Интернет Образование
Орловский Региональный Центр

Гордиенко А.П.

Функциональное и логическое программирование: методические указания к лабораторным работам

г. Орел

ББК 32.98

Г68

Гордиенко А.П. Функциональное и логическое программирование: методические указания к лабораторным работам / А.П. Гордиенко / Учебное электронное издание. — Орел: АНО "Центр интернет-образования", 2015. — 25 с.

Методические указания содержат описание лабораторных работ, направленных на закрепление теоретических знаний и приобретение практических навыков программирования для систем искусственного интеллекта.

Введение

Данные методические указания содержат описание лабораторных работ по дисциплине «Функциональное и логическое программирование». Лабораторные работы направлены на закрепление теоретических знаний и приобретение практических навыков программирования для систем искусственного интеллекта.

1. ПОДГОТОВКА К ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

Лабораторные работы рассчитаны на студентов, имеющих минимальный опыт работы на языках программирования высокого уровня. Студент должен уметь работать в любом текстовом редакторе и с проводником. Основной же целью выполнения лабораторной работы является приобретение практических навыков программирования на языках Пролог и Haskel.

При подготовке к каждой лабораторной работе студент должен повторить лекционный материал, относящийся к изучаемому вопросу, а также ознакомиться с материалом, приведенным в соответствующем разделе данных методических указаний. При этом необходимо обращать особое внимание на разобранные примеры и фрагменты программ. Выполнение большинства работ основывается на материале, освоенном в предыдущих работах.

Готовность студента к работе определяется преподавателем путем проведения собеседования. Основным материалом для собеседования являются контрольные вопросы, приведенные в разделе 5.

Защита лабораторной работы производится после написания студентом программы и оформления отчета.

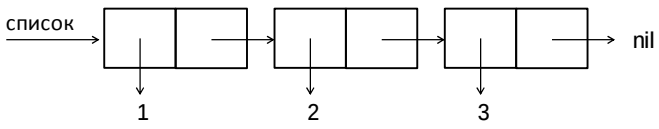
2. СТРУКТУРЫ ДАННЫХ: СПИСКИ, КОРТЕЖИ

2.1. СПИСКИ

В программах часто требуется обрабатывать последовательности значений, таких как, например, последовательность первых простых чисел, не превышающих некоторое число **N**. Длина такой последовательности заранее не известна, поэтому нужно обеспечить возможность добавления элементов в последовательность. Для этой цели служат списки. Списки на Паскале определяются так:

```
type List = ^Node;
type Node = record
              a: integer;
              b: List;
            end;
```

Здесь список **List** — это указатель на запись типа **Node**, имеющую два поля: **a** – обычно называемое головой списка и **b** – называемое хвостом. Список, содержащий числа **1,2,3** в памяти машины будет иметь вид:



В Хаскеле используется более абстрактное представление списков, в котором явно не показаны указатели. Известно, сколько хлопот бывает у программиста, когда ему требуются объекты в динамической памяти, обращение к которым идёт через указатели. В Хаскеле есть специальные объекты — *конструкторы данных*, посредством которых данные можно собирать в структуры. Конструктор данных имеет имя и список аргументов. Для построения списков требуется конструктор, обозначаемый оператором **:**, аргументы которого соответственно голова и хвост списка. Он ассоциативен вправо и имеет приоритет **5**. Этим конструктором строится запись, аналогичная записи **Node**.

Для построения списка требуется ещё конструктор **[]**, не имеющий аргументов и обозначающий пустой список. Например, список из трёх чисел: **1,2,3** задаётся как **1:2:3:[]**. В Хаскеле есть синтаксическое расширение, позволяющее записывать списки перечислением элементов через запятую, заключёнными в квадратные скобки. Пример: **[1,2,3]**. Это более привычная запись списка: **1:2:3:[]**.

Отметим, что список определяется рекурсивно:

1. Пустой список — это список.

2. Применение оператора **:** к элементу и списку тоже список.

В каждом списке все элементы должны быть одного типа. Подробно о типах мы будем говорить в следующих главах. Итак, *список* — это последовательность однотипных элементов.

Функция, обрабатывающая список, должна иметь возможность разборки его на части. В Хаскеле есть стандартные функции **head** и **tail**, которые берут соответственно голову и хвост списка. То есть:

```
head [1,2,3] ⇒1
tail [1,2,3] ⇒[2,3]
```

Очевидно, что `tail[1] ⇒ []`, а вызовы `head []` и `tail []` приведут к ошибке при выполнении программы. Важное значение имеет предикат `null`, который истинен, если его аргумент – пустой список.

Одновременно конструктор : является функцией от двух аргументов, которая возвращает список, когда заданы его голова и хвост: `1:[]⇒[1].` Эта функция строит в памяти запись вида

:	1	[]

,

которая имеет тэг : и два поля `1` и `[]`. Точнее: создаётся область в динамической памяти с тегом и аргументами. Список – это указатель на такую запись. Вопрос: а если нужно будет разбирать список, то потребуется освобождать память? Об этом программисту заботиться не нужно потому, что в программе есть универсальный «сборщик мусора», который автоматически будет освобождать неиспользуемую память.

Поскольку списки имеют рекурсивную структуру, их легко обрабатывать рекурсивными функциями. Для примера напишем функцию `append`, которая строит список приписыванием к элементам первого списка элементов второго списка, то есть строит соединение двух списков. Например,

```
append [1,2,3] [4,5] ⇒[1,2,3,4,5]
```

Таким образом, получается список, в котором сначала в том же порядке идут элементы первого списка, а потом – второго. Рассмотрим, как можно определить рекурсивную функцию соединения списков.

Рекурсию определим по первому списку. Рассуждать будем так. Нужно вычислить `append xs ys`. Одним рекурсивным вызовом `append (tail xs) ys` нам будет дано решение задачи для хвоста первого списка. Для получения окончательного результата к нему нужно добавить голову – голову первого списка. Таким образом, в *общем случае*, функция соединения списков формулируется так: в результате получается список, с головой как у первого списка, а хвостом – результатом соединения хвоста первого списка со вторым списком. В *базовом случае*, когда первый список пуст, результатом будет

второй список. На Хаскеле эта функция примет вид:

```
append xs ys = if null xs then ys
               else head xs : append (tail xs) ys
```

Для примера **append [1,2] [3,4]** получим последовательность операций:

```
append [1,2] [3,4] Þ
{xs=[1,2],ys=[3,4]} if null xs then ys
                    else head xs : append (tail xs)
ys Þ
{xs=[1,2],ys=[3,4]} head xs : append (tail xs) ys Þ
{xs=[1,2],ys=[3,4], xs'= tail xs,ys'=ys} head xs : if
null xs' then ys'
                    else head xs' : append (tail
xs') ys' Þ
{xs=[1,2],ys=[3,4], xs'= [2],ys'=ys} head xs : head xs'
: append (tail xs') ys' Þ
{xs=[1,2],ys=[3,4], xs'= [2],ys'=ys, xs"= tail
xs',ys"=ys'}
   head xs : head xs' : if null xs" then ys"
                    else head xs" : append (tail
xs") ys" Þ
{xs=[1,2],ys=[3,4], xs'= [2],ys'=ys, xs"=[],ys"=ys'}
   head xs : head xs' : ys"
Þ 1:2:[3,4] ≡ [1,2,3,4]
```

Функция сцепления списков используется очень часто, поэтому для неё в стандартную библиотеку введён специальный ассоциативный вправо оператор **++**, имеющий приоритет **5**.

2.2. СОПОСТАВЛЕНИЕ С ОБРАЗЦОМ

Определение функций, обрабатывающих списки, можно сделать понятнее, если применить сопоставление с образцом. Рассмотренные нами в предыдущей главе шаблоны можно дополнить конструкторами данных. Шаблон – конструктор данных требует, чтобы соответствующий ему аргумент был построен таким же конструктором, а соответствующие компоненты были сопоставимы. Например, шаблон **[]** сопоставим с пустым списком, шаблон **(x:xs)** со списком, имеющим голову и хвост (здесь

переменные — тоже шаблоны), а шаблон **(1:ys)** — со списком, начинающемся с единицы. Шаблоны могут иметь и вложенную структуру, например, шаблон **((x:xs):ys)** сопоставим со списком, голова которого — непустой список. В качестве шаблонов можно использовать и синтаксические расширения вида: **[x]** — список из одного элемента, **[x,y]** — список из двух элементов. Понятно, что они соответствуют шаблонам **x:[]** и **x:y:[]** соответственно. Используя сопоставление с образцом, функцию **append** можно определить в виде:

```
append [] ys = ys
append (x:xs) ys = x: append xs ys
```

Очевидно, что такая запись более лаконична и понятна. Рассмотрим ещё один пример. Напишем функцию **member**, которая проверяет принадлежность элемента списку. Например:

```
member 1 [1,2,3] ⇒ True
member 4 [1,2,3] ⇒ False
```

Рекурсивное определение этой функции формулируется так. Базовый случай: если список пуст, то результат — **False**, поскольку пустому списку не принадлежит ни один элемент. Общий случай: элемент принадлежит списку, если он совпадает с его головой или принадлежит его хвосту. Это определение можно сразу перевести на Хаскель:

```
member _ [] = False
member x (y:ys) = x==y || member x ys
```

Здесь использован ещё один вид шаблона — *символ подчёркивания*. Он обозначает значение, которое не используется для получения результата. В стандартной библиотеке есть функция **elem**, которая делает то же самое, что и рассмотренная нами функция **member**. Оператор **`elem`** не ассоциативен и имеет приоритет **4**. Кроме того, есть функция **notElem**, которая проверяет, что элемент не принадлежит списку. Оператор **`notElem`** имеет ассоциативность и приоритет такие же, как и у оператора **`elem`**.

Задание на самостоятельную работу. Написать функцию, которая:

1. складывает элементы списка, **sum' [1,2,3]** Þ **6**;
2. вычисляет длину списка, **length' [1,2,3]** Þ **3**;
3. берёт от списка первые n элементов: **take' 2 [1,2,3,4]** Þ **[1,2]**;
4. исключает из списка первые n элементов: **drop' 2 [1,2,3,4]** Þ **[1,2]**;

5. делает «плоским» список списков, то есть для списка, имеющего двухуровневую структуру, строит одноуровневый список: **`concat' [[1,2],[3,4,5],[6]]` Þ `[1,2,3,4,5,6]`**.

2.3. ИСПОЛЬЗОВАНИЕ НАКАПЛИВАЮЩЕГО ПАРАМЕТРА

Мы рассмотрели сравнительно простые примеры функций, оперирующих со списками. Они имеют две особенности. Во-первых, они всегда результат строят конструктором **:**. Это значит, что результирующий список строится из уже вычисленных головы и хвоста. Во-вторых, входные списки обрабатываются слева направо, то есть на каждом шаге берётся голова и работа продолжается с хвостом входного списка. К сожалению, не все задачи имеют такую особенность. Например, возьмём задачу реверсирования списка, когда нужно построить список, элементы которого располагаются в обратном порядке по сравнению с входным списком.

Первое решение этой задачи можно сформулировать так. Базовый случай: если исходный список пуст, то результатом будет тоже пустой список. Общий случай: если исходный список имеет голову и хвост, то результатом будет отреверсированый хвост, в конец к которому добавлена голова. В этом определении требуется постановка элемента не в начало списка, а в конец. Для элемента **`x`** и списка **`xs`** эта операция определяется выражением **`reverse xs ++ [x]`**. То есть нужно соединить список **`reverse xs`** с одноэлементным списком **`[x]`**. На Хаскеле получим следующее определение.

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Из определения функции **append** видно, что для построения значения, вычисляемого в общем случае, список **reverse xs** нужно построить заново. Поскольку такую операцию нужно выполнять для каждого хвоста исходного списка, приведённый алгоритм будет весьма неэффективным.

Эффективность алгоритма можно было бы повысить, если бы функция имела локальную переменную для накопления результата, которую на каждом шаге можно было бы менять, добавляя к ней головы исходного списка. Образно говоря, если бы организовать стек, пустой вначале, в который на каждом шаге рекурсии заносить голову списка, то по исчерпании списка в стеке бы сформировался необходимый результат. В функциональном программировании можно использовать такой приём, смоделировав изменяемую переменную введением дополнительного параметра в функцию. Такой параметр называют накапливающим.

Рассмотрим, как можно определить реверсирование списка с использованием накапливающего параметра. Теперь функция **reverse** передаёт свою работу новой функции, назовём её **revrec**. Эта функция имеет

дополнительный накапливающий параметр, который в начале работы алгоритма содержит пустой список:

```
reverse xs = revrec [] xs
```

Функция **revrec**, в общем случае, когда исходный список имеет голову и хвост, добавляет голову к списку, связанному с накапливающем параметром и продолжает работу над хвостом. В базовом случае, когда исходный список пуст, результатом будет список, сформированный в накапливающем параметре. На Хаскеле получается следующее определение.

```
revrec acc (x:xs) = revrec (x:acc) xs
revrec acc [] = acc
```

Мы видим, что на каждом шаге алгоритма просто добавляется голова к списку, что является простой операцией. Таким образом, новый алгоритм получился значительно эффективнее первоначального. Подведём итог: накапливающий параметр моделирует локальную переменную, значение которой может меняться на каждом шаге рекурсии и доступно для вычисления окончательного результата функции.

2.4. АЛГОРИТМЫ СОРТИРОВКИ СПИСКОВ

Рассмотрим функции сортировки списков. На их примере увидим, как чётко и лаконично они могут быть написаны на Хаскеле. Возьмём сначала алгоритм сортировки вставкой. Его принцип формулируется так: чтобы отсортировать список, нужно вставить его голову в отсортированный хвост, не нарушая порядка сортировки. При этом нужно учесть, что пустой список уже отсортирован. Эту формулировку легко записать на Хаскеле:

```
isort (x:xs) = ins x $ isort xs
isort [] = []
```

Здесь используется функция **ins**, вставляющая элемент в список, не нарушая порядка сортировки. Она, в общем случае, определяется следующим образом. Если вставляемый элемент меньше головы списка, то результат — список, голова которого — вставляемый элемент, а хвост — исходный список. Иначе, получим список с головой исходного списка, а хвостом — результатом вставки заданного элемента в хвост исходного списка. Базовый случай: если список пуст, то результат — список, содержащий один вставляемый элемент. Эта формулировка на Хаскеле:

```
ins x l@(y:ys)
  | x<=y = x:l
  | otherwise = y: ins x ys
ins x [] = [x]
```

Базовый случай помещён в конце потому, что он встречается гораздо реже,

чем общий случай. Так повышается эффективность алгоритма.

2. 4.1 СОРТИРОВКА СЛИЯНИЕМ

В этом алгоритме задача сортировки списка делится на две подзадачи, каждая из которых работает со своей половиной исходного списка. Два результата сливаются без нарушения порядка сортировки в общий список. В этом определении используется функция слияния списков, которую можно определить рекурсивно следующим образом. Общий случай - если оба списка не пустые. Тогда, если голова первого списка меньше или равна голове второго списка, то результатом будет список с головой первого списка, а хвостом, полученным слиянием хвоста первого списка со вторым списком; иначе, результатом будет список с головой второго списка, а хвостом, полученным слиянием первого списка с хвостом второго списка. Базовые случаи — если один из списков пуст — результатом будет другой список. В соответствии с приведёнными определениями функция слияния упорядоченных списков на Хаскеле будет выглядеть так:

```
merge lx@(x:xs) ly@(y:ys)
                | x<=y = x: merge xs ly
                | x>y  = y: merge lx ys
merge xs [] = xs
merge [] ys = ys
```

Теперь функция сортировки может быть определена следующим образом. Если в списке меньше двух элементов (базовый случай), то он не нуждается в сортировке и может быть выдан в качестве результата, иначе (общий случай), нужно вычислить половину длины списка, разбить его на две половины, отсортировать их и слить в один список. На Хаскеле это определение примет вид:

```
mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs = merge (mergeSort front)
(mergeSort back)
  where size = length xs `div` 2
        front = take size xs
        back = drop size xs
```

2.4.2 БЫСТРАЯ СОРТИРОВКА

Алгоритм быстрой сортировки, как и алгоритм сортировки слиянием, делит задачу на две подзадачи, а из их решений строит окончательный результат. Итак, отсортированный список получается если:

1) взять голову исходного списка;
2) его хвост разделить на два списка: в одном собрать все элементы меньшие головы, а во втором — большие или равные;
3) отсортировать полученные списки и
4) собрать результирующий список так, чтобы в нём сначала шли отсортированные меньшие элементы, потом - голова списка, а за ней - отсортированные большие элементы.

Это определение общего случая. Под базовый случай попадают пустой список и список из одного элемента. Они не нуждается в сортировке. На Хаскеле получится следующий код:

```
qsort [] = []
qsort l@[_] = l
qsort (x:xs) = qsort l1 ++ x: qsort l2
  where (l1,l2) = splitList xs
```

Функция разделения списка **splitList**, чтобы многократно не передавать одно и то же значение, использует локальную функцию **split'**, которая возвращает пару списков.

```
splitList e l = split’ l
  where split’ [] = ([],[])
        split’ (x:xs)
          | x<e = (x:l1,l2)
          | otherwise = (l1,x:l2)
          where (l1,l2) = split’ xs
```

2.5. ПЕРЕБОР ЭЛЕМЕНТОВ СПИСКА

Алгоритмы, обрабатывающие списки, чаще всего последовательно перебирают их элементы, чтобы выбрать те из них, которые удовлетворяют нужному свойству и из выбранных элементов строят результирующий список.

Похожая ситуация наблюдается в теории множеств, когда новые множества строятся из уже известных множеств. Например, множество натуральных чётных чисел задаётся так:

{x| xÎN, чётное(n)},

где **N** - множество натуральных чисел.

Смысл этой записи таков: строится множество с элементами **x**, которые берутся из множества **N** при условии, что они чётные.

Другой пример: прямое (декартово) произведение множеств:

{<x,y>| xÎR, yÎS}.

Здесь определяется множество пар **<x,y>**, где **x** берётся из **R**, а **y** – из **S**.

Если множества представить списками, то подобные определения можно автоматически преобразовывать в выражения на Хаскеле. Поэтому в Хаскеле есть специальное синтаксическое расширение – выражение перебора элементов списка (по-английски «list comprehension»). С его использованием приведённые выше примеры будут записаны так:

```
[x | x<-n, even x]
```

```
[(x,y) | x<-r, y<-s]
```

В этих выражениях предполагается, что уже заданы списки **n**, **r** и **s**. Список n, представляющий собой бесконечное множество натуральных чисел, тоже бесконечен. В дальнейшем мы специально рассмотрим, как строятся бесконечные списки.

В общем виде выражение перебора элементов списка строится так:

[exp| qual$_1$, … qual$_n$], **n≥1**,

где

exp – любое выражение Хаскеля, определяющий элемент результирующего списка.

qual$_i$ – квалификаторы трёх типов:

1) Генератор **pat <- lexpr**, где **pat** – шаблон (образец) такой же, как и в определении функции, а **lexpr** – выражение, значение которого - список. То есть из списка **lexpr** по очереди выбираются значения по шаблону pat.
2) Локальное определение **let decls**, такое же, как выражение **let**, но без части **in**.
3) Булево выражение, называемое фильтром.

Рассмотрим теперь, как вычисляется выражение перебора элементов списка (перебора списков).

```
[a+b | a <-[1,2,3], odd a, b <-[3,4]]
```

Генератор **a <-[1,2,3]**	Фильтр **odd a**	Генератор **b <-[3,4]**	Результат **a+b**
a®1	**+**	**b®3**	**4**
		b®4	**5**
a®2	**-**		
a®3	**+**	**b®3**	**6**
		b®4	**7**

Ответ: **[4,5,6,7].**

2.6. СТРУКТУРЫ ДАННЫХ: КОРТЕЖИ

Кортеж – это упорядоченная последовательность элементов, имеющая фиксированную длину. Кортежи записываются перечислением элементов в круглых скобках. Примеры: **(1,2)**, **(True, 5, 8)**, **(3,[1,2,3])**. Первый пример — пара чисел: **1** и **2**, второй — тройка значений: булево и два числа, третий — пара: число и список чисел. В отличие от списков, для кортежа не требуется, чтобы его компоненты были однотипными.

Основополагающее значение имеют кортежи из двух компонентов — пары, поскольку из них можно собрать кортежи любой длины. Так, кортеж **(True, 5, 8)**можно представить вложенными парами: **(True, (5, 8))**. Поэтому пары относят к базовому языку, а остальные кортежи — к синтаксическим расширениям.

В программе кортеж можно построить, во-первых, записав в круглых скобках его компоненты через запятую, а во-вторых, применив функцию, обозначаемую кортежем без компонент (для пары — это **(,)**), к соответствующему числу аргументов. Примеры:

(,) 1 2 ⇒(1,2)

(,,,) 1 2 3 4 ⇒(1,2,3,4)

В обозначении кортежа можно выделить конструктор данных и его компоненты. Так, для кортежа **(True, 5, 8)** компонентами являются значения **True**, **5** и **8**, а консруктором - **(,,)** – то, что осталось. То есть конструктор кортежа – это оператор, который охватывает и разделяет свои операнды. Его можно использовать и как функцию[1], то есть записи **(True, 5, 8)** и **(,,) True 5 8** – эквивалентны. Чаще всего, конечно, используется первый вариант, но и второй, как мы увидим позже, имеет важное значение.

Для извлечения элементов из пары: первого и второго, используются соответственно стандартные функции **fst** и **snd**. Примеры:

fst (1,2) ⇒1

snd (1,2) ⇒2

Однако, основной приём работы с кортежами – сопоставление с образцом. Рассмотрим ещё один вид шаблона $(p^1,p^2,...p^N)$, где $p^1,p^2,...p^N$ – шаблоны. Такой шаблон сопоставим с N-местным кортежем. Например, определение стандартных функций **fst** и **snd** имеет вид:

```
fst (a,_) = a
snd (_,b) = b
```

Рассмотрим пример на использование кортежей. Напишем функцию, которая ставит в соответствие элементы двух списков. Это стандартная фуннкция **zip**, которая так называется потому, что её действие напоминает застёгивание молнии на одежде. Итак, эта функция, если, к примеру, получит два списка: **[1,1,2,3,5]** и **[0,1,2,3,4]** построит список пар: **[(1,0),(1,1),(2,2),(3,3),(4,5)]**.
То есть в данном случае она прономерует первые пять чисел Фибоначчи.

Рекурсивное определение фуннкции застёгивания сисков простое: в общем случае, когда оба списка не пустые, результатом будет список с головой, содержащей пару голов исходных списков, и с хвостом – застёгнутые хвосты исходных списков; иначе, в базовом случае, результатом будет пустой список. На Хаскеле получим следующее определение:

```
zip (x:xs) (y:ys) = (x,y): zip xs ys
zip _ _ = []
```

Легко написать функцию, которая подобным образом обрабатывает три списка и строит список троек. Она весьма полезна и поэтому включена в стандартную библиотеку под именем **zip3**. В качестве упражнения предлагается читателю самостоятельно определить эту функцию.

---

[1]В префиксной форме.

Одной из областей применения кортежей является построение функций, вычисляющих одновременно несколько результатов. Эту возможность мы рассмотрим на примере более эффективной функции, вычисляющей числа Фибоначчи. Прежде всего отметим, что рассмотренное нами ранее определение этой функции имеет недостаток: в нём заложено многократное вычисление одного и того же выражения. Это легко проверить, выполнив следующие преобразования:

```
fib 5 ⇒fib 3 + fib 4  ⇒fib 3 + fib 2 + fib 3
```

Мы видим в полученной сумме два одинаковых слагаемых **fib 3**, которые будут вычисляться по-отдельности и дадут заведомо одинаковые результаты. Более того, каждое слагаемое потребует отдельного вычисления выражения **fib 2**. Таким образом, число одинаковых вычислений будет стремительно расти.

Построим функцию, которая будет лишена указанного недостатка. Она будет использовать идею, что предыдущие два числа Фибоначчи будут сохраняться и передаваться для вычисления следующего числа. Один шаг такого вычисления представим функцией:

```
f (i1,i2,n) = (i2,i1+i2,n+1)
```

В ней входной параметр и вычисляемое значение – тройки вида **(i1,i2,n)**, где **n** – номер числа Фибоначчи, **i2** - его значение, **i1** - его предыдущее значение. Теперь для вычисления **n**–ного числа Фибоначчи , нужно **n-1** раз вызвать функцию **f**. Это сделает новая функция **fib**:

```
fib 0 = 1
fib n = fib' (1,1,1)
  where
  fib' t@(_,x,m)
    | n==m = x
    | otherwise = fib'(f t)
```

Она для аргумента **0** сразу выдаёт результат **1**, а для больших значений — рекурсивной функцией **fib'** делает последовательность шагов применения функции **f** до тех пор, пока не получится кортеж с требуемым третьим компонентом; тогда результатом будет значение второго компонента.

3. ЗАДАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ

3.1. Лабораторная работа « Введение в логическое программирование»

Цель работы

1. Изучение принципов логического программирования. Определение фактов, правил, запросов.

2. Приобретение навыков компиляции и выполнения логических программ.

3. Получение представления работе логических программ.

Контрольные вопросы

1. Понятие атома, предиката, структуры. Их обозначение в программе.

2. Как выполняется унификация?

3. Декларативный и процедурный смысл правил.

4. Выполнение конъюнктивных запросов.

5. Выполнение рекурсивных правил.

Задание на лабораторную работу

1. Написать на Прологе программу, в которой содержатся сведения о родственных отношениях.

1.1. Определить факты, в которых утверждаются сведения о том, кто чей родитель.

1.2. Определить факты утверждающие свойство принадлежности женскому полу.

1.3. Определить правила, задающие отношения сестра, двоюродная сестра, свекровь, предок по женской линии.

3.2. Лабораторная работа «Работа со списками в логическом программировании»

Цель работы

1. Изучить строение списков и основные операции над ними.

2. Приобретение навыков рекурсивного программирования.

3. Получение представления работе рекурсивных программ.

Контрольные вопросы

1. Как с помощью структур строятся списки?

2. Как выполняется унификация списков?

3. Каковы условия сходимости рекурсивных процессов на списках.

4. Выполнение рекурсивных запросов.

Задание на лабораторную работу

1. Написать на Прологе процедуры, реализующие основные операции на множествах. Множества представлены списками, в которых нет одинаковых элементов.

1.1. Дать рекурсивное определение операций объединения, пересечения и разности множеств.

1.2. Написать соответствующие процедуры на Прологе, используя отсечение.

1.3. Сформулировать по-русски рекурсивные определения, задав базовые и общие случаи.

3.3. Лабораторная работа « Комбинаторные алгоритмы на Прологе»

Цель работы

1. Углублённо изучить принципы рекурсивного программирования на списках.

2. Изучение на практике различных видов рекурсии.

3. Приобрести навыки рекурсивного определения алгоритмов.

Контрольные вопросы

1. Как работает оператор отсечения?

2. Как строятся альтернативные определения?

3. Как реализуется отрицание в Прологе?

4. Как построить список всех ответов на запрос?

Задание на лабораторную работу

1. Написать на Прологе процедуры, строящие для заданного списка все его префиксы, постфиксы, сегменты и перестановки.

1.1. Префикс – это начальная часть списка. Всеми префиксами списка [1,2,3] будет список [[],[1],[1,2],[1,2,3]].

1.2. Постфикс – это конечная часть списка. Всеми постфиксами списка [1,2,3] будет список [[],[3],[2,3],[1,2,3]].

1.3. Сегмент – это некоторая средняя часть списка. Всеми сегментами списка [1,2,3] будет список [[],[1],[2],[3],[1,2],[2,3],[1,2,3]].

3.4. Лабораторная работа «Использование грамматических правил в Прологе»

Цель работы

1. Изучить принципы грамматического разбора на Прологе.

2. Приобрести навыки синтаксического разбора предложений.

3. Применение на практике аппарата контекстно-свободных грамматик.

Контрольные вопросы

1. Как работает синтаксический анализатор, построенный по методу рекурсивного спуска?

2. Как строятся разностные списки?

3. Какие синтаксические расширения добавлены в Пролог для работы с грамматиками?

4. Как строится результат трансляции?

Задание на лабораторную работу

1. Написать на Прологе программу, разбирающую строку, в которой записано арифметическое выражение, и вычисляющее его результат.

3.5. Лабораторная работа « Работа с графами на Прологе»

Цель работы

1. Изучить принципы построения машинного представления графов.

2. Приобрести навыки работы с графами.

3. Применение на практике алгоритмов поиска на графах.

Контрольные вопросы

1. Как в Прологе определяются графы?

2. Как строятся алгоритмы поиска в графе?

3. Как оценивается эффективность этих алгоритмов?

4. Как осуществляется топологическая сортировка графа?

Задание на лабораторную работу

Написать на Прологе программу, заносящую узлы и дуги в граф, выполняющую поиск в ширину и в глубину.

3.6. Лабораторная работа «Решение задач методом поиска в ширину на Прологе»

Цель работы

1. Изучить принципы построения решателей интеллектуальных задач.

2. Приобрести навыки построения пространств поиска.

3. Применение на практике метода поиска в ширину для решения интеллектуальных задач.

Контрольные вопросы

1. Как в Прологе определяются задачи, предполагающие поиск в пространстве состояний?

2. Как строятся алгоритмы поиска в ширину?

3. Как оценивается эффективность этих алгоритмов?

Задание на лабораторную работу

Написать на Прологе программу, решающую задачу о миссионерах и каннибалах методом поиска в ширину.

3.7. Лабораторная работа «Работа со списками в функциональном программировании»

Цель работы

1. Изучить строение списков и основные операции над ними.

2. Приобретение навыков рекурсивного программирования.

3. Получение представления работе рекурсивных программ.

Контрольные вопросы

1. Как с помощью структур строятся списки?

2. Как выполняется сопоставление с образцом списков?

3. Каковы условия сходимости рекурсивных процессов на списках.

4. Как проверяются типы функций, работающих со списками.

Задание на лабораторную работу

1. Написать на Хаскеле функции, реализующие основные операции на множествах. Множества представлены списками, в которых нет одинаковых элементов.

2. Дать рекурсивное определение операций объединения, пересечения и разности множеств.

3. Написать соответствующие процедуры на Хаскеле, используя сопоставление с образцом.

4. Сформулировать по-русски рекурсивные определения, задав базовые и общие случаи.

3.8. Лабораторная работа «Комбинаторные алгоритмы на Хаскеле»

Цель работы

1. Углублённо изучить принципы рекурсивного программирования на списках.

2. Изучение на практике различных видов рекурсии.

3. Приобрести навыки рекурсивного определения алгоритмов.

Контрольные вопросы

1. Как определяются функции с использованием сопоставления с образцом?

2. Как строятся альтернативные определения?

3. Как реализуется перебор вариантов на Хаскеле?

4. Какое синтаксическое значение имеют отступы в программе на Хаскеле?

Задание на лабораторную работу

1. Написать на Хаскеле функции, строящие для заданного списка все его префиксы, постфиксы, сегменты и перестановки.

1.1. Префикс – это начальная часть списка. Всеми префиксами списка [1,2,3] будет список [[],[1],[1,2],[1,2,3]].

1.2. Постфикс – это конечная часть списка. Всеми постфиксами списка [1,2,3] будет список [[],[3],[2,3],[1,2,3]].

1.3. Сегмент – это некоторая средняя часть списка. Всеми сегментами списка [1,2,3] будет список [[],[1],[2],[3],[1,2],[2,3],[1,2,3]].

3.9. Лабораторная работа « Использование деревьев на Прологе»

Цель работы

1. Изучить принципы построения алгебраических типов данных.

2. Приобрести навыки обработки древовидных структур.

3. Применение на практике алгоритмов для двоичных деревьев.

Контрольные вопросы

1. Как в Прологе определяются алгебраические типы данных?

2. Как строятся алгоритмы поиска, вставки и удаления элементов двоичного дерева?

3. Как оценивается эффективность этих алгоритмов?

4. Как строится текстовое представление дерева?

Задание на лабораторную работу

Написать на Прологе программу, заносящую элементы в двоичное дерево, удаляющие элементы, осуществляющую поиск по ключу и распечатывающую дерево в удобном для просмотра виде.

3.10. Лабораторная работа «Использование комбинаторов для синтаксического анализа на Хаскеле»

Цель работы

1. Изучить принципы грамматического разбора на Хаскеле.

2. Приобрести навыки синтаксического разбора предложений.

3. Применение на практике аппарата контекстно-свободных грамматик.

Контрольные вопросы

1. Как работает синтаксический анализатор, построенный по методу рекурсивного спуска?

2. Как строятся комбинаторы альтернативы, конкатенации и замыкания Клини?

3. Какие операторы необходимы для работы с грамматиками?

4. Как строится результат трансляции?

Задание на лабораторную работу

Написать на Хаскеле программу, разбирающую строку, в которой записано арифметическое выражение, и вычисляющее его результат.

3.11. Лабораторная работа « Использование деревьев на Хаскеле»

Цель работы

1. Изучить принципы построения алгебраических типов данных.

2. Приобрести навыки обработки древовидных структур.

3. Применение на практике алгоритмов для двоичных деревьев.

Контрольные вопросы

1. Как в Хаскеле определяются алгебраические типы данных?

2. Как строятся алгоритмы поиска, вставки и удаления элементов двоичного дерева?

3. Как оценивается эффективность этих алгоритмов?

4. Как строится текстовое представление дерева?

Задание на лабораторную работу

Написать на Хаскеле программу, заносящую элементы в двоичное дерево, удаляющие элементы, осуществляющую поиск по ключу и распечатывающую дерево в удобном для просмотра виде.

3.12. Лабораторная работа « Работа с графами на Хаскеле»

Цель работы

1. Изучить принципы построения машинного представления графов.

2. Приобрести навыки работы с графами.

3. Применение на практике алгоритмов поиска на графах.

Контрольные вопросы

1. Как в Хаскеле определяются графы?

2. Как строятся алгоритмы поиска в графе?

3. Как оценивается эффективность этих алгоритмов?

4. Как осуществляется топологическая сортировка графа?

Задание на лабораторную работу

Написать на Хаскеле программу, заносящую узлы и дуги в граф, выполняющую поиск в ширину и в глубину.

3.13. Лабораторная работа «Грамматический разбор на Хаскеле»

Цель работы

1. Изучить принципы построения комбинаторов синтаксических анализаторов.

2. Приобрести навыки построения синтаксических анализаторов.

3. Применение на практике комбинаторов синтаксических анализаторов.

Контрольные вопросы

1. Как в Хаскеле определяются операции альтернативы, конкатенации, замыкания Клини?

2. Как строятся алгоритмы синтаксического разбора на основе комбинаторов?

3. Как оценивается эффективность этих алгоритмов?

4. Как строится абстрактное синтаксическое дерево?

Задание на лабораторную работу

Написать на Хаскеле программу, распознающую строки, соответствующие заданной грамматике и строящую абстрактное синтаксическое дерево.

3.14. Лабораторная работа «Решение задач методом поиска в глубину на Хаскеле»

Цель работы

1. Изучить принципы построения решателей интеллектуальных задач.

2. Приобрести навыки построения пространств поиска.

3. Применение на практике метода поиска в глубину для решения интеллектуальных задач.

Контрольные вопросы

1. Как в Хаскеле определяются задачи, предполагающие поиск в пространстве состояний?

2. Как строятся алгоритмы поиска в глубину?

3. Как оценивается эффективность этих алгоритмов?

Задание на лабораторную работу

Написать на Хаскеле программу, решающую задачу о кувшинах методом поиска в глубину.