, -9.3, -10.9, -9.8)))
get_negative_values(test_data)

$V1
[1]  -9.7 -10.0 -10.5  -7.8  -8.9
$V2
[1] -10.2 -10.1  -9.3 -12.2
$V3
[1]  -9.3 -10.9  -9.8

test_data <- as.data.frame(list(V1 = c(NA, 0.5, 0.7, 8), V2 = c(-0.3, NA, 2, 1.2), V3 =
c(2, -1, -5, -1.2)))
get_negative_values(test_data)

$V2
[1] -0.3
$V3
[1] -1.0 -5.0 -1.2

test_data <- as.data.frame(list(V1 = c(NA, -0.5, -0.7, -8), V2 = c(-0.3, NA, -2, -1.2),
V3 = c(1, 2, 3, NA)))
get_negative_values(test_data)

      V1   V2
[1,] -0.5 -0.3
[2,] -0.7 -2.0
[3,] -8.0 -1.2
```

Вопросы для контроля из материалов лабораторного занятия

1. Векторизация;
2. Основные объекты языка R.
3. Создание собственных функций
4. Векторизованные функции семейства apply.

Вопросы для поиска и письменного ответа

1. Особенности языка программирования R;
2. Языки, поддерживающие парадигму векторизации;
3. CRAN;
4. Плюсы и минусы языка R.

Лабораторная работа №5

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ (ИТЕРАЦИИ И КОНВЕЙЕРЫ)

Цель: познакомиться с особенностями функционального программирования. Научиться применять функциональное программирования с использованием пакета purrr. Решить задания в соответствующем стиле программирования. Составить отчет.

Теоретические сведения

Язык R поддерживает функциональную парадигму программирования, однако для лучшей ее реализации, и для увеличения функционала, рекомендуется использовать пакет purrr, кроме того, необходимо также скачать и подключить пакет repurrrsive, содержащий наборы данных для выполнения задания №1.

Итерации

Итерация – организация обработки данных, при которой действия повторяются многократно. В программировании итерации чаще рассматриваются в качестве элементов структурного программирования, а именно как единичный шаг выполнения цикла. На практике итерации также важны и в функциональном программировании, например в качестве итерабельного процесса можно рассматривать применение одной и той же функции к разным элементам. Рассмотрим пример выполнения кода, написанного в структурном и функциональном стиле над однотипным набором данных.

```
# Создание тестового списка
iris_list <- as.list(iris[1:4])

# Реализация в структурном стиле
iris_mean <- list()
for (i in seq_along(iris_list)) {
  iris_mean[[i]] <- mean(iris_list[[i]])
}

# Реализация в функциональном стиле через map
iris_mean <- map(iris_list, mean)
```

Рисунок 1 – Сравнение функциональной и структурной реализации

У функции map есть два варианта реализации (рисунок 2).

```
# Реализация map без указания точки входа в функцию
iris_mean <- map(iris_list, mean)

# Реализация map с указанием точки входа в функцию
iris_mean <- map(iris_list, ~ mean(.x))
```

Рисунок 2 – Варианты синтаксиса map

Второй вариант реализации позволяет показать место передачи элемента листа (“.x”) через знак тильда (“~”). Второй способ является предпочтительнее для реализации.

Вариации функции map. map_*

Функция map на выходе выдает список, аналогичной структуры, однако часто необходимо сразу преобразовать вывод в некоторый вариант, для этого используются производных от функции map:

```
map(.x, .f, ...)
map_if(.x, .p, .f, ...)
map_at(.x, .at, .f, ...)
map_lgl(.x, .f, ...)
map_chr(.x, .f, ...)
map_int(.x, .f, ...)
```

```
map_dbl(.x, .f, ...)
map_dfr(.x, .f, ..., .id = NULL)
map_dfc(.x, .f, ...)
walk(.x, .f, ...)
```

```r
# Создание именованного числового вектора с помощью map_dbl
iris_mean <- map_dbl(iris_list, ~ mean(.x))
```

Рисунок 3 – Создание именованного числового вектора с помощью map_dbl

Конвейеры (pipeline)

Конвейер – инструмент для передачи значения исходной функции в последующую. Конвейер представляет типичный элемент функционального программирования. В языке R конвейер имеет вид %>%. Синтаксис конвейера представлен на рисунке 4.

```r
function_before() %>%
  function_after()
```

Рисунок 4 – Синтаксис конвейера

Практическая часть

Задание 1. Используя тестовые данные пакета repurrrsive выполните следующее задание. Создайте именованный список аналогичный по структуре списку sw_films, для установления имени полезно использовать функцию set_names пакета purrr. В качестве имени элементов списка необходимо использовать соответствующие название фильмов (обратите внимание, что обращаться к элементам списка можно используя как индекс, так и название элемента). Выполните задание в функциональном стиле.

Задание 2. Используя документацию пакета purrr опишите отличия и особенности функций семейства map_*. Приведите примеры реализации с использование различных тестовых данных. Данные можно брать из пакета datasets или создав свои тестовые наборы. Для просмотра данных из пакета datasets выполните код library(help = "datasets")

Лабораторная работа №6
Грамотное программирование

Цель: познакомиться с особенностями грамотного программирования. Научиться применять грамотное программирование для создания динамических отчетов с использованием технологии R Markdown. Решить задания в соответствующем стиле программирования. Составить отчет.

Пример создания документа динамического документа

Для создания динамического документа в RStudio необходимо выбрать соответствующий пункт: New File > R Markdown. При первом создании файлов R Markdown потребуется установка соответствующего пакета. При создании покажутся соответствующие настройки (рис. 1), которые в дальнейшем можно будет изменить, используя YAML нотацию.

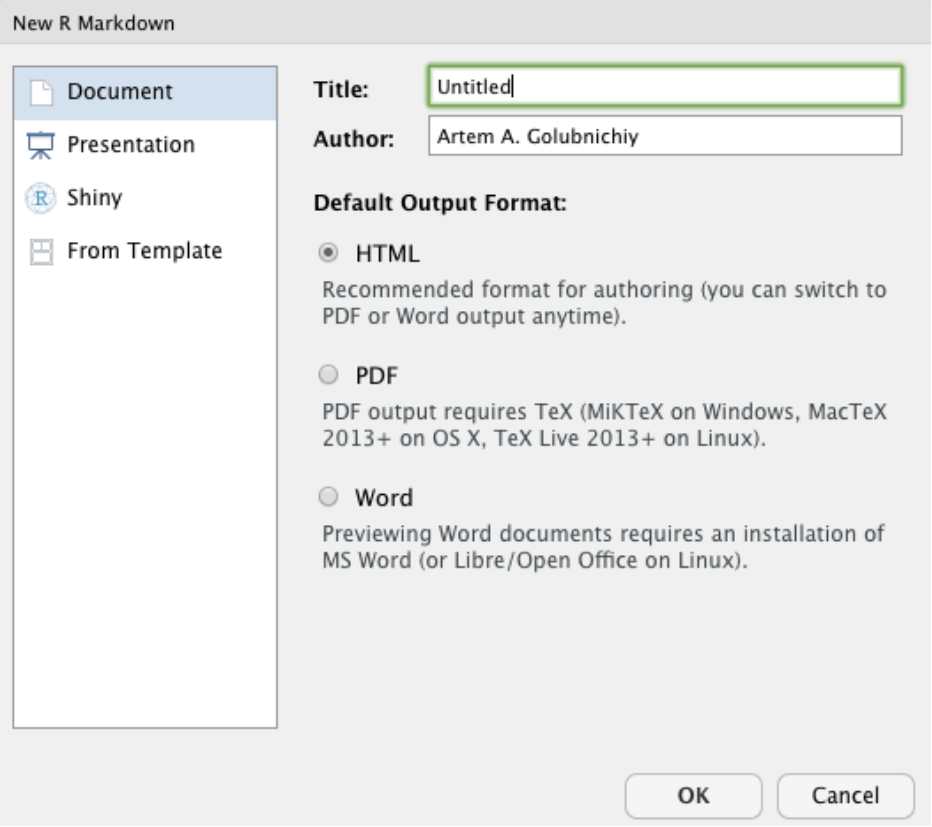


Рисунок 1 – Создание нового R Markdown документа

Если при генерации файлов используются дополнительные файлы такие как каскадные таблицы стилей *.css, то необходимо чтобы они располагались в той же директории, что и основной файл с кодом.

```
---
title: "Ozone"
output:
  html_document:
    css: faded.css
---

## Data

The `atmos` data set resides in the `nasaweather` package of the *R* programming language. It contains a collection of atmospheric variables measured between 1995 and 2000 on a grid of 576 coordinates in the western hemisphere. The data set comes from the [2006 ASA Data Expo](http://stat-computing.org/dataexpo/2006/).

Some of the variables in the `atmos` data set are:
```

```
* **temp** - The mean monthly air temperature near the surface of the Earth (measured in kelvins (*K*))

* **pressure** - The mean monthly air pressure at the surface of the Earth (measured in millibars (*mb*))

* **ozone** - The mean monthly abundance of atmospheric ozone (measured in Dobson units (*DU*))

You can convert the temperature unit from Kelvin to Celsius with the formula

$$ celsius = kelvin - 273.15 $$

And you can convert the result to Fahrenheit with the formula

$$ fahrenheit = celsius \times \frac{9}{5} + 32 $$

```{r, echo = FALSE, results = 'hide'}
example_kelvin <- 282.15
```

For example, `r example_kelvin` degrees Kelvin corresponds to `r example_kelvin - 273.15` degrees Celsius.
```

Рисунок 2 – Пример R Markdown файла

Для корректного применения таблицы стилей код, представленный на рисунке 3 должен быть сохранен в файл с названием faded.css.

```
h1{
  color: white;
  padding: 10px;
  background-color: #3399ff
}

ul {
  list-style-type: square;
}

.MathJax_Display {
  padding: 0.5em;
  background-color: #eaeff3
}
```

Рисунок 3 – Содержание файла таблицы стилей

Практическая часть

Задание 1.

Используя технологию R Markdown создайте динамический документ с произвольными расчетами. Документ должен содержать вставки кода по типу inline и в виде чанков. В документе должно быть использовано различное форматирование. Также для оформления используйте каскадную таблицу стилей. Итоговый документ конвертируйте в html формат и представьте в отчете, соответствующие скрины.

Лабораторная работа №7

Параллельное программирование

Цель: познакомиться с особенностями параллельного программирования. Научиться применять параллельное программирование для ускорения работы программ, используя стандартный пакет parallel. Решить задания в соответствующем стиле программирования. Составить отчет.

Теоретическая часть

Существует несколько методов для запуска кода параллельно. В языке программирования R, для распараллеливания выполнения кода можно использовать пакета из ядра языка parallel, так и другие пакеты, например foreach и future.apply.

Методы разделения задачи для решения в параллельном стиле:

1. Разделение на задачи. Каждую из задач можно выполнить независимо друг от друга. В данных задачах используются разные данные.

2. Разделение по данным. Данный метод встречается часто для обработки больших данных.

Примером решения задачи разделения по данным может быть нахождения суммы элементов ряда: 1 + 2 + 3 + … + 100. В данном случае ряд можно разбить на 4 части и посчитать их параллельно и независимо друг от друга:

```
sum(1:25) + sum(26:50) + sum(51:75) + sum(76:100)
```

С точки зрения использования памяти существует два принципиальных подхода для решения задач параллельных вычислений:

1. Использование общей памяти (рис. 1)

2. Использование разделенной памяти (рис. 2)

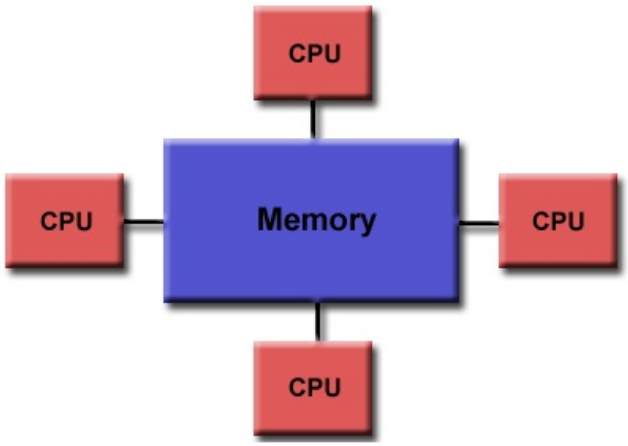


Рисунок 1 – Общая память

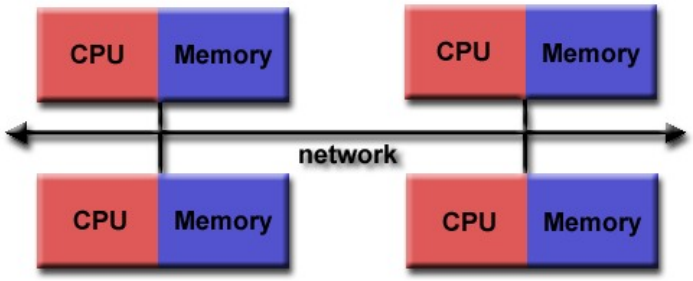


Рисунок 2 – Разделенная память

Существует два варианта реализации парадигмы параллельного программирования: модель «Master-worker» (мастер-работник), также встречается в названии «Master-slave» и модель Map-reduce (уменьшение карты).

На Map-шаге происходит предварительная обработка входных данных. Для этого один из компьютеров (называемый главным узлом – master node) получает входные данные задачи, разделяет их на части и передает другим компьютерам (рабочим узлам — worker node) для предварительной обработки.

На Reduce-шаге происходит свертка предварительно обработанных данных. Главный узел получает ответы от рабочих узлов и на их основе формирует результат – решение задачи, которая изначально формулировалась.

Модель Master-worker наиболее типичная для простых параллельных вычислений и имеет вид, представленный на рисунке 3.

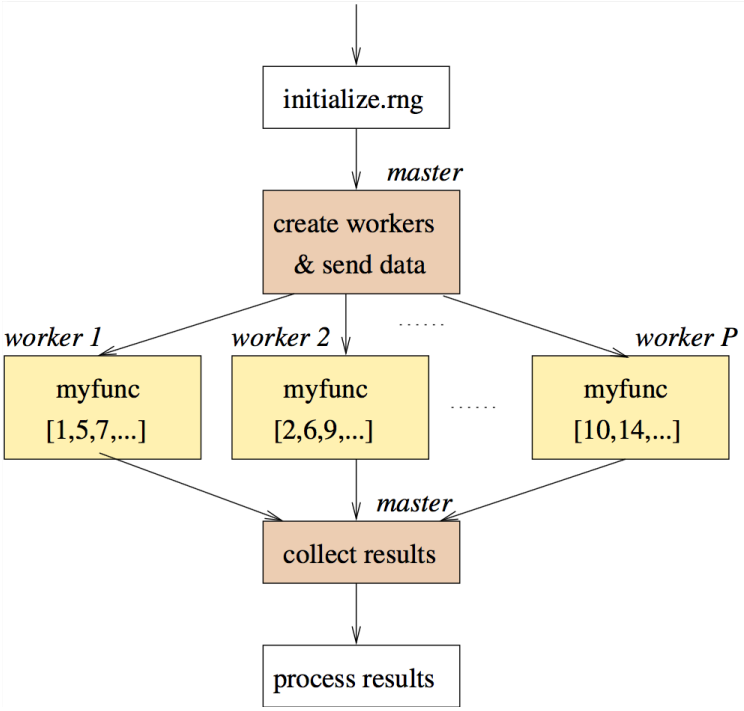


Рисунок 3 – Модель Master-worker

В самом простом виде реализация этой модели возможна с использованием пакета parallel, входящего в состав базового ядра языка R.

```r
# Подключение пакета, определение количества ядер
library(parallel)
ncores <- detectCores(logical = FALSE)
n <- ncores:1

# Создание кластера
cl <- makeCluster(ncores)

# Использование кластера с clusterApply для вызова функции rnorm для
# каждого n параллельно с параметрами mean = 10 и sd = 2
clusterApply(cl, n, rnorm, mean = 10, sd = 2)

# Остановка кластера
stopCluster(cl)
```

Рисунок 4 – Пример кода создания и использования вычислительного кластера

Практическая часть

Задание 1.

Используя заранее подготовленные функции визуализируйте сведения о наиболее часто встречающихся словах из книг Джейн Остин по буквам английского алфавита. Книги, необходимые для анализа, находятся в пакете janeaustenr. Также для работы потребуется пакет stringr. После установки пакетов добавьте следующие функции:

```
extract_words <- function(book_name) {
  text <- subset(austen_books(), book == book_name)$text
  str_extract_all(text, boundary("word")) %>% unlist %>% tolower
}

janeausten_words <- function() {
  books <- austen_books()$book %>% unique %>% as.character
  words <- sapply(books, extract_words) %>% unlist
  words
}

max_frequency <- function(letter, words, min_length = 1) {
  w <- select_words(letter, words = words, min_length = min_length)
  frequency <- table(w)
  frequency[which.max(frequency)]
}
select_words <- function(letter, words, min_length = 1) {
  min_length_words <- words[nchar(words) >= min_length]
  grep(paste0("^", letter), min_length_words, value = TRUE)
}
```

Рисунок 5 – Функции необходимые для решения задания

Для решения задачи необходимо с помощью функции janeausten_words() создать новый объект – вектор слов. Далее, используя одну из функций семейства apply, и заранее созданную функцию max_frequency создать именованный вектор, содержащий значение максимально встречающих слов английского алфавита, длиной не менее 5 букв.

Полезной для работы будет использование встроенной переменной letters, содержащей строчные буквы английского алфавита. Для визуализации используйте функцию barplot() с аргументом las = 2. В случае полностью правильного выполнения задания будет представлен график как на рисунке 6.

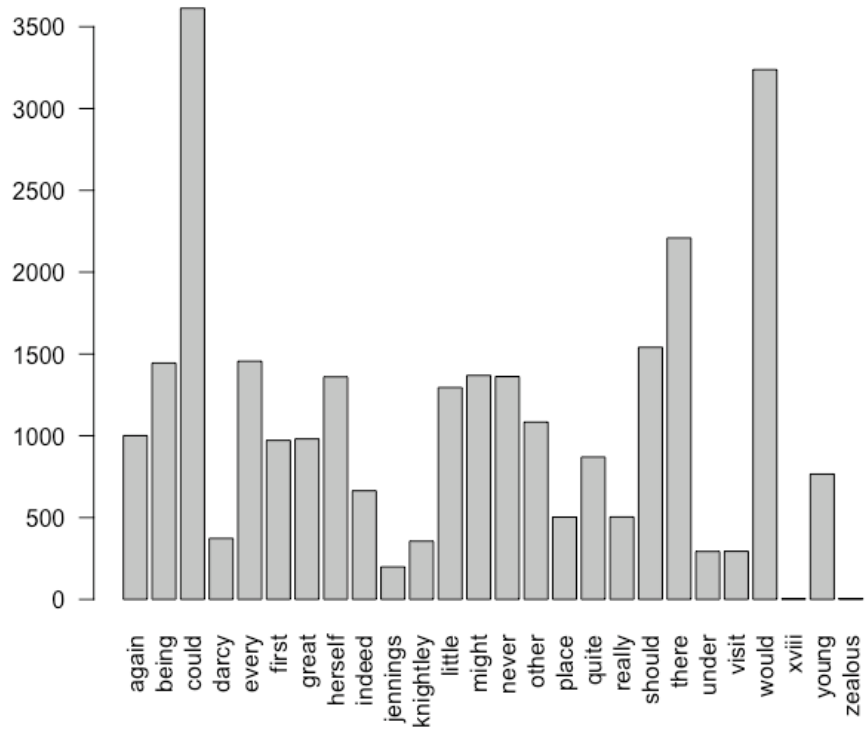


Рисунок 6 – Решение задания 1

Задание 2.

Распараллельте фрагмент кода, представленный ниже, используя вычислительный кластер:

```
for(iter in seq_len(50))
    result[iter] <- mean_of_rnorm(10000)
```

Для решения подгрузите функцию

```
mean_of_rnorm <- function(n) {
  random_numbers <- rnorm(n)
  mean(random_numbers)
}
```