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

Кафедра Информатика и вычислительная техника

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ПРОВЕДЕНИЯ ПРАКТИЧЕСКИХ ЗАНЯТИЙ И ЛАБОРАТОРНЫХ РАБОТ ПО ДИСЦИПЛИНЕ «ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ»

Ростов-на-Дону

2019 г.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ПРОВЕДЕНИЯ ПРАКТИЧЕСКИХ ЗАНЯТИЙ И ЛАБОРАТОРНЫХ РАБОТ ПО ДИСЦИПЛИНЕ

«Функциональное программирование»

Пособие предназначено для проведения занятий со студентами направления 09.03.01.

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

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

МУ обсуждены и одобрены на заседании

кафедры ИВТ

Протокол №1 от 26.08.19

Практическое занятие №1. (ОПК-2, ПК-2)

Цель занятия: Приводить выражения к нормальной и СЗНФ в системе редукций λ - исчисления, отличать понятия чистого и расширенного λ - исчисления.

1. Представление выражений в λ - исчислении.

Функциональные языки программирования появились в качестве средства для написания программ, не содержащих в явном виде понятий ячеек памяти для хранения значений (переменных) и последовательности вычислений как процесса изменения состояния памяти.

Основа для создания таких языков была предложена ещё в середине 1930-х гг. Алонзо Черчем и Стефаном Клини. Теория Черча, названная им λ - исчисление, рассматривается в качестве теоретической основы и «минимального» функционального языка программирования. Любую программу, написанную на любом функциональном языке программирования, можно свести к формуле λ - исчисления.

Черч и Клини не создавали язык программирования. Они занимались формализацией понятия вычислимой функции и создания универсального математического аппарата для определения и работы с вычислимыми функциями. Черч сумел построить такую систему, при которой базовых функций нет, а вместо способов построения сложных функций из простых используются правила преобразований. В результате Черчу удалось создать исчисление чистых функций, в котором все математические понятия, такие как числа, арифметика и др., сводятся к понятию функции.

Процесс преобразования формул в лямда – исчислении Черча (***редукции***) напоминает процесс вычисления функции, происходящий при исполнении программы.

Основным понятием в λ - исчислении является понятие выражения или формулы. В формулах переменные обычно обозначают аргументы функций, задаваемых λ - выражениями, однако сама по себе переменная является простейшим видом выражения.

λ - выражение имеет вид **(λx . e)**, где x – имя переменной, а **e** – выражение. Семантически такое выражение обозначает функцию с аргументом *x* и телом *e*. Применение функции записывается в виде (e1e2), где *e1* и *e2* – выражения (*e1* –функция, а *e2* – её аргумент).

Примеры:

- λx . x – простейшая функция, выдающая свой аргумент; скобки опущены, поскольку это не вызывает неоднозначности;

– λf . λx . f x – функция с двумя аргументами, применяющая свой первый аргумент ко второму. Строго говоря, надо было бы расставить скобки, чтобы выражение приняло вид λf . (λx . (f x)). Однако принято соглашение, по которому «операция» применения функции к аргументу имеет более высокий приоритет, чем «операция» образования λ - выражения. Функции при этом применяются в порядке слева направо. Выражение f x y понимается как (f x) y – применение функции **f** к аргументу **x** и применение полученного результата к аргументу **y .** Образование же безымянной функции (операция «лямбда») наоборот применяется справа налево, так что выражение **λx . λy . e** понимается как **λx . (λy . e).**

Следует учитывать, что при применении стандартных функций используется только префиксная запись операций, т. е. вместо привычного 3+5 записывается выражение + 3 5. Естественно, бессмысленным будет выражение / 3 0, поскольку результат деления числа 3 на число 0 не определен. Бессмысленным будет выражение + TRUE 0, поскольку невозможно выполнить сложение логического значения TRUE с числом 0. Условное выражение **if** B **then** E1 **else** E2 может быть представлено применением стандартной функции IF с тремя аргументами B, E1, и E2. Кортежи представляются в виде применения стандартных функций кортежирования TUPLE – n, где n – произвольное натуральное число. Такая функция будет иметь n аргументов, и выдавать в результате работы кортеж, содержащий эти аргументы.

Обратная функция извлечения элемента кортежа по заданному номеру будет представлена стандартной функцией INDEX, аргументами которой будет номер элемента кортежа и сам этот кортеж.

Пример:

– Если составить кортеж из чисел 2,4 и 10, а затем извлечь из него второй элемент, то должны в результате получить число 4. Соответствующая конструкция расширенного λ - исчисления будет выглядеть так:
INDEX 1 (TUPLE- 3 2 4 10). Так как элементы кортежа нумеруются с нуля, так что второй элемент кортежа будет иметь номер 1.

В λ - исчислении определена система *редукций*, т.е. набор правил преобразования выражений. С их помощью можно переходить от одних выражений к другим, эквивалентным им.

2. Нормальная форма.

Вхождение переменной в некоторое выражение будет *связанным*, если оно находится внутри некоторого λ - выражения, в заголовке которого эта переменная упомянута в качестве аргумента. Если переменная не является связанной в некотором выражении, то она в нём будет *свободной*.

Рассматриваются четыре вида редукций в λ - исчислении.

Первое называется переименованием переменных, или α-преобразованием.

Его смысл состоит в том, что суть функции не меняется, если заменить имя её формального аргумента.

Пример:

В выражении λx . λf . f x y можно заменить переменную x на переменную z.

Два следующих правила редукций – это «вычислительные» преобразования, которые приводят к упрощению выражения.

Преобразование δ-редукция имеет следующий вид: пусть имеется выражение f e1 e2 …ek, где f – константа, представляющее имя «встроенной» функции с *k* аргументами, а *e1, e2, … ek* – значения, могущие служить аргументами этой функции. Тогда такое выражение можно заменить на эквивалентное ему выражение.

Пример:

Если константа + представляет функцию арифметического сложения целых, а константа OR – функцию логического «ИЛИ», то в результате δ-редукции выражение + 1 4 может быть преобразовано к выражению 5.

Преобразование β - редукция соответствует применению функции, представленной λ - выражением, к аргументу.

Пример:

Выражение ((λx . + x x) 3) в результате применения β - редукции будет преобразовано в (+ 3 3), которое, в свою очередь, может быть преобразовано в 6 с помощью применения - δ-редукции.

Если некоторое выражение содержит в себе подвыражение, к которому можно применить одну из редукций, то такое подвыражение называется *редуцируемым* или сокращенно *редексом.* Таким образом процесс

преобразования выражения сводится к применению β и δ-редукций к *редексам*, содержащимся в исходном выражении.

η - преобразование выражает тот факт, что две функции, которые при применении к одному и тому же аргументу дают один и тот же результат, эквивалентны.

Основное назначение λ - исчисления состоит в том, чтобы показать, что любая вычислимая функция может быть представлена в виде λ - выражения.

Пример:
Функция (λx . * x x) представляет собой функцию возведения в квадрат. Функция действительно выдаст результат 36, если применить её к аргументу 6. Для этого составим выражение ((λx . * x x) 6) и выполним те редукции, которые можно применить к этому выражению. Сначала в результате применения β - редукции получится выражение (* 6 6) к которому можно применить δ-редукцию и окончательно получить значение 36.

Редукции можно применять до тех пор, пока в выражении имеется хотя бы один редекс. Если ни одного редекса в выражении нет, то говорят, что выражение находится в ***нормальной форме***.

3. Слабая заголовочная нормальная форма (СЗНФ).

В результате последовательного применения β и δ-редукций можно не получить нормальной формы выражения, а получим довольно близкий её эквивалент – так называемую слабую заголовочную нормальную форму (СЗНФ).

Выражение находится в СЗНФ, если оно имеет один из следующих видов:

- константа c (в том числе – одна из встроенных функций);
- переменная x;
- λ - выражение **λx . e,** где e – выражение, находящееся в нормальной форме;
- f e1 e2 … ek, где f – встроенная функция с *n* аргументами, e1,e2, … ek – произвольные выражения, причём k<n.

Если записать последнее выражение в эквивалентном виде λ - выражения, то можно видеть, что нормальная форма отличается от СЗНФ

только тем, что в СЗНФ не производится приведение к нормальной форме выражений, находящихся внутри тела λ - выражения.

Контрольные вопросы к выполнению практического занятия

Практическое занятие №.1 (ОПК-2, ПК-2)

1. Кем и когда была предложена математическая основа для создания функциональных языков программирования.
2. Какую систему в **λ(лямда) –исчислении** построил Алонзо Черч.
3. Основное понятие в λ(лямда) – исчисления Черча.
4. Представление выражений в λ(лямда)-исчислении.
5. Место и роль различных *редукций* в λ(лямда)-исчислении .
6. Преобразование исходного выражения к его нормальной форме.
7. Слабая заголовочная нормальная форма выражения (СЗНФ).
8. Рекурсия в λ(лямда)-исчислении.
9. Чистое λ(лямда)-исчисление.

Задание №1: Определить, какие из переменных в заданном выражении λ - исчисления связаны, а какие свободны?

X (λx. λy.y (λz.x) z) (λx.y);

Задание №2:

На языке Haskell даны определения следующих двух взаимно – рекурсивных функций:

```
f x = x : g (2*x)
g y = y : f (y+1)
```

Составить выражения для этих функций в расширенном **λ(лямда) –исчислении.**

Выполнение заданий практического занятия оформляется в виде отчёта с ответами на контрольные вопросы и с пояснениями выполнения заданий.

Практическое занятие №2 (ОПК-2, ПК-2).

Цель занятия: Решить задачи на языке Java или Паскаль в императивном стиле и с использованием функционального стиля.

Функциональное программирование – это ветвь программирования, при котором программирование ведётся с помощью определения функций.

Алгоритмы, записанные с помощью языков функционального программирования, во- первых, допускают сравнительно простой анализ и формальное преобразование программ. Во – вторых, отдельные части программ могут исполняться независимо друг от друга. Такими замечательными свойствами и обладают языки функционального программирования.

Описание алгоритмов в функциональном стиле сосредоточено не на том, как достичь нужного результата (в какой последовательности выполнять шаги алгоритма), а больше на том, что должен представлять собой этот результат.

Отличительными особенностями функций, определяемых любым функциональным языком программирования, являются следующие:

- каждая функция в программе выдаёт один и тот же результат на одном и том же наборе входных данных (аргументов функции), т.е. результат работы функции является «повторяемым»;
- вычисление функции не может повлиять на результат работы других функций, т.е. функции являются «чистыми».

Если программа представляет собой набор чистых детерминированных функций , то она будет функциональной независимо от того, написана ли она на специальном языке функционального программирования Haskell или на традиционном Java.

Пример: Функция вычисления суммы элементов числового списка.

```
double sumList (List<Double> list)  {
  double  sum = 0;
  for     (Double element : list)      {
     sum += element;
}
return  sum;
}
```

Эта функция детерминированная и «чистая», она выдаёт всегда один и тот же результат на одинаковых входных данных и не влияет на поведение других функций.

Однако всё же с точки зрения функционального стиля имеет одну

«неправильность». В функции определяется и используется локальная переменная *sum*, которая нужна для запоминания промежуточных значений суммы.

В чисто функциональных программах присваиваний вообще нет, как нет и понятия последовательного вычисления. Каждая функция должна представлять собой суперпозицию обращений к другим функциям.

Представим приведенную программу в функциональном стиле.

```
double sumList (List<Double> list)  {
    final int size = list. size () ;
    final int mid = size / 2;
    return
       size == 0 ? list . get (0)  :
              sumList (list. subList (0,  mid ) )  +
              sumList (list. subList (mid, size ) ) ;
```

Вместо цикла использована рекурсия, а вместо условного оператора в этой программе применяются условные выражения, которые соединяют условиями не отдельные части последовательно выполняющейся программы, а отдельные подвыражения. Это является характерной особенностью функционального стиля программирования.

Контрольные вопросы к выполнению практического занятия

1. Отличительные особенности функций, определяемых функциональным языком программирования.
2. Особенности функционального стиля программирования.
3. Недостатки и преимущества функционального стиля по отношению к императивному стилю программирования.
4. «Чистые» и «нечистые» функции.
5. Что представляет собой каждая функция в функциональном программировании.
6. Что используется вместо цикла в функциональной программе.

Задание: Составить программы решения задач на языке Java в привычном императивном стиле и в функциональном стиле программирования

1. Приближенное вычисление числа ***e*** по формуле для

разложения e^x в ряд Тейлора.

2. Вычисление приближённого значения корня уравнения cos x = x методом бисекции.

Выполнение заданий практического занятия оформляется в виде отчёта с ответами на контрольные вопросы и с пояснениями выполненных заданий.

Лабораторная работа №1. (ОПК-2, ПК-2)

Цель работы: установка системы Haskell Platform.

Для того чтобы начать работу с функциональном языком программирования, нужно загрузить на компьютер систему программирования на языке Haskell, в состав которой входят, в частности, компилятор и интерпретатор программ, написанных на этом языке, а также простая оболочка для операционной системы Microsoft Windows. Систему Haskell Platform можно установить с сайта https://www.haskell.org, перейдя по ссылке https://www.haskell.org/platform/.

После установки системы можно сразу же запустить интерпретатор выражений, записанных на языке Haskell, GHCi (интерпретатор командной строки) или (в системе Windows) WinGHCi.GHC (Glasgow Haskell Compiler) – это популярный компилятор программ на языке Haskell.

Все примеры работы интерпретатора будут приводиться непосредственно в тексте. Если в интерпретаторе набрать выражение 2*2, то он выведет результат вычисления – 4. В тексте это будет показано следующим образом:

```
>> 2*2
4
```

Команды, подаваемые интерпретатору, - это выражения для вычисления или собственно команды, с помощью которых можно управлять работой интерпретатора, а также получать всевозможную информацию о текущем контексте вычислений, получать отладочную информацию, устанавливать режимы работы, загружать программы и программные модули и т.д.

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

Компилятор GHC умеет вычислять тип поданного ему выражения с помощью команды **: type,** которую можно задавать в сокращенном виде - **: t:**

Посмотреть на конкретный тип вычисленного выражения можно, включив опцию компилятора с помощью команды “ : set + t”. После включения этой опции интерпретатор после каждого вычисления будет показывать тип вычисленного значения

Задания к выполнению лабораторной работы.

1. Система программирования Haskell Platform:
2. Установка системы программирования Haskell Platform;
3. Запуск интерпретатора выражений на языке Haskell – GHCi (интерпретатор командной строки)
4. Получение и просмотр полного списка команд;
5. Установление различных режимов работы интерпретатора.

Выполнение заданий лабораторной работы оформляется в виде отчёта с ответами на контрольные вопросы и с пояснениями выполненных заданий.

Лабораторная работа №2. (ОПК-2, ПК-2)

Цель работы: Подготовить, загрузить и исполнить программы на языке Haskell с помощью установленного интерпретатора.

Программа на языке Haskell – это набор определений значений и функций. Загрузив программу в интерпретатор, можно затем выполнять действия по вычислению результатов, просто вызывая функции, определения которых заданы в программе. Для примера, чтобы показать, каким образом можно подготовить, загрузить и исполнить программу на языке Haskell с помощью установленного интерпретатора, определим функцию с тремя аргументами, которая выдаст результат вычислений по известной формуле Герона.

Определим в программе функцию ***triangle*** с аргументами ***a,b*** и ***c***, которая сначала введёт обозначение ***p*** для значения полупериметра треугольника, а затем вычислит его площадь, используя как вычисленное значение полупериметра, так и аргументы – длины сторон треугольника.

```
triangle a b c = let p = (a + b + c) / 2 in  sqrt (p * (p-a) * (p-b) * (p-c))
```

Запишем программу в текстовый файл с именем triangle.hs (расширение имени файла .hs используется для текстов программ на языке Haskell).

Пусть, например, полное имя файла в системе Windows будет
C:\ Haskell\triangle.hs

Теперь запустим интерпретатор и подадим команду на загрузку и проверку программы:

```
>> : load  C:\Haskell\triangle.hs
```

Если имя файла записано правильно и не содержит пробелов и букв, отличных от букв латинского алфавита, то программа будет загружена, и интерпретатор проверит правильность синтаксиса. Если все правильно, то на экране увидим результат работы интерпретатора.

Теперь можно попробовать вычислить площадь какого-либо треугольника, записав выражение:

```
>> triangle 3 4 5
```

что приведёт к запуску функции и выдаче ожидаемого результата 6.0.

Задания к выполнению лабораторной работы.

1. Составление простой программы (например, вычисление площади треугольника по формуле Герона).
2. Запись программы в текстовый файл с именем triangle.hs (полное имя файла в системе Windows будет C:\Haskell\ triangle.hs).
3. Запуск интерпретатора и подача команды на загрузку и проверку программы.
4. Загрузка выражения для применения функции.
5. Проверить правильность работы функции (программы) при нескольких вычислениях.
6. Для подготовки программ изучить следующие вопросы:
   - элементарные типы данные Haskell;

Задания на выполнение лабораторной работы:

- подготовить и исполнить программу вычисления площади треугольника при различных аргументах;
- подготовить программу нахождения чисел Фибоначчи.

Выполнение заданий лабораторной работы оформляется в виде отчёта с ответами на контрольные вопросы и с пояснениями выполненных заданий.

Практическое занятие №3. (ОПК-2, ПК-2)

Цель занятия: Изучение типов и классов в языке программирования Haskell.

Haskell – строго типизированный язык. Такими же свойствами обладают большинство современных языков программирования, такие как Паскаль, Java, C++ и многие другие. Лисп – язык без строгой типизации.

Основу системы типов Haskell составляют типы: целые (Integer и Int); вещественные (Float и Double); логические (Bool); символьные (Char). Все идентификаторы в Haskell чувствительны к регистру букв, так что integer, Integer, INTEGER – это три разных идентификатора.

Регистр первой буквы идентификатора определяет встроенный (заглавная буква) или определённый программистом тип или класс. С заглавных букв начинаются также имена модулей и пакетов. Идентификаторы объектов – значения простых и сложных типов, в том числе функций – начинаются со строчной буквы или символа подчёркивания. Идентификаторы строятся из букв, подчёркиваний и цифр и символа апострофа. Ни апостроф, ни цифра не могут быть первым символом идентификатора.

В программе можно вводить собственные идентификаторы для имеющихся значений.

Классы в языке Haskell имеют примерно тот же смысл, что и других языках программирования: они определяют набор операций (функций), которые можно производить над объектами разных типов, принадлежащих этому классу. Например, полный список разрешенных операций для класса *Num* можно получить, набрав команду *: info*, в результате выполнения которой получим примерно следующий результат:

```
>> : info  Num
Class Num  a where
(+) : :  a –> a –> a
(*) : :   a –> a –> a
(-)  : :   a –> a –> a
negate  : :  a –> a
abs  : : a –> a
signum  : :   a  –>  a
fromInteger  : :  Integer  –> a
```

В данном описании указаны все функции и операции, разрешённые для типов из этого класса, так что, если выполнить операцию сложения над двумя значениями, каждое из которых имеет тип, принадлежащий классу Num, то в результате получим значение того же типа, что и типы операндов.

Задания контрольные вопросы к выполнению заданий практического занятия.

1. Различие между типами и классами в языке программирования Haskell.
2. Особенности базовых типов Haskell, непривычных по другим языкам программирования.
3. Объединение произвольных типов в кортежи.
4. Особенности языка Haskell касающиеся изображения чисел.
5. Операции и стандартные функции в языке программирования Haskell.
6. Две особенности записи выражений.
7. Объединение произвольных типов в кортежи.
8. Пустой кортеж

Выполнение заданий практического занятия оформляется в виде отчёта с ответами на контрольные вопросы и с пояснениями выполнения заданий.

Лабораторная работа №3(ОПК-2, ПК-2)

Цель работы: Определение функций с помощью уравнений. Подготовить, загрузить и исполнить программы на языке Haskell с помощью установленного интерпретатора.

Для того, чтобы написать сколь-нибудь полезную программу, необходимо помимо новых идентификаторов и правильного написания выражений уметь определять новые функции.

Каждая функция в Haskell – это тоже некоторое значение, для которого определён тип. В типе функции указывают типы её аргументов и тип значения функции, при этом типы аргументов и значения функции отделяются друг от друга символами " –>". Так что, например, тип функции одного вещественного аргумента с вещественным результатом (такой, как, например, функция вычисления синуса) можно записать в виде

Double –> Double ,

а тип функции с двумя целыми аргументами и одним логическим результатом (такой, как, например, операция сравнения двух целых значений по величине) – в виде Int –>Int –>Bool.

Саму функцию можно определить с помощью «уравнения», в котором выясняется, как функция должна себя вести, если ей задать значение аргумента. Например, определим функцию удвоения, которая удваивает значение своего вещественного аргумента и выдаёт получившееся значение. Для этого задаётся идентификатор функции *twice*, задаётся её тип и записывается уравнение, в котором показывается, что вызов этой функции с заданным значением аргумента эквивалентен выражению, в котором это значение умножается на 2:

```
twice :: Double –> Double
twice x = 2 * x
```

Теперь можно загрузить и скомпилировать функцию для удвоения вещественных чисел.

Если функция twice определена, то вычисление выражения twice 5.5 можно представить следующим образом. Сначала происходит сопоставление фактического значения аргумента 5.5 с формальным аргументом x. Затем вызов функции заменяется правой частью уравнения, в которой вместо формального аргумента используется сопоставленное с ним значение

фактического аргумента. Таким образом, после сопоставления и замены вместо выражения twice 5.5 получаем выражение 2*5.5, которое после вычисления даёт значение 11.Процесс преобразования выражения можно записать следующим образом:

twice 5.5 →2 * 5.5 →11

Преобразование выражений, подобное приведенному, называют *редукциями.*

Таким образом, вычисление выражений в Haskell (исполнение программы) осуществляется с помощью последовательных редукций исходного выражения.

В целом процесс вычислений можно представить следующим образом. Сначала имеется некоторое исходное выражение. В этом выражении выбирается некоторый вызов функции(более точно – применение функции к аргументам). Затем рассматривается уравнение, определяющее эту функцию, и осуществляется сопоставление аргумента с формальным параметром функции. После этого правая часть уравнения, в которой все формальные параметры заменены на фактический аргумент, подставляется на место вызова. Так продолжается до тех пор, пока в выражении больше не останется применений функций к аргументам. Тогда говорят, что исходное выражение приведено к нормальной форме.

Задание к выполнению лабораторной работы.

Составить программу вычисления факториала натурального числа:

- определить значение функции при задании отрицательного значения аргумента;
- при выдаче интерпретатором сообщения об аварийном завершении работы для определения функции записать несколько уравнений, определяющих поведение функции при различных значениях аргумента;
- определить ту же функцию, задав аргументу вызова нулевое значение;
- проверить правильность работы функции при нескольких вычислениях.

Выполнение заданий лабораторной работы оформляется в виде отчёта с ответами на контрольные вопросы и с пояснениями выполненных заданий.

Практическое занятие №4. (ОПК-2, ПК-2)

Цель занятия: Рекурсивное определение функций в языке программирования Haskell.

Функции «встроенные» или «примитивные», работа которых не определяется уравнениями, «встроены» в язык. Например, примитивными функциями являются все арифметические операции. Редукция выражений, содержащих примитивные операции, осуществляются за один шаг, без сопоставления и подстановки.

Определение функции может быть рекурсивным, т.е. в правой части уравнения может быть вызов определяемой функции. В этом случае в процессе преобразования выражения, содержащего вызов рекурсивной функции, может получиться выражение, также содержащее вызов той же самой функции. Для того чтобы процесс вычисления мог закончиться, необходимо использовать условные выражения, которые приводят к выбору одной из двух альтернатив при вычислении сложных выражений. Условное выражение имеет вид:

if <условие> **then** <выражение – «то»> **else** <выражение – «иначе»>

Зададим определение простой рекурсивной функции, предназначенной для вычисления факториала заданного целого числа. Определение этой функции может выглядеть следующим образом:

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

Вместо символа →, который использовался для того, чтобы показать один шаг редукции, можно просто записывать результаты последовательных редукций в последовательных строках текста.

```
Factorial  3
if  3 == 0  then  1  else  3  *  factorial  2
if  False  then  1  else  3  *  factorial  2
3  *  factorial  2
3  * if  2  ==  0  then  1  else  2  * factorial  1
3  *  2  *  factorial  1
3  *  2  *  if  1  ==  0 then  1  else  1  *  factorial  0
```

```
3 * 2 * 1 * factorial 0
3 * 2 * 1 * if 0 == 0 then 1 else 0 * factorial (n-1)
3 * 2 * 1 * 1
6
```

Вместо применения условного выражения можно использовать более наглядную запись уравнения с «охраняющими условиями» (guards).

Контрольные вопросы и задания.

1. Редукция выражений, работа которых не определяется уравнениями, содержащих примитивные встроенные операции (арифметические операции).
2. ***Рекурсивно*е** определение функции.
3. Вычисление условного выражения.
4. Использование записи уравнения с «охраняющими условиями» вместо применения условного выражения
5. Процесс преобразования выражения при вычислении факториала числа.

Выполнение заданий практического занятия оформляется в виде отчёта с ответами на контрольные вопросы и с пояснениями выполнения заданий

Лабораторная работа №4 (ОПК-2, ПК-2)

Цель работы: Техника работы со списками.

В традиционных языках программирования для построения объектов, более сложных, чем элементарные числа или логические значения, используют различные встроенные механизмы для построения сложных объектов из более простых.

В языке Haskell имеется способ, позволяющий строить структуры данных произвольной степени сложности из более простых объектов – списки. Списки напоминают и списковые структуры традиционных языков программирования, и обычные массивы.

Список - это последовательность объектов одного и того же типа, состоящая из произвольного числа объектов (возможно, ни одного - список называется пустым). Тип списка определяется типом его компонент и обозначается [T]. Таким образом, [Integer] – тип компонентов целые числа, [([Char], Double)]-компонентами являются кортежи из двух полей, причём первым полем в каждом кортеже будет список символов, а вторым полем вещественное число. Сами объекты списки задаются перечислением своих компонентов в квадратных скобках через запятую.

Список, состоящий из символов, можно записать в виде строки, заключив составляющие его символы в двойные кавычки. Например, строка «Haskell» обозначает в программе список из семи символьных значений. Для списков символов можно использовать идентификатор типа **String** вместо [Char].

Второе средство построения списков - это задание списка чисел, образующих арифметическую прогрессию. Когда шаг прогрессии равен единице, то указывается только первый и последний элементы списка [1..10]. Если шаг прогрессии не равен единицы, то указываются два первых элемента и последний элемент. [3, 5..11].

Для того , чтобы писать программы обработки списков, нужно научиться извлекать отдельные элементы списка. Это можно делать с помощью встроенных операций, позволяющих извлечь отдельные составные части списка, либо посредством аппарата сопоставления с образцом.

Важнейшие встроенные операции обработки списков:

- head, last – функции, которые по заданному аргументу-списку выдают его первый и последний элементы

соответственно;

- tail – функция, выдающая остаток списка, полученный отбрасыванием первого элемента;
- !! – операция, которая по списку и заданному номеру элемента выдаёт соответствующий элемент списка. Список должен быть непустым, а номер должен лежать в пределах от нуля (первый элемент списка имеет номер ноль) до количества элементов списка без единицы. Например вычисление конструкции [1, 2, 3, 4] !! 2 приведёт к выдаче в качестве результата числа 3 – элемента списка, имеющего номер 2;
- null – функция проверки пустоты списка. Если в качестве аргумента этой операции будет задан пустой список, то функция выдаст значение True, в противном случае – False;
- length – функция, вычисляющая количество элементов в списке;
- ++ - операция соединения двух списков.

Задания к выполнению лабораторной работы:

1. Определить функцию (составить программу), которая по заданным натуральным числам определяет их наибольший общий делитель согласно алгоритму Евклида.
2. Вызвать стандартную функцию ***g c d*** для проверки результата.
3. Определить функцию для проверки того, является ли заданное натуральное число простым:
   - кроме основной функции *p r i m e* применить вспомогательную функцию *p r i m e',* так как рекурсию по основному аргументу функции (проверяемому числу) организовать трудно.;
   - использовать идентификатор ***otherwise*** в качестве условия при написании уравнения для функции *prime';*
   - проверить правильность работы функции при нескольких вычислениях.
4. Подготовить программу суммирования элементов целочисленного списка.

Выполнение заданий лабораторной работы оформляется в виде отчёта с ответами на контрольные вопросы и с пояснениями выполненных заданий.

Практическое занятие №5 (ОПК-2, ПК-2)

Цель занятия: Уметь пользоваться стандартными функциями высших порядков для массовой обработки данных.

В языке Haskell имеется большое количество стандартных функций, которые имеют функциональные аргументы и результаты. Рассмотрим стандартную функцию обработки списков **map** (отображение), которая, получив в качестве аргументов список и функцию преобразования элементов этого списка, выдаёт в качестве результата список из преобразованных элементов, полученных применением функционального параметра к каждому элементу списка. Если, например, функция *sqr* возводит свой аргумент в квадрат, то с помощью функции **map** можно , имея список целых чисел, получить список их квадратов: map sqr [1, 2, 5, -2] → [1, 4, 25, 4].

Функция, подобная map, которая в качестве аргумента получает другую функцию или выдаёт другую функцию в качестве результата, называется ***функцией высшего порядка***, или ***функционалом***.

В стандартной библиотеке языка Haskell имеются две функции высшего порядка, одна из которых применяет заданную бинарную операцию к элементам списка от начала списка к его концу, а вторая начинает обработку элементов с конца списка, применяя заданную в качестве аргумента операцию последовательно, двигаясь по направлению к началу списка Первая из этих операций называется ***foldl,*** вторая – ***foldr***. Общее название для этих операций - операции свёртки списка.

Мощным средством обработки списков является сочетание функций отображения и свёртки. Их последовательное применение позволяет сначала подготовить элементы списка к последовательной обработке, причём эта подготовка будет производиться параллельно, а затем уже будет получен окончательный результат.

Например, пусть имеется таблица, в которой некоторым символам приписан номер (класс символа). К различным классам отнесём гласные буквы, согласные буквы, цифры, знаки препинания и все прочие символы. Таблица должна представлять собой список строк, в которой в каждой строке собраны символы одного класса. Программа сбора статистики по тексту представлена следующим образом.

```
-- 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  =l  | restClass ==  0  = 0
                                  | otherwise  = restClass  +  1
Where restClass  =  getClass  c  rest
……..
```

Контрольные вопросы и задания.

1. Понятия функций высшего порядка .
2. Понятие **λ-выражений** и методы их использования в функциях высших порядков.
3. Объединение нескольких уравнений в одно выражение с помощью специальной конструкции для выбора по образцам.
4. Операции свёртки списка ***f o l d l u f o l d r.***
5. Уравнения функций свёртки списков.
6. Рекурсивные вызовы внутри функции ***f o l d l .***
7. Обработка списков с помощью функций высших порядков.
8. Дописать программу статистики символов по заданному тексту

Выполнение заданий практического занятия оформляется в виде отчёта с ответами на контрольные вопросы и с пояснениями выполнения заданий.

Лабораторная работа №5 (ОПК-2, ПК-2)

Цель работы: Использовать навыки применения функций высших порядков в программировании на языке Haskell.

Конструктор для формирования списков имеет обозначение – двоеточие. Конструкторы и функции имеют много общих черт. И конструктор, и функция служат для образования новых значений из имеющихся. Разница состоит в том, что функция для образования нового значения требуются правила, с помощью которых это значение и образуется, а конструктору никаких правил не нужно – аргументы собираются конструктором в единый объект нового типа механически.

В качестве простого примера функции обработки двоичных деревьев рассмотрим функцию вычисления высоты дерева:

```
-- функция для определения высоты заданного двоичного дерева

Height :: Tree a –> Int
height Null
height (Tree _ t1 tr) = 1 + max (height t1) (height tr)
```

Что касается свёртки, то для дерева можно определить много вариантов прохода по его узлам, соответственно, можно будет несколько вариантов функции свёртки дерева согласно алгоритмам его прохождения. Имея такую функцию, можно легко вычислить сумму значений узлов, хранящихся в дереве, или определить максимальное значение узла для непустого дерева:

```
tsum :: Num a => Tree a –> a
tsum t = tfoldr (+) 0 t
tmax :: Ord a => Tree a –> a
tmax t@ ( Tree root _ _) = tfoldr max root t
```

Контрольные вопросы и задания для выполнения лабораторной работы

1. Составить матрицы смежности для заданного графа.
2. Представить обход графа в ширину, построив дерево обхода.
3. Программировать обход графа.
4. Составить функцию (программу) построения пути между двумя заданными вершинами в графе.

5. Проверить правильность работы функции (программы)

Выполнение заданий лабораторной работы оформляется в виде отчёта с ответами на контрольные вопросы и с пояснениями выполненных заданий.

Литература

Авторы	**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