ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ
Северо-Кавказский филиал
ордена Трудового Красного Знамени федерального государственного
бюджетного образовательного учреждения высшего образования
«Московский технический университет связи и информатики»

С. И. Конева

Кафедра Информатика и вычислительная техника

«ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ»

УЧЕБНОЕ ПОСОБИЕ, часть 1.

Ростов-на-Дону
2019 г.

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие, часть 1

Пособие предназначено для проведения занятий со студентами направления 09.03.01.

Составитель: Ст. преподаватель кафедры ИВТ Конева С. И.

В подготовке пособия принимали участие студенты: Гладыщук.С.В ДВ-31, Сакалова А.И. ДВ-21

Рецензент: Доцент кафедры ИВТ к.т.н. доцент Чикалов А. Н

Пособие обсуждено и одобрено на заседании

Кафедры ИВТ

Протокол №1 от 26.08.19

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Функциональное программирование — это ветвь программирования, в которой программирование ведется с помощью определения функций.

Программирование родилось в 1940-х г.г., когда велась разработка первых ЭВМ. Первые цифровые компьютеры в 40х годах 20 века программировались при помощи переключения тумблеров, кнопок, проводов. С ростом сложности программ число таких переключений достигло несколько сотен.

Следующим шагом развития программирования стало создание ассемблерных языков с простой мнемоникой, отражающий используемые машинные коды. Однако мнемокоды требовали от разработчика знания машинных кодов в шестнадцатеричной системе счисления, тем более что всякий ассемблер жестко связан с архитектурой технических средств, на которых он исполнялся.

Поэтому, следующим шагом после ассемблера стало развитие императивных языков высокого уровня (Pascal, C, Ada, C++, Java, включая объектно - ориентированные). Под императивным (от латинского повелительный) понимается такая модель (парадигма) программирования, в которой основным методом построения программ является задание последовательности шагов для исполнения. Вызовы функция и процедур не избавляет такие языки от императивности (предписания).

Принцип последовательного исполнения команд процессором стал препятствием для дальнейшего развития компьютерной техники. Скорость работы процессора стала зависеть не столько от его архитектуры и технических элементов, сколько просто от его размеров, т.к. на скорость работы процессора решающее влияние стала оказывать скорость прохождения сигналов по цепям процессора, которая не может превысить скорость света. При этом, чем меньше процессор, тем быстрее внутри него могут проходить сигналы. Однако с уменьшением процессора стало труднее

отводить от такого миниатюрного устройства вырабатываемое при работе его элементов тепло.

Перед производителем вычислительной аппаратуры возникла серьезная проблема: дальнейшее повышение производительности стало почти невозможным без изменения основополагающего принципа всего современного программирования — последовательного исполнения команд.

Проблема решалась разными способами:

1. Созданием специальной программы с фрагментами, которые выполнялись бы параллельно;

2. Переход к параллельным вычислениям - создание языка программирования, в котором сам алгоритм имел не последовательную структуру, допуская независимое исполнение отдельных частей алгоритма (функциональное программирование).

Функциональная программа представляет собой набор определения (деклараций) функций. Функции определяются через другие функции или рекурсивно — через самих себя. В процессе выполнения программы функции получают параметры, вычисляют и возвращают результат, в случае необходимости вычисляя значения других функций.

В начале 1960х годов появился первый язык программирования, поддерживающий функциональную модель программирования, LISP, непохожий на традиционные языки, для которого последовательность выполнения отдельных частей написанной программы была несущественной. Появилось множество диалектов этого языка, которые удовлетворили всем требования, необходимым для исполнения программ несколькими параллельными процессами.

Помимо своей хорошей приспособленности к параллельным вычислениям функциональное программирование обладает следующими свойствами:

- программы на этих языках записываются коротко, часто короче, чем в императивном языке;

- отдельные части программ могут исполняться независимо друг от друга;
- описание алгоритмов в функциональном стиле сосредоточенно не на том, как достигнуть нужного результата (в какой последовательности выполнять шаги алгоритма), а на том, что должен представлять этот результат.

LISP не чистый функциональный язык программирования. Это язык не типизирован, содержит массу императивных свойств, имеет множество диалектов. Язык LISP перестал удовлетворять многих разработчиков и в первую очередь из-за его непосильного синтаксиса — множества скобок.

В конце 70х начале 80х г.г. XX в. интенсивно разрабатываются и появляются множество типизированных функциональных языков:

ML, Hope, Miranda, Clean и многие другие. Функциональные языки разрабатывались для использования в узких рамках научных исследований. В результате вышло, что практически каждая группа разработчиков и исследователей, занимающаяся функциональным программированием, использовала собственный функциональный язык. Это препятствовало дальнейшему распространению этих языков.

Объединенная группа ведущих исследователей в области функционального программирования решала воссоздать достоинства различных языков в новом универсальном языке.

В числе разработчиков математических основ функционального программирования можно назвать. Мозеса Шенфинкеля и Хаскелла Карри, разработавших комбинаторную логику, а также Алонзо Черча, создателя λ - исчисления. Именно Х. Карри начал применять комбинаторную логику для описания вычислительных процессов.

Первая реализация функционального языка, названного Haskell в честь Хаскелла Карри, была создана в начале 90х годов 20 века. В настоящее время действителен стандарт Haskell-98.

Именно парадигма функционального программирования, как и логическое программирование, нашло большое применение в искусственном интеллекте и его приложениях.

Единственный недостаток функционального стиля программирования в том, что он не достаточно универсальный. Многие последовательные процессы игровые и другие программы, организующие взаимодействие компьютера с человеком, плохо поддаются их записи в функциональном стиле.

1 ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

1.1 Особенности функционального стиля программирования

Программа, написанная в функциональном стиле, представляет собой функцию, аргументом которой служат входные данные из допустимого набора, и выдающую определенный результат.

Любая программа, которая взаимодействует с пользователем, зависит от конкретных данных введенных пользователем. В императивном программировании при вызове одной и той же функции с одинаковыми параметрами, но на разных этапах выполнения алгоритма, получают разные данные из-за влияния на функцию изменение переменных. Это называется побочным эффектом. Особенности функций в функциональном программировании заключаются в том, что:

- каждая функция в программе выдает один и тот же результат на одном и том же наборе данных;
- вычисление функции не влияет на результат работы других функций, функции являются «чистыми»;

В функциональном языке при вызове функции с одними и теми же аргументами получается одинаковый результат. Выходные данные зависят только от входных данных. Это позволяет вызвать их в порядке, не определяемом алгоритмом, и распараллеливать их, что обеспечивает «чистые» функции.

Результат вызова «чистой функции» может быть сохранен в таблице значений вместе с аргументами вызова.

Если программа состоит только из чистых функций, то порядок вычисления аргументов этих функций будет несущественен.

При наличии нескольких независимых процессоров, работающих в общей памяти, вычисления, происходящие по программе, легко

распараллелить, увеличив количество процессоров, повышая скорость вычисления программ.

Можно составлять программы в функциональном стиле и на императивном языке программирования. Если программа состоит из набора чистых функций, то она будет «функциональной» независимой от того, написана она на языке функционального программирования или на императивном языке Java.

Например, в программе для вычисления приближенного числа ***e*** суммирование ряда Тейлора будет прекращено, когда очередной член ряда станет по абсолютной величине меньше ***eps*** (точность). Результат вычисления сводится к определению функции ***ex*** (x, eps).

Листинг 1.1 – Программа для вычисления приближенного числа ***e*** по формуле для разложения e^X в ряд Тейлора

```
Public static double ex (double x, double eps ) {
double ex = 0; // исходная сумма = 0
double u = 1; // промежуточное значение члена ряда
int n = 1; // номер очередного члена ряда
while (Math.abs(u) >= eps) { // вычисление очередного члена ряда, суммирование
ex  +=  u;
u = u * x  /  n;
n++;
 }
return ex;        //   результат работы
}
```

Здесь программа содержит повторяющиеся вычисления, записанные в виде цикла.

Для записи этой программы в *функциональном* стиле необходимо написать функцию, которая определит зависимость e^X от ***x*** и ***eps***.

Зависимость может быть выражена с помощью рекурсивного соотношения, которое вычисляет частичные суммы ряда.

Листинг 1.2 – Программа в функциональном стиле

```
public static double ex (double x, double eps) {
// вычисление с помощью вызова вспомогательной функции
    return exRec (x,  eps,  0,  1,  1);
}
public static double  exRec (double x, double eps, double sum, double u, int  n) {
return Math.abs (u) < eps  ?
sum:
exRec (x, eps, sum  +  u, u  *  x  /  n,  n  + 1);
}
```

Программа представляет набор функций, каждая из которых является суперпозицией других функций и операций. Основную работу выполняет рекурсивная функция exRec.

Далее представлены примеры программ вычисления корня уравнения $\cos x = x$ на языке Java и в чисто функциональном стиле. Вычисления выполнены методом половинного деления (бисекции). Этот численный метод требует знания интервала, где непрерывная функция имеет единственный корень. Функция $\cos x - x = 0$ имеет единственный корень на промежутке $[0, \pi / 2]$, что следует из графика функции. Значение точности вычислений *eps* будет аргументом основной функции.

Листинг 1.3 Определение корня функции методом бисекции на языке Java.

```
/* *
   *  cos  x  = x  на интервале (0,  pi/2 с заданной точностью.
   *  @param eps Точность вычислений ─ положительное число.
   *  @return  Приближенное значение корня уравнения.
   *  /
```

```
    static double  rootCos (double eps)     {
        return root (x  →  Math.cos (x)  – x,  eps,  0,  Math.PI  /  2 )  ;
    }
   /* * функция приближенного вычисления корня функции  f  на
        *   заданном интервале   (a,  b).
        *   @param   f     Функция, корень которой определяется.
        *   @param    a     Левый конец интервала, содержащего корень.
        *   @param   b     Правый конец интервала, содержащего корень.
        *    @return    Приближенное решение корня.
        * /
        public static double root (Function<Double, Double>  f, double eps,
                                                    double  a,  double  b)  {
        double   f a  =  f . apply (a) ;     // значения функции на левом конце.
        double  m,  fm;                      // вспомогательные переменные.
While  (b  –  a   >   eps)   {
     m  =  (a  +   b )  /  2;    fm  =  f . apply (m);
      if  (Math.signum (fa)  = =  Math.signum (fm) )  {
       a  =  m;   fa  =  fm;
     }  else  {
       b  =  m;
    }
  }
  return   a;
}
```

Если эту программу записать в функциональном стиле, то цикл следует заменить рекурсией. Решение состоит в том, что уточняются данные об интервале методом деления интервала пополам и выбирается тот, который содержит корень. Переменная *fa* является еще одним аргументом рекурсивной функции.

Листинг 1.4 – Уточнение корня функции методом половинного деления в функциональном стиле

```
static double root {
                Function<Double, Double> f,
                double eps, double a, double b, double fa) {
    final double m = (a + b) / 2;
    final double fm = Math.signum (f . apply (m) ) ;
     return b – a < eps ?          / / требуемая точность достигнута.
     a : fa = = fm ?
     root ( f, eps, m, b, fm) :
     root ( f, eps, a, m, fa) ;
     }
```

В решении рекурсия заменена циклом, а вместо условных операторов *if* используются условные выражения, представленные тернарным оператором (? :). В решении фактически не используются переменные. Константы f, m, fm просто помечены атрибутом *final*, чтобы подчеркнуть, что они обозначают вычисленные значения, но присваиваний в написанных функциях им не делается. Аргументы функций тоже являются константами, и им тоже можно было приписать атрибут *final*.

1.2 Система программирования Haskell Platform

В состав системы программирования на языке Haskell входят компилятор и интерпретатор программ, а так же оболочка для операционной системы Microsoft Windows.

Системы Haskell Platform можно установить с сайта https://www.haskell.org перейдя по ссылки https://www.haskell.org/platform/.

После установки системы можно сразу запустить интерпретатор выражений, записанных на языке Haskell, - GHCi (интерпретатор командной строки) или в системе Windows WinGHCi.GHC (Glasgow Haskell Compiler).

Функционально оба интерпретатора GHCi и WinGHCi, а также аналогичные интерпретаторы для других операционных систем совершенно одинаковы.

Программное средство GHC состоит из двух компонентов интерпретатора и компилятора. На деле для запуска того или иного режима используется просто разные параметры, а программный модуль для запуска существует один.

Команды, подаваемые интерпретатору это:

a) выражения для вычисления;
b) команды для управления работой интерпретатора;
c) получение информации о текущем вычислении, и получении отладочной информации;
d) установление режима работы;
e) загруженные программы и программные модули и т.д.

Полный список команд можно получить, набрав команду “:?”

В тексте пособия набор выражения в интерпретаторе, например, 2*2 будет отмечен следующим образом

```
>> 2 * 2
```

Результат вывода интерпретатора — 4.

Компилятор GHC вычисляет тип выражения с помощь команды:

```
: type
```

Сокращенно эта команда записывается

```
: t
```

Для нашего примера

```
>> : t  2 * 2
2 * 2   : :  Num a => a            - - ответ компилятора
```

Такая запись Num a => a – ответ компилятора, не Integer, означает, что переменная типа a – это любой тип, но с условием, что он должен принадлежать классу, для значений которого разрешены арифметические операции (класс Num).

С помощью команды

```
: set  +  t
```

можно включить опцию компилятора, чтобы показать тип вычисленного значения. Например, вывод интерпретатора после вычисленного выражения может быть таким:

```
>> 3.14  +  2 * 2
7.140000000000001
It  : :  Double
```

Операции присваивания в функциональном программировании нет. С помощью команды let a = 10, например, даем значению 10 имя a.

2 ОСНОВНЫЕ КОНСТРУКЦИИ ЯЗЫКА HASKELL

2.1 Вызов функции (применение функции)

Функциональная программа на языке Haskell состоит из набора определений значений и функций. Загрузив программу в интерпретатор, можно выполнить действия по вычислению результатов, просто вызывая функции (правильно сказать «применяя функции»), определение которых заданы в программе.

Для примера составим программу для вычисления площади треугольника по формуле Герона.

Определим функцию triangle с аргументами a, b, c и вычислением полу периметра p треугольника

1. Запишем программу в текстовый файл с именем triangle.hs triangle a b c = let p = (a + b + c) / 2 in sqrt (p *(p-a)*(p-b)*(p- c)). Полное имя файла в системе Windows будет c:\Haskell\triangle.hs
2. Запустим интерпретатор и подадим команду на загрузку и проверку программы: >> : load C:\Haskell\triangle.hs. При правильной записи имени файла (без пробелов, буквами латинского алфавита) на экране будет выведено сообщение интерпретатора с именем Main модуля (по умолчанию). Модуль загружен, скомпилирован и готов к выполнению.
3. Теперь можно применить функцию вычисления площади треугольника и получения ожидаемого результата

   >> triangle 3 4 5

   6.0

2.2 Система типов языка Haskell

Первая стандартная версия языка Haskell появилась в 1998г. Haskell 98. язык сохранился без существенных изменений и в следующих версиях Haskell 2010, Haskell 2012.

Haskell – строго типизированный язык. Любая конструкция в языке имеет определенный тип, который можно увидеть до начала выполнения программы статически.

Языком без строгой типизации является первый язык функционального программирования Лисп.

Основу системы типов языка Haskell составляют элементарные встроенные типы данных.

Следует обратить внимание на то, что все идентификаторы в Haskell чувствительные к регистру букв. Если идентификатор начинается с заглавной буквы, то он обозначает встроенный или определенный программой тип или класс. С заглавных букв начинаются имена модулей или пакетов.

Идентификатор объектов – значений простых и сложных типов, в том числе функций – начинается со строчной буквы или символа подчеркивания.

Идентификаторы строятся из букв, подчеркивания, цифр, и символа апострофа « ' ». Ни апостроф, ни цифра не могут быть первым символом идентификатора.

Апостроф применяется для построения вспомогательных имен функций, связанных по смыслу с исходной функцией, например, triangle’ или triangle’ ’. В языке Haskell описание типа функции так же, как и в математической нотации.

Символ (- >) кодируется в виде последовательности символов (– >). Так написание типа некоторой функции someFc, которая принимает на вход два целочисленных аргумента и возвращающая вещественное число, будет выходить так:

someFc : : Integer –> Integer –> Double

В языке Haskell используется символ : : , который читается как «имеет тип».

В качестве базовых типов используются следующие типы.

Целые - представлены двумя типами Integer и Int. Тип Integer определяет бесконечный набор целых чисел произвольной длинны. Аналог – BigInteger в языке Java.

В пределах выделенной для работы программы памяти в операциях над целыми типа Integer никогда не происходит переполнения. Для повышения эффективности работы программ используются традиционные целые ограниченной длины. Для них применяется идентификатор типа Int – для представления целых чисел [-2147483648, 2147483648].

Тип для предоставления строк символов – Char.

Вещественные числа представлены двумя типами Floate и Double. Они имеют традиционные литеральные обозначения, такие как 3.14, 2.71828, 0.12e-10.

Принято, отрицательное число заключать в скобки, чтобы избегать двусмысленности. Так, выражение 3 * -1 некорректное, следует записать 3 * (-1).

Символьный тип также имеет литеральное обозначение следующего вида 'a'. Это обозначение представляет символ a.

Логический тип представлен двумя значениями – истина и ложь. Литеральных обозначение нет, но имеются два стандартных идентификатора True и False, обозначающие истинное и ложное значение соответственно.

В программе возможно введение собственных идентификаторов для имеющихся значений. Это эквивалентно средствам описания констант в разных языках программирования, например, stud = 500. Здесь константе 500 введен идентификатор stud. Хотя уточнение значения типа идентификатора не обязательно, т.к. система программирования Haskell сама устанавливает (выводит) типы всех встречающихся объектов и выражений, но иногда это существенно.

Литерал 500 может, обозначен как целое типа Int, так и Integer, и Float и Double в зависимости от контекста.

Явное указание типа stud : : Integer устраняет неоднозначность.

Проверка типа с помощью интерпретатора GHCi системы Haskell Platform даст результат:

```
>> : : t 500
500 : : Num a => a
```

Выражение 500 имеет неопределенный тип *a* при условии, что тип принадлежит классу Num.

Класс определяет набор операций (функций), которые можно производить над объектами, принадлежащих этому классу.

Полный список разрешенных операций для класса Num можно получить набрав команду : info.

```
>> : info Num
class Num a where
(+) : : a → a → a
(*) : : a → a → a
(-) : : a → a → a
negate : : a → a
abs : : a → a
sigrum : : a → a
fromInteger : : Integer → a
```

Выполнение операции сложения (+) над двумя значениями принадлежащих классу Num приведет к получению значения того же самого типа, что и типы операндов.

Функция fromInteger, входящая в состав класса Num, позволяет сделать явно преобразование типов.

2.3 Объединение в кортежи

Значение объектов произвольных типов можно объединять в кортежи. Объединяемые в кортежи значения просто перечисляются в скобках через запятую.

Например, кортеж (3, ' b ') представляет собой кортеж из двух значений — целого числа 3 и символа ' b '.

Тип кортежа, для примера, в котором введено обозначение кортежа,

prim : : (Int, Char)

prim : : (3,' b ')

Естественно, кортежи могут входить в состав других кортежей. Также бывает пустой кортеж ( ) , который не содержит ни одного элемента. Ясно, что ( ) - это не функция, а скорее значение и тип этого значения ( ).

2.4 Особенности записи выражений

Над значениями можно выполнять определенные в языке операции и применять функции.

Полный набор стандартных функций и операций имеется в документах по языку. В языке имеется обычный набор арифметических и логических операций, операций сравнения и функцией преобразования значений из одних типов в другие.

Однако имеются две особенности записи выражений:

а) при вызове функций аргументы отделяются от идентификатора функции пробелом, а если аргументов несколько, то и сами аргументы отделяется друг от друга пробелами.

Например, sin 0.5

Функцию определения максимального из двух значений можно вызвать:

max 5 (x+1) (x – некоторое числовое значение)

Здесь скобки определяют порядок действия.

Если бы их не было, то интерпретатор прочитал его (max 5 x) + 1, т. к. операция применения функции к аргументу считается более приоритетной. Поэтому можно записать без скобок

sin x * cos y – sin y * cos x

Операция применения функции к аргументу имеет и собственный знак операции **$**, но он имеет самый низкий приоритет.

В эквивалентной записи скобок опускать нельзя

(sin $ x) * (cos $ y) – (sin $ y) * (cos $ x)

Выражение sin (x + 2 * y) можно записать без скобок sin $ x + 2 * y;

б) Вызов бинарной операции можно записать как обычную двуместную функцию. Для превращения знака бинарной операции в двуместную функцию нужно заключить знак операции в скобки

Например, (+) x y запись эквивалентна записи

x + y

Для того, чтобы превратить обычный идентификатор функции двух аргументов в бинарный оператор, следует заключить идентификатор функции в обратные апострофы . Поэтому функцию определения максимального из двух значений можно записать 5 'max' (x+1).

Даже операцию объединения значений в кортежи можно записать в виде префиксной функции. Вместо (3, True, ' a ') можно написать эквивалентное выражение (, ,) 3 True 'a'

Существуют стандартные функции для образования кортежей: (,) , (, ,) , (, , ,) и т.д.

2.5 Определение и вычисление выражений в Haskell

В функции описывается процесс вычисления. Функция в Haskell – это некоторое значение, для которого определен тип.

В типе функции указывают типы ее аргументов и тип значения функции.

При этом типы аргументов и значения функции отделяются друг от друга символом «→». Например, тип функции с двумя целыми аргументами и одним логическим результатом может быть записан в виде Int → Int → Bool. Общий вид определения функции в языке Haskell имеет вид:

<function_name> <patterns> “ = ” <expression>

Наименование функции является правильным идентификатором с точки зрения языка.

Терм expression – это некоторое выражение, правильно построенное с точки зрения синтаксиса языка, patterns – образец.

Таким образом, функцию можно определить с помощью уравнения, в котором указывается, как функция должна себя вести, если ей задать значение аргумента.

```
tri : : Double ─> Double
tri x = 3.2 + x
```

Вызов этой функции с заданным значением аргумента эквивалентен выражению, в которое к этому значению прибавляется 3.2

Левая часть уравнения задаёт форму вызова функции. Правая часть, в которой записывается выражение, заменяет вызов функции в тот момент, когда будет задано значение её аргумента.

```
>> tri 12
15.2
it : : Double
```

Сначала происходит сопоставление фактического значения аргумента 12 с формальным аргументом x. Затем вызов функции заменяется правой частью управления, в которой вместо формального аргумента используется фактический аргумент. После сопоставления и замены вместо выражения tri 12, получим выражение 3.2 + 12, которое после применения операции сложения дает значение 15.2. Процесс преобразования (вычисления) выражения можно записать:

tri 12 —> 3.2 + 12 —> 15.2

Число 12 здесь сразу рассматривается как вещественное (нет преобразования целого числа в вещественное, как в Java), поскольку контекст выражения tri 12 требует, чтобы аргумент функции tri был вещественным.

Преобразование выражений, называют редукциями.

Вычисление выражений в Haskell (исполнение программы) осуществляется с помощью последовательных редукций исходного выражения.

Вызов функции – это применение функции к аргументам.

Примитивными функциями являются все арифметические операции. Редукция выражений, содержащих примитивные операции, осуществляется за один шаг, без сопоставления и подстановки.

Определение функции может быть рекурсивным, т.е. в правой части уравнения может быть вызов определяемой функции.

Для того, чтобы процесс вычисления мог закончиться, необходимо использовать условные выражения.

Условное выражение имеет вид:

If <условие> then <выражение-«то»> else <выражение-«иначе»>

При вычислении условного выражения сначала вычисляется условие. Тип вычисленного выражения–условия должен быть логическим (Bool). Аналогично, как в императивных языках программирования, если вычисленное значение оказывается истинным (True), то вместо всего условного выражения подставляется выражение – «то», а выражение – «иначе» отбрасывается. Если же наоборот, то отбрасывается выражение – «то», остается только часть, определенная выражением – «иначе».

Например, определенные простой рекурсивной функции вычисления факториала целого числа.

```
factorial : : Integer —> Integer
factorial n = if n = = 0 then 1 else n * factorial (n-1)
```

Вместо применения условного выражения можно использовать запись уравнения с «охраной» (сторожами, guards).

Охрана или охраняющее выражение — это логическое выражение (то есть возвращающее значение тип bool), которое накладывает некоторые ограничения на переменные.

Охрана может использоваться в языке Haskell в качестве дополнения к сопоставлению с образцом.

Выражения (охрана) проверяются последовательно, и в качестве выражения для выполнения редукции выбирается первое из тех, в которых охрана выдала истинные значения условия.

При определении функций охраняющие выражения записываются после образцов (аргументов), но перед выражениями, описывающие вычислительный процесс.

Для разграничения охраняющих выражений и образцов используется символов (|).

Листинг 2.1 – Программа вычисления факториала натурального числа

```
factorial : : Integer ─> Integer
factorial  n  | n = = 0  =  1
              | n  >  0  =  n  *  factorial  (n-1)
```

При задании отрицательного значения аргумента ни один из двух сторожей (охрана) не будет истинным. Значение функции для данного значения аргумента не определяется, функция закончится аварийно, и интерпретатор выдаст сообщение об аварийном завершении работы программы.

```
>>  factorial  ( -2 )
*** Exception: test. hs (2, 1) – (3, 41) : Non - exhaustive patterns in
function  factorial
```

Для определения функции можно записать несколько уравнений, определяющих функцию при различных значениях аргумента.

```
factorial  : :  Integer ─> Integer
factorial  n  |  n = =  0  = 1
factorial  n  |  n > 0  =  n * factorial (n-1)
```

Если первое уравнение применить не удастся (ни один из сторожей не стал истинным для заданного значения аргумента), то проверяется второе уравнение. Или будет найдено уравнение, в котором один из сторожей «сработает», либо работа функции завершится аварийно.

Можно сразу задать случай, когда аргументом вызова будет нулевое значение.

```
factorial  : :  Integer ─> Integer
factorial  0 = 1
factorial  n = n  *  factorial (n-1)
```

Листинг 2.2 – Определение функции, сравнивающей два значения и возвращающей строку с текстом относительно равенства этих значений

```
comparison  x   y   | x  <  y     = “Первое значение меньше второго.”
                    |  x  >  y    =  “Первое значение больше второго.”
                    | otherwise  = “Значения равны.”
```

Otherwise означает «иначе» или «в противном случае».

Язык Haskell содержит и набор средств выходящих за рамки функционального программирования. К ним относится функция ***error*** и функции ввода – вывода. Рассмотрим определение функции, которая по заданным натуральным числам определяет их наибольший делитель согласно алгоритму Евклида. Напомним, что идея Евклида состоит в том, что если ни одно из чисел не равно нулю, то наибольший общий делитель равен наибольшему общему делителю наименьшего из них и остатка от деления большего на меньшее. Поскольку остаток от деления одного натурального

числа на другое можно вычислить с помощью стандартной бинарной операции ***mod,*** то программа примет вид:

Листинг 2.3 – Наибольший общий делитель натуральных чисел.

```
gd : : Integer —> Integer —> Integer
– – если первый аргумент меньше второго, то меняем их местами
gd  m  n | m  <  n  =  gd  n  m
– – функция определена только для неотрицательных аргументов
            | n  <  0  =  error   “gd : Negative argument”
– – алгоритм Евклида:
gd  m   0  =  m
gd  m   n  = gd   n   (m  ' mod '   n)
```

В состав стандартных функций языка Haskell входит функция gcd (greatest common divider) – наибольший общий делитель. Аргументы этой функции не ограничены натуральными числами, поэтому задавать в качестве аргументов отрицательные значения можно.

2.6 Концевая рекурсия и накапливающие аргументы

Рассмотрим последовательность чисел Фибоначчи.

Известно, что первые два числа в последовательности чисел Фибоначчи равны единице, а каждое из следующих чисел равно сумме двух предыдущих.

Пример нахождения числа ***n*** с заданным номером в последовательности чисел Фибоначчи.

Листинг 2.4 – Нахождение числа Фибоначчи по его номеру в последовательности

```
fib : : Integer —> Integer
fib  1  =  1
```

fib 2 = 1

fib n = fib (n-1) + fib (n-2)

Это решение неэффективно. В главном уравнении производятся два рекурсивных обращения к этой же самой функции. Из-за этого число обращений к функции резко возрастет при увеличении значения аргумента. Для примера, рассмотрим изменение выражения fib 5 при последовательных рекурсиях.

fib 5

fib 4 + fib 3

(fib 3 + fib 2) + fib 3

((fib 2 + fib 1) + fib 2) + fib 3

((1 + 1) + 1) + (fib 2 + fib 1)

((2 + 1) + (1 + 1)

3 + 2

5

Увеличивается и количество памяти, требуемое для хранения выражения.

Для получения эффективного решения следует добавить в функцию дополнительную информацию, чтобы при каждом вызове не находить ранее найденные значения чисел Фибоначчи.

Например, если вычислены первые ***k*** чисел Фибоначчи, и имея значение хотя бы двух последних вычисленных чисел можно определить функцию, которая вычисляет число Фибоначчи с последним n. Такая вспомогательная функция будет иметь четыре аргумента:

n – номер числа Фибоначчи;

k – номер последующего вычисляемого числа Фибоначчи;

fk – значение к-го числа Фибоначчи;

fk1 – значение предыдущего, (k-1)-го числа Фибоначчи.

Листинг 2.5 – Программа вычисления чисел Фибоначчи

```
– – fib – основная функция вычисления n-го числа Фибоначчи.
fib : : Integer ─> Integer
– – fib' – вспомогательная функция.
fib' : : Integer ─> Integer ─> Integer ─> Integer
fib'  n  k  fk  fk1  | k = = n  =  fk
                     | k  <  n  =  fib'  n  (k + 1)  (fk  +  fk1)  fk
fib 1 = 1
– – первоначально известны первые два числа Фибоначчи:
fib n  =  fib'  n  2  1  1
```

Описанный вариант программы мгновенно вычисляет, например, 100-е число.

В этом примере аргументы вспомогательной функции fib' увеличивали значение от вызова к вызову, накапливая результат. Такие аргументы играют роль обычных переменных в императивных языках программирования. Они сохраняют информацию о вычисленных значениях при последовательном исполнении программы. Иногда их называют накапливающими аргументами функции.

Пусть требуется вычислить сумму числового ряда для получения приближенного значения трансцендентного числа ***e***. Аргумент основной функции – наибольшее положительное число, задающего величину последнего суммируемого члена ряда. Здесь требуется несколько накапливающих параметров для хранения значения очередного члена ряда, его номера, накопленной суммы.

Листинг 2.6 – Вычисление трансцендентного числа ***e***

```
– – Основная функция.
– – Аргумент:  eps – точность вычислений.
– – результат – вычисления числа e
e  : : Double ─> Double
```

– – Вспомогательная функция с дополнительными аргументами

– – Аргументы : eps – точность

– – n – номер последнего вычисленного члена суммы.

– – mem – значение последнего вычисленного члена суммы.

– – sum – накопленное значение суммы.

```
e' : : Double —> Double —> Double —> Double —> Double
e  eps = e' eps 1 1 0
e' eps n mem sum | mem < eps = sum
                 | otherwise = e' eps (n + 1) ( mem / n) (sum + mem)
```

Рекурсивный вызов можно заменить на циклическое вычисление, если в теле функции рекурсивное обращение к ней является последним исполняемым в нём вызовом в правой части уравнения. Такая рекурсия называется концевой рекурсией.

Листинг 2.7 – Использование концевой рекурсии при вычислении факториала числа

```
factorial :: Integer → Integer
factorial' Integer → Integer → Integer
factorial' of = f
factorial' kf = factorial' (k-1  (k*f)
factorial n = factorial' n 1
```

Таким образом, любую рекурсивную функцию можно преобразовать таким образом, чтобы все рекурсивные вызовы в ней были концевыми.

Приведенное определение функции с использованием накапливающего параметра близко к стандартному определению функции в традиционном языке программирования с использованием цикла.

3 СПИСКИ В HASKELL

Способов построения массивов в языке Haskell нет. В Haskell имеется способ строить структуры данных произвольной степени сложности из более простых объектов в виде списков.

Списки – это последовательность объектов одного и того же типа, состоящая из произвольного числа объектов. Список называется пустым, если в нем нет ни одного объекта.

3.1 Тип списка и задание объектов списков

Тип списка определяется типом его компонентов и в языке Haskell обозначается [T].

[Integer] будет обозначать тип списка, содержащего в качестве компонентов целые числа.

[([Char] , Double)] – компонентами списка являются кортежи из двух полей, первым полем в каждом кортеже – список символов, а вторым полем – вещественное число.

Объекты – списки задаются перечислением своих компонентов в квадратных скобках через запятую. Например, [1, 2, 5, 6] . Типом этого списка будет Num t => [t]. Запись, в которой компонентами списка будут объекты разных типов, синтаксически неверна. Так [2, 3, ‘*’] не может служить изображением списка.

Для списков символов в языке также есть специальный идентификатор типа – String.

Список [(”pi ”, 3.14), (“e”, 2.72)] является значением типа [(String, Double)].

Задание списка чисел, образующих арифметическую прогрессию. Если прогрессия равна единице, то можно указать только первый и последний

элементы списка, разделив их символом из двух точек. Например, [1 .. 10] – список из 10 последовательных натуральных чисел от 1 до 10.

Если шаг прогрессии не равен единице, то указывают два первых элемента списка и последний элемент.

Изображение списка [1, 3 . . 9] задает список из пяти целых чисел 1, 3, 5, 7, 9.

Изображение [1 . . n] применяется для создания списка первых натуральных чисел от единицы до его текущего значения n.

Списки могут создаваться в программе путем присоединения последовательно элементов к началу списка с помощью операции – конструктора списков. Конструктор списков имеет изображение в виде « **:** » (двоеточия).

Например, если x – список из целых чисел, равный [3, 4, 5], то конструкция 2 : x создаст новый список из элементов 2, 3,4, 5.

В конструкции 2 **:** 3 **:** 4 **:** [] построение списка происходит «справа налево», константа [] изображает пустой список.

В языке Haskell имеется много встроенных в язык операций обработки списков.

Некоторые, наиболее важные операции:

- head, last – функции, по заданному аргументу вызывают его первый и последний элементы соответственно. Эти операции не применимы к пустому списку;
- !! – операция, которая по списку и заданному номеру элемента выдает соответствующий элемент списка. Номер должен лежать в пределах от нуля до количества элементов списка без единицы.

  Например, вычисление [1, 2, 3, 4] !! 2 выдает в качестве результата число 3, т.к. первый элемент списка, имеет номер ноль.
- null – функция проверки пустоты списка. Если, в качестве аргумента этой операции – пустой список, то функция выдаст значение True, иначе False;

- length – функция вычисления длины списка (количества элементов в списке). Если аргумент имеет пустой список, то результат функции будет ноль;
- ++ – операция соединения двух списков. Из двух списков из элементов одного и того же типа получается новый список, составленный из элементов первого списка, за которым следуют элементы второго списка. Эта операция используется для соединения (катенции) двух строк.

Рассмотрим вычисление суммы элементов списка, получающий список из целых значений в качестве аргумента и выдающей их сумму. Оформим программу в виде функции sumList.

Листинг 3.1 – Суммирование элементов списка

```
– – Функция суммирования элементов целочисленного списка.
sumList : : [Integer] ─> Integer
sumlList s | null s = 0
           | otherwise = head s + sumList (tail s)
```

Здесь использованы встроенные функции head и tail. Это можно сделать с помощью аппарата сопоставления с образцом. В уравнении для определения функции записывается «образец» – формальное выражение с конструктором, в котором отдельные части списка – его первый элемент и остаток – обозначены отдельными идентификаторами. Второе уравнение в определении функции sumList может выглядеть так:

```
sumList (x : xs) = x + sumList xs
```

Идентификатор x обозначает значение головы, а xs – хвост списка. В правой части уравнения эти идентификаторы используются при рекурсивном вызове функции и в операции сложения целых.

Тогда второй вариант текста функции будет выглядеть так:

Листинг 3.2 – Второй вариант суммирования элементов списка

```
– – Функция суммирования элементов целочисленного списка.
```

```
sumList  : : [Integer] —> Integer
sumList [ ]   =   0
sumList   (x : s)  =  x   +   sumList   s
```

Следует отметить, что использование встроенных функций last и init удлинит время работы по сравнению с функциями head и tail, а функция init еще и будет строить заново список из всех начальных элементов.

Во всех функциях используется механизм сопоставления аргумента с образцом. Если образец состоит из имени переменной, то сопоставление будет успешным и значение аргумента будет сопоставлено с этой переменной. Если образец состоит из константы (например, 0 или пустого списка []), сопоставление будет успешным в том случае, когда фактический аргумент представляет собой эту константу.

Если функция определена с помощью серии уравнений, то сопоставление с образцами продолжится до тех пор, пока не произойдет успешного сопоставления. Тогда, при условии, что хотя бы один из сторожей выдал истинный результат, в правую часть уравнения будут подставлены результаты успешного сопоставления с образцом, и редукция будет завершена.

Аргументами встроенных функций могут быть любые списки независимо от типов их элементов, но типы списков должны соответствовать друг другу.

Например, в операции соединения списков (++) аргументы должны быть списками и элементами одного и того же типа.

Запись типа функции length:

```
length  : :  [a] —> int
```

Аргументом функции length может быть произвольный список, а результатом работы функции может быть короткое целое число.

```
( ++ )  :  [a] —> [a] —> [a]
```

Здесь два операнда этой операции должны быть списками элементов одного и того же типа. Результат будет список с элементами того же типа.

Функции, в определении типа которых участвуют переменные типа, называются полиморфными. Полиморфная функция может иметь аргументы типа, которой не определен в момент описания функции, а определяется только в момент применения этой функции к конкретному аргументу.

Итак, типы переменных встроенных операций:

head, tail : : [a] —> a

init, last : : [a] —> [a]

null : : [a] —> Bool

(!!) : : [a] —> Int —> a

sum, product : : Num a —> [a] —> a

Функция sum является обобщением функции sumList. Функция **product** вычисляет произведение элементов списка (произведение элементов пустого списка равно 1).

С помощью функции **reverse**, которая получает список в качестве аргумента, можно получить список с теми же элементами, но расположенными в этом списке в обратном порядке (обращение списка). Собирать список придется не с помощью конструктора списков, а посредством операции соединения списков.

Листинг 3.3 – Применение функции reverse

– – обращение списка

reverse : : [a] —> [a]

reverse [ ] = [ ]

reverse (x : s) = reverse s ++ [x]

Время работы функции пропорционально квадрату длины списка-аргумента.

Задачу можно решить за меньшее время, используя накапливающие аргументы.

Листинг 3.4 – Новый вариант функции обращения.

```
– – обращение списка
reverse : : [a] —> [a]
reverse s = reverse'  [ ]  s
– – дополнительный аргумент вспомогательной функции reverse'
– – служит для накопления результата.
reverse' : : [a] —> [a] —> [a]
reverse' s [ ] = s
reverse'  s1  (x : s2)  =  reverse'  (x : s1)   s2
```

Позже будет рассмотрена возможность использования функции reverse, определение которого есть в стандартной библиотеке. Над списками можно производить и следующие полезные операции:

- take – строит список из заданного количества первых элементов списка-аргумента.

  Например, take 4 [3, 1, 6, 5, 8] – значением этого выражения будет список [3, 1, 6, 5]. Если в качестве первого аргумента передать ноль или отрицательное значение, то в результате получится пустой список. Если первый аргумент больше, чем длина второго аргумента, то результатом будет весь исходный список;
- drop – аналогичен take, но отбрасывает первые элементы. В результате вычисления drop 4 [3, 1, 6, 5, 8] получится список из одного элемента [5].

  drop n list при отрицательном или нулевом значении n выдает второй аргумент list без изменений, а при n > length list выдает пустой список;
- splitAt – представляет комбинацию take и drop, выдавая и префикс, и постфикс заданного списка в виде кортежей из двух списков. Например, splitAt 4 [3, 1, 6, 5, 5, 8] будет пара ([3, 1, 6, 5], [8]).

Генератор списков позволяет строить списки из элементов других списков, выполняя над исходными элементами те или иные операции.

Конструкция генератора списков:

[<выражение> | <источники> , <условия>],

где <источники> – конструкции для порождения элементов исходных списков, имеющие вид

<образец> ← <список>

например, x ← [1..5]

<условия> – выражения над элементами образца, вычисляющие логические условия над элементами образца, определяющие, нужно ли использовать этот элемент для построения нового списка.

Например, [x * x | x ← [1..100], x mod 3 / = 0, x mod 5 / = 0]

Задает список квадратов чисел из первой сотни, которые не делятся ни на 3 ни на 5.

Если источников несколько, тогда для построения списка берутся все комбинации элементов из всех источников. Например, построение списка всех двухбуквенных слов в заданном алфавите в заданном алфавите alfa с помощью такой конструкции:

[[a, b] | a < – alfa, b < – alfa]

Результат:

>> let alfa = ”abc”

>> [[a, b] | a < – alfa, b < – alfa]

[“aa”, “ab”, “ac”, “ba”, “bb”, “bc”, “ca”, “cb”, “cc”]

Рассмотрим стандартную функцию обработки списков ***map***, которая, получив в качестве аргументов список и функцию преобразования элементов этого списка, выдает в качестве результата список из преобразованных элементов, полученных применением функционального параметра к каждому элементу списка. Например, функция sqr возводит аргумент в квадрат, а с помощью функции ***map*** можно получить список квадратов этих элементов списка.

map sqr [1, 2, 4, -3] —> [1, 4, 16, 9]

Исходные элемента могут иметь произвольный тип ***a***, если функция, являющаяся первым аргументом функции map, имеет тип (a → b), то результирующий список будет содержать элементы типа b.

Листинг 3.5 – Определение функции ***map***

```
map : : (a —> b) —> [a] —> [b]
– – результат применения map к пустому списку
– – пустой список
map _ [ ] = [ ]
map func (x : s) = (func x) : (map func s)
```

Первое уравнение определяет результат работы отображения в случае пустого списка. В этом случае первый аргумент функции не применяется вовсе, т.к. результатом будет пустой список независимо от того, какая функция применяется для отображения его элементов.

Второе уравнение использует рекурсивное обращение к функции map для того, чтобы построить отображение хвоста списка, а затем присоединяет к результату головной элемент, который получается с помощью применения к первому элементу исходного списка отображающей функции. Функция map, которая в качестве аргумента получает функцию или выдает функцию в качестве результата, называется функционалом.

Листинг 3.6 – Вычисление списка простых сомножителей списка чисел

```
– – функция factors выдает упорядоченный список простых
– – сомножителей заданного натурального числа
factors : : Integer —> [Integer]
factors n = reverse $ factors' n 2 [ ] where
– – Вспомогательная функция factors' имеет два
– – накапливающих аргумента – очередной проверяемой делитель d и
– – уже вычисленный список простых сомножителей, меньших d.
factors' 1 _ list = list
```

```
factors' n d list
        | d * d > n = add n list
      | n 'mod' d = = 0 = factors' (n 'div' d) d (add d list)
      | otherwise = factors' n (d + 1) list
– – функция add добавляет очередной сомножитель
– – в начало списка, так что окончательный список оказывается
– – перевернутым.
add d [ ] = [ d ]
add d list @ (x : xs) | d = = x = list
                      | otherwise = d : list
– – функция merge сливает без повторений два
– – упорядоченных списка целых в один
merge : : Ord a => [a] —> [a] —> [a]
merge [ ] list = list
merge list [ ] = list
merge list1@ (x1 : xs1) list2@ (x2 : xs2)
      | x1 < x2 = x1 : merge xs1 list2
      | x1 = = x2 = x1 : merge xs1 xs2
      |otherwise = x2 : marge list1 xs2
– – функция mergeLists сливает без повторений список
– – упорядоченных списков в один список.
mergeLists : : Ord a => [(a)] —> [ a ]
mergeLists [ ] = [ ]
mergeLists (list : lists) = merge list $ mergeLists lists
– – основная функция для решения задачи получения
– – списка всех простых делителей заданного списка
– – натуральных чисел.
factorList : : [Integer] —> [Integer]
factorList list = mergeLists $ map factors list
```

Функция map относится к функциям высшего порядка.

Если некоторая функция служит только для ее передачи другой функции, то ей не даётся постоянное имя и явно не определяется её тип.
В этом случае используется λ – выражение (лямбда – выражение), которое задаёт образцы для аргументов функции и правые части её уравнений. Однако, она не связывает с этой функцией никакого имени, а тип такой функции определяется из контекста.

λ – выражение представляет собой функциональное значение, изображающее безымянную функцию.

В простейшем случае, когда функция определяется с помощью одного уравнения, λ – выражение имеет такой же вид, как и это единственные уравнения. Разница в том, что в левой его части вместо имени функции стой символ обратной косой черты " \ ", изображающей греческую букву **λ**, а вместо знака равенства образцов для аргументов от правой части уравнения используется последовательность символов '→'.
Например, если функция для возведения числа в квадрат не определена, то можно написать λ – выражение для её определения там, где это функция используется. Тогда вызов функции map для получения списка квадратов будет следующим

```
map ( \ x → x * x ) [ 1, 2, 5, -2]
```

Если уравнений несколько, то их можно объединить в одно выражение с помощью условного выражения if либо специальной конструкцией для выбора по образцу **case.**

Например, для получения списка из знаков заданного списка целых чисел с помощью функции map можно отдельно определить функцию для вычислений знака числа.

```
signum 0 = 0
signum n | n < 0 = −1
         | otherwise = 1
```

Вызов для нахождения списка знаков заданного списка чисел :

```
map signum [ 2, 5, 0, -3, 2, -2]
```

(в результате получим список [1, 1, 0, -1, 1, -1], знак зависит от того, является ли число нулевым, положительным или отрицательным)

Определение функции λ – выражением будет выглядеть так:

```
map ( \ n → if  n == 0 then 0 else if n<0 then – 1 else 1)
[ 2, 5, 0, -3, 2, -2]
```

Определение функции λ – выражением с использованием конструкции case будет следующим:

```
map ( \ n → case n of
    0 → 0
    n | n < 0 → −1
      | otherwise → 1) [ 2, 5, 0, -3, 2, -2]
```

Можно определение функции signum записать с помощью λ выражения

```
signum  =  \  n → case  n  of
0 → 0
n | n < 0 → −1
  | otherwise → 1
```

В λ – выражении можно использовать рекурсивный вызов.

Например, вычисление факториала первых 10 натуральных чисел

```
map factorial [ 1 … 10 ] where
factorial = \ n → if  n == 0  then  1  else  n  *  факториал  (n −1)
```

3.2 Операции свертки списка foldl и foldr, композиция и фильтрация функции

Списки – самая используемая структура данных в Haskell. Самые популярные из них – отображение (map), свертка (foldr, foldl и их разновидности), и фильтрация.

В стандартной библиотеке Haskell есть две функции foldl и foldr (операции свертки списка)

Функция foldl применяет заданную бинарную операцию к элементам списка от начала списка к его концу.

Функция foldr применяет заданную бинарную операцию к элементам списка от конца к началу.

Если применяемая к элементам списка бинарная операция ассоциативна (как сложение или умножение, то порядок обработки элементов не важен)

В программе определения функции свертки f – бинарная операция (функция с двумя аргументами, seed – начальное значение). Если бинарную операцию сложения применить последовательно ко всем элементам списка (начальное значение 0), то получается сумма всех элементов списка. Если бинарная операция умножение, получим произведение всех элементов (начальные значения 1)).

Если обрабатываемый список не содержит ни одного элемента, то результатом работы функции свёртки будет начальные значения.

Листинг 3.7 – Определение функций свёртки списков – foldl и foldr

```
foldl : : ( b → a → b ) → b → [a] → b
foldl _ seed [ ] = seed
foldl f seed ( x : : s ) = foldl f ( f seed x) s
foldr : : ( a → b → b ) → b → [a] → b
foldr _ seed [ ] = seed
foldr f seed ( x : s ) = f x ( foldr f seed s )
```

Сумма всех элементов заданного списка list можно получить простым вызовом любой из функций foldl или foldr, когда в качестве начального значения выбирается ноль:

```
foldl ( + ) 0 list
```

Рекурсивные вызовы “спрятаны” внутри функции foldl.

Функции высших порядков позволяют писать короткие и выразительные программы.

Например, вычисление факториала, в которой для вычисления сначала строится список целых чисел от единицы до заданного значения аргумента , а

потом все элементы полученного списка перемножаются для получения результата

factorial n = foldr (*) 1 [1 … n]

В библиотеке Haskell наряду с функциями foldl и foldr, которые не требуют начального значения, а в качестве такого берут первый или, соответственно, последний элементы самого списка. Естественно, эти функции не применимы к пустому списку.

Важно, если бинарная операция ассоциативна, то эти функции могут обрабатывать различные участки списка параллельно, повышая тем самым производительность в случае многопроцессорных систем.

Функции высших порядков, выдающие функции в качестве результата можно определять, чтобы строить новые функции на основе других.

Самая распространённая операция над функциями является их композиция.

Такая операция позволяет по двум функциям f и g получить новую fg такую, что результат последовательного применения функций f и g : f (g x).

В императивных языках такую функцию описать невозможно, а в языке Haskell такое описание не составляет никакого труда.

comp : : (b → a) → (c → b) → (c → a)

comp f g = \ x → f (g x)

Такая функция имеется в стандартной библиотеке языка Haskell в виде бинарной операции (.). Для примера, если имеется функция возведения числового значения в квадрат sqr, то функция возведения числа в четвёртую степень может быть получена

power4 = sqr . sqr

Фильтрация списка позволяет отобрать в списке те элементы, которые удовлетворяют некоторому заданному условию (предикату).

Предикат – это функция, которая принимает в качестве аргумента элемент списка и выдает логическое значение True или False. В соответствии с этим элемент остается в списке или удаляется.

Например:

```
Filter : : (a → Boll) → [a] → [a]
filter _ [ ] = [ ]
filter condition ( x : xs ) | condition x = x : filtered
                            | otherwise = filtered
    where filtered = filter xs
```

Если требуется не просто отбросить элементы, не удовлетворяющие условию, а собрать их в отдельный список, то пригодится похожая функция partition, которая в отличии от filter определена в модуле Data.List. Его необходимо подключить с помощью конструкции import.

Использование функций фильтрации дает возможность запрограммировать сложную процедуру быстрой сортировки списка. Функция quicksort, реализующая этот алгоритм выглядит так:

```
quicksort : : Ord a => [a] → [a]
quicksort [ ] = [ ]
quicksort (x : xs) quicksort first ++ [x] ++ quicksort second
where (first, second) = partition ( \ e → e < x ) xs
```

Из-за частого применения операции (++) соединения списков следует оптимизировать работу этой функции, добавив накапливающий аргумент.

```
quicksort list = quicksort’ list [ ] where
quicksort’ [ ] list = list
quicksort’ ( x : xs ) list = quicksort first (x : quicksort’ second list) where
    (first, second) = partition ( \ e → e < x) xs
```

Мощным средством обработки списков является сочетание функции отображения и свертки. Их последовательные применение позволяет сначала подготовить элементы списка к последовательной обработке. Эта подготовка

будет производиться параллельно, а затем будет, получен окончательный результат.

Например, пусть имеется таблица, в которой символом приписан номер (класс символа). В троке таблицы собраны символы (класса символа). К разным классам отнесены гласные буквы, согласные буквы, цифры, знаки препинания и другие.

Одна из вероятностей изображения таблицы:
[“aeiouy”, ”bcdfghjklmnpqrstvwxz”, ”0123456789”, ”( ) , . ; : ? ! -“]

Символы, не входящие ни в одну строку, будут иметь нулевой класс, гласные буквы – класс 1, согласные – класс 2 и т.д. Требуется по заданному тексту собрать статистику, сколько символов текста относятся к тому или иному классу. Например, по тексту “Test” будет собрана статистика [1,1,3,0,0], т.е. в тексте имеется символ, не относящийся ни к какому классу (заглавная ‘T’), одна гласная букв, три согласных, цифры и знак препинания отсутствуют. В функции, которая по заданному символу будет определять его класс, использована стандартная операция elem. Эта операция проверяет, является ли некоторое значение элементом синтаксиса. Здесь она используется для того, чтобы проверить, к какому классу относится символ.

Вычисление класса для каждого символа текста, осуществляется функцией process1, которая с помощью функции отображения map может параллельно обрабатывать каждый символ текста.

С помощью функции process2 каждый номер класса будет превращен в шкалу из нулей и единиц, в которой единица будет стоять на месте, соответствующему номеру класса, а остальные элементы шкалы станут нулем. Список номеров классов – allClasses.

Для почленного сложения списков используется операция (+++).

Сбор статистики заключается в свертке полученного списка шкал с помощью операции сложения элементарных статистик (+++)

Листинг 3.8 – Сбор статистики символов по заданному тексту

```
– – Таблица классов символов
classTable :: [[Char]]
classTable = [“aeiouy”,”bcdfghjklmnpqrstvwxz”,”0123456789”,”(),.;:?!-“]
– – Функция определения класса символа
getClass  :: Char →[[Char]] → Int
getClass c [ ] = 0
getClass c ( s : rest)  | c ‘elem’  s = 1
                        | restClass  = =  0  = 0
                        | otherwise = restClass + 1
      where restClass = getClass  c  rest
– – Список классов
allClasses :: [Int]
allClasses = [ 0 . . length classTable ]
– – Первый шаг обработки текста
process1 : : string → [Int]
process1 text  =  map (\c → getClass  c  classTable)  text
– – Второй шаг обработки текста
process2  : :  [Int] → [ [ Int ] ]
process2  classes   =  map flags classes where
     flags  n  =  map  (\i  → if  i  = = n  then  1 else 0)  allClasses
– – По членное сложение элементарных статистик
(+++)  : :  [Int] → [Int] → [Int]
[ ] +++ [ ] = [ ]
(x1 : xs1) +++ (x2 : xs2)  =  (x1 + x2)  :  (xs1  +++  xs2)
– – Сбор статистики
statistics  : :  String → [Int]
statistics text  =  foldll ( +++ )   (process2   (process1  text))
```

В этой программе символы текста обрабатываются по одному применением отображения map в функциях process1 и process2, а затем

полученные данные – элементарные статистики собираются в единую статистику.

Так как обработка отдельных символов текста может производиться параллельно, то даже для длинного текста вычисление классов символов и формирование элементарных статистик может производиться параллельно, то даже для длинного текста вычисление классов символов и формирование элементарных статистик может производиться быстро.

Пусть требуется по заданной строке получить список содержащих в ней слов. Любая последовательность рядом стоящих символов будет словом. Стандартная функция *isLetter* из модуля Data. Char подтвердит (выдаст) значение True.

Одно из решений - можно заменить в тексте все символы, не являющиеся буквами на пробелы, а потом воспользоваться стандартной функцией words, которая и выполнит всю требуемую работу. Эта функция может быть такой:

```
wordsList : : String → [String]
wordsList text = words $
                    map (\c → if isLetter c then c else ‘ ‘) text
```

Другое решение - символы текста собираются в группы по определённому признаку, в данном случае по тому, является ли символ буквой. В библиотеке Haskell имеется функция *group*, которая собирает элементы списка в группы равных между собой рядом стоящих элементов. Например, group [2, 3, 3, 2, 2, 2, 4, 4, 5] выдаст результат [[2], [3, 3], [2, 2, 2,] , [4, 4] , [5]] .

Эту функцию можно обобщить: вместо равенства использовать любое отношение эквивалентности, которое будет считать «одинаковыми» два значения, если некоторая функция от этих значений выдаст True. Такое обобщение тоже есть - это функция groupBy, имеющая тип

```
groupBy : : (a –> a –> Bool ) –> [a] –> [ [ a ] ].
```

После этого два символа можно считать равными, если они оба являются буквой

equalsAsLetter c1 c2 = isLetter c1 $\delta\delta$ isLetter c2

Применим функцию groupBy к тексту, в котором в качестве функции «равенства» используется функция equalsAsLetter:

>> groupBy equalsAsLetter “hello, world ! ”

[“ hello” , “, “, “ “, “ world” , “ ! ”]

Все слова выделились в группы символов, остальные символы остались в строках из одного этого символа. Затем следует отфильтровать полученный список.

Листинг 3.9 - Программа получения списка слов текста

```
wordsList : : String → [String]
wordsList text - filter isWord $ groupBy equalsAsLetter text
    where equalsAsLetter c1 c2 = isLetter c1 δδ isLetter c2
          isWord w = isLetter $ head w
```

Есть ещё две функции обработки списков: zip, unzip. zip - берёт два списка и составляет из них элементов список пар соответствующих друг другу элементов. Если выполнить команду

>> zip “hello” “world”

то результат будет:

[(‘h’ , ‘w’) , (‘e’ , ‘ o’) , (‘l’ , ‘r’) , (‘l’ , ‘l’) , (‘o’ , ‘ d’)]

Кортежи сравниваются с помощью операций сравнения, если можно сравнивать их элементы, при этом сравнение происходит в лексикографическом порядке: сначала сравниваются первые элементы кортежей, при их равенстве - вторые, и т. д., пока не определится, какой из кортежей больше, или не окажется, что кортежи равны. Такое же правило действует для любых списков.

Обратной к функции zip является функция unzip, которая из списка пар получает два списка одинаковой длины. Так как функция может иметь только один результат, то два списка упаковываются в кортеж из двух элементов, так что функция unzip имеет тип unzip : : [(a, b)] – > ([a] , [b])

Для поэлементной обработки трёх списков имеются функции zip3 и zipWith3, а также unzip3, а если потребуется параллельная обработка сразу четырёх или более списков, то можно подключить модуль Data.List, в котором определены аналогичные функции для четырёх, пяти, шести и семи списков.

3.3 Пошаговая свёртка списка, функции scanl и scanr

Функции scanl и scanr выполняют пошаговую свёртку списка, оставляя на каждом шаге промежуточный результат так, что в результате их работы образуется новый список из промежуточных результатов свёртки. По сравнению с функциями foldl и foldr, которые вырабатывают один конечный результат, функции scanl и scanr вырабатывают целый список.

```
>> it foldr
foldr : : (a → b →b ) → b → [a] → b
>> it scanr
scanr : : (a → b →b ) → b → [a] → [b]
>> foldl (+) 0 [1 . . 5]
15
>> scanl (+) 0 [1 . . 5]
[0, 1, 3, 6, 10, 15]
```

Функции scan могут строить списки из элементов, полученных вычислением на основе предыдущих элементов.

Например, список факториалов натуральных чисел от 1 до 7

```
>> scanl (*) 1 [1 . . 7]
[1, 1, 2, 6, 24, 120, 720, 5040]
```

Первый элемент полученного списка – факториал нуля, который равен единицы. Последующие элементы получаются умножением предыдущего элемента на очередное значение из исходного списка.

Если строка состоит из десятичных цифр, то преобразование строки в число может иметь следующий вид:

```
>> readInt : : String → Integer
readInt str = foldl (\ number c → number *10 + digitToInt c) 0 str
```

Теперь решение задачи суммирования чисел, которые есть в тексте можно представить в виде программы (листинг 3.10).

Листинг 3.10 – Программа суммирования чисел, изображения которых имеются в заданном тексте

```
import Data . Char (isDigit, digitToInt)
– – Функция замены символов, удовлетворяющих заданному условию
– – на заданный символ.
– – Аргументы:
– – условие замены символа;
– – символ для замены;
– – исходный текст.
– – Результат:
– – текст с замененными символами.
replace : : (Char → Bool) → Char → String → String
replace condition replacement text = map replaceChar text
    where replaceChar c = if condition c then replacement else c

– – Функция выделения из текста подстрок, изображающих числа.
– – Выделяемые числа - целые, беззнаковые.
getNumbers : : String → [String]
getNumbers text = words $ replace (not . isDigit) ‘ ‘ text
```

```
– – Функция преобразования строки в целое число при условии,
– – что в исходной строке нет ничего, кроме цифр.
readInt : : String → Integer
readInt str = foldl attachDigit 0 str
    where attachDigit n c = n*10 + (fromIntegral $ digitToInt c)

– – Функция суммирования чисел, представленных своими
– – изображениями в заданной строке.
sumNumbers : : String → Integer
sumNumbers text = sum $ map readInt $ getNumbers text
```

Любую программу можно написать, используя только встроенные простые типы данных, кортежи и списки. Однако для удобства организации данных в языке Haskell имеются механизмы, с помощью которых можно определить свои типы данных или ввести обозначения для уже имеющихся типов, а также определить допустимый набор операций над объектами. Эти вопросы освещены в ученом пособии, часть 2.

ЗАКЛЮЧЕНИЕ

Несмотря на лучшие возможности для распараллеливания вычислений, на компьютерах традиционной архитектуры программы, написанные в функциональном стиле, иногда показывают заметно меньшую производительность, чем программы, решающие те же задачи с помощью функциональных средств.

Возможности функционального программирования раскрываются в полной мере при организации программ, исполняющихся в распределенных системах или на компьютерах, обладающих существенно более широкими возможностями для распараллеливания вычислений, чем привычные персональные компьютеры.

В некоторых специализированных областях программирования, таких, как системы искусственного интеллекта, базы знаний, текстовая обработка и некоторых других функциональное программирование предоставляет удобные средства программирования задач.

ЛИТЕРАТУРА

Авторы, составители	Заглавие	Издательство, год
Душкин Р. В.	Функциональное программирование на языке Haskell	М.: ДМК Пресс, 2016.
Зыков С. В.	Программирование. Функциональный подход.: учебник и практикум для академического бакалавриата.	М.: Юрайт, 2016
Уорбэртон Р.	Лямда–выражения в Java 8	М.: ДМК Пресс, 2014.
Кубенский А. А.	Функциональное программирование	М.: Юрайт, 2016