Краткое введение в Haskell 98

1. Введение

Целью написания этого руководства являлось не обучение программированию, даже не обучение функциональному программированию. Данное руководство, скорее, предназначено как дополнение Haskell Report [4], который сам по себе является лишь сжатой программной документацией. Нашей целью является обеспечение введения в Haskell для людей, имеющих опыт работы, по крайней мере, с одним из других, желательно функциональных языков (или с «почти функциональным» языком, таким как ML или Scheme). Если читатель надеется научиться больше, чем стилю функционального программирования, мы очень рекомендуем *«Введение в функциональное программирование»* Берда [1] или *«Введение в системы функционального программирования, использующие Haskell»* Дэйви [2]. Для ознакомления с функциональными языками программирования и техникой, включающей некоторые основные принципы языков программирования, используемые в Haskell, смотрите [3].

Язык Haskell значительно развился с момента его возникновения в 1987. Это руководство по Haskell 98. Ранние версии языка сейчас вышли из употребления; пользователи Haskell поддерживают использование Haskell 98. Имеется также много расширений Haskell 98, которые широко реализуются. Они не являются еще формальной частью языка Haskell и не охвачены в этом руководстве.

Наша общая стратегия ознакомления со свойствами языка состоит в следующем: мотивировать идею, определить некоторые соглашения, привести несколько примеров и затем направить к Haskell Report для дальнейших деталей. Мы советуем, однако, чтобы читатель полностью игнорировал детали до полного прочтения Краткого Введения. С другой стороны файл Haskell The Standard Prelude (в Report в приложении A и стандартных библиотеках (находящихся в Library Report[5])), содержит множество полезных примеров кодов Haskell; мы советуем ознакомиться с ним после прочтения этого руководства. Это не только покажет читателю, как выглядит код Haskell, но также ознакомит с набором встроенных функций и типов Haskell.

В завершении, сайт Haskell ,http://haskell.org, имеет много информации о языке Haskell и его реализациях.

[Мы предпочли отказаться от изобилия лексических и синтаксических правил в начале руководства. Скорее, мы постепенно знакомим с ними читателей в процессе приведения примеров, и записываем их в скобках, как этот параграф. Такая организация сильно отличается от структуры Report, хотя Report и остается авторским ресурсом (ссылки вида “§2.1”, относятся к разделу Report).]

Haskell является *типизированным* языком:[1] типы в нем имеют ключевое значение, и новичку лучше всего с самого начала ознакомиться со всеми возможностями и полнотой типов в Haskell. Для людей, имеющих опыт работы с относительно «нетипизированными» языками, такими как: Perl, Tcl, или Scheme, приспособиться к этому может быть тяжело; для тех же, кто хорошо знаком с Java, C, Modula, или даже ML, приспособление должно быть легче, но потрудиться все же придется, потому что система типов Haskell сложнее и в какой-то степени богаче, чем в большинстве других языков. В любом случае *«типизированное программирование»* является частью программирования на Haskell, и этого невозможно избежать.

[1] Термин ввел Лука Карделли

2. Значения, типы и другие вещи

Так как Haskell вполне функциональный язык, все вычисления делаются посредством означивания *выражений* (синтаксических термов) до выходных *значений* (абстрактные сущности, которые мы рассматриваем как ответ). Каждое значение имеет связанный *тип*. (Интуитивно, мы можем представлять тип, как множество значений.) Примеры выражений включают атомарные значения, такие как: целое число 5, символ ‘a’, и функцию \x -> x + 1, а также структурированные значения, такие как список [1,2,3] и пара (‘b’,4).

Так же, как выражения обозначают значения, тип выражения есть синтаксический терм, который указывает тип переменных (или просто *типы*). Например, тип выражений включает атомарный тип Integer (бесконечное целое), Char (символ), Integer - > Integer (функция, отображающая Integer в Integer), а также структурированный тип [Integer] (однородный список целых чисел) и (Char, Integer) (пара - “символ, целое число”).

Все значения в Haskell являются “первичным классом” - они могут быть переданы в качестве аргументов функции, возвращаться в качестве результата, быть составными частями структуры данных и т.д. С другой стороны, типы в Haskell не являются “первичными классами”. Типы в какой-то степени описывают значение, связывание значений с их типами называется *приписыванием типов*. Используя приведенные выше примеры значений и их типов, запишем приписывание типов:

```
5  :: Integer
‘a’ ::  Char
inc :: Integer -> Integer
[1,2,3] :: [Integer]
(‘b’,4)  :: (Char, Integer)
```

“::” может быть прочтено как “имеет тип”.

Обычно функции в Haskell определяются рядом *уравнений*. Например, функция *inc* может быть определена одним уравнением:

```
inc n        =   n+1
```

Уравнение является примером *объявления*. Объявлением так же является *определение типа* (§4.4.1), с помощью которого мы можем объявить явный тип для *inc*:

```
inc          :: Integer -> Integer
```

Подробнее об определениях функций мы поговорим в разделе 3.

Когда мы хотим показать, что выражение e_1 “приводится” к другому выражению или значению e_2, мы пишем:

```
e1 => e2
```

К примеру:

```
inc (inc 3) => 5
```

Система статических типов Haskell определяет формальное отношение между типом и значением (§4.1.3). Система статических типов обеспечивает *безопасность* кода Haskell,

то есть не позволяет программисту неверно приписывать типы. Например, мы не можем в общем случае сложить два символа, так выражение ‘a’ + ‘b’ неверного типа. Главное преимущество статически типизированного языка хорошо известно: все ошибки, связанные с типами, обнаруживаются во время компиляции. Не все ошибки могут быть обнаружены системой; выражения, такие как 1/0 допустимы, но оно завершаются ошибкой во время выполнения. И все же система типов находит много программных ошибок во время компиляции, помогает пользователям лучше понять программу, и также позволяет оптимизировать код (к примеру, во время выполнения нет необходимости в приведении и проверке типов).

Система типов также обеспечивает правильность определенных пользователем типов. Фактически, система типов Haskell достаточно мощна, чтобы позволить избежать приписывания типа вообще;[2] мы говорим, что система типов самостоятельно выводит корректные типы для нас. Однако так как определения типов является очень эффективной формой документации, и помогают в обнаружении ошибок в коде, хорошей идеей является разумное размещение приписывания типа - такое, как мы дали для функции *inc*.

[Читатель отметит, что идентификаторы, обозначающие специфический тип, такой как Integer и Char, мы записали с прописной буквы, но идентификаторы, определяющие значения, такие как inc, записали со строчной буквы. Это не только соглашение: это предписано лексическим синтаксисом Haskell. Кстати, запись других символов также имеет значение: foo, fOo, fOO – все это различные идентификаторы.]

2.1.Полиморфные типы

Haskell также включает *полиморфные* типы – типы, которые являются универсальными в некотором смысле относительно всех других типов. Описания полиморфных типов по существу определяют семейства типов. К примеру, $(\forall a)[a]$– семейство типов, элементами которого являются списки элементов типа *a*. Списки целых чисел ([1,2,3]), список символов ([‘a’,’b’,’c’]), даже списки списков целых чисел, и др. - члены этого семейства. (Примечание, однако, [2,’b’] – не является примером, т.к. не существует отдельных типа, содержащего и 2 и ‘b’.)

[Идентификаторы вышеупомянутого типа называются *переменными по типам*, и записываются строчными буквами, чтобы отличить их от специальных типов таких, как Int. Кроме того, в Haskell имеется лишь универсально определенный тип, и поэтому не имеется никакой необходимости явно выписывать символ для универсального описания, и, таким образом, мы просто пишем [a] в вышеприведенном примере. Другими словами, все переменные по типам являются полностью универсально определенными.]

Списки – часто используемая в функциональных языках структура данных, и хорошее средство для объяснения принципов полиморфизма. Список [1,2,3] в Haskell фактически краткая запись для списка 1:(2:(3:[])), где [] – пустой список и “:” инфиксный оператор, который добавляет первый аргумент в начало второго аргумента (списка)[3]. Так как “:” является правоассоциативным, мы можем также писать список как 1:2:3:[].

В качестве примера определенной пользователем функции, которая работает со списками, рассмотрим задачу подсчета числа элементов в списке:

---

[2] С некоторыми исключениями, которые будут описаны позже

3 “:” и [] являются подобием cons и nil в Lisp, соответственно

```
length              :: [a] -> Integer
length  []          = 0
length   (x:xs)     = 1 + length xs
```

Это определение почти очевидно. Мы можем читать уравнение, как высказывание: «Длина пустого списка = 0, а длина списка, у которого первый элемент – x, а хвост – xs, = 1 + длина xs ».

Хотя и интуитивно, этот пример выдвигает на первый план важный аспект Haskell, который должен все же быть объяснен: *сопоставление с образцом*. Левые части уравнений содержат образцы, такие как [] и x:xs. При применении функции эти образцы интуитивно сопоставляются с фактическими параметрами ([] только соответствует пустому списку, а x:xs будет успешно сопоставлен с любым списком, содержащим, по крайней мере, один элемент, связывая x с первым элементом и xs с хвостом списка). Если сопоставление успешно, то в качестве результата возвращается вычисленная правая сторона. В противном случае, пробуются следующие уравнение, и если все уравнения не подходят, выдается ошибка.

Определение функций через сопоставление с образцом весьма распространено в Haskell, и пользователь должен ознакомиться с различными видами образцов, которые разрешены; мы вернемся к этой проблеме в разделе 4.

Функция *length* также является примером полиморфной функции. Она может быть применена к списку, содержащему элементы любого типа, например, [Integer], [Char], или [[Integer]].

```
length [1,2,3]          =>  3
length ['a', 'b', 'c']  =>  3
length [[1], [2], [3]]  =>  3
```

А вот две другие полезные полиморфные функции над списками, которые будут использоваться позже. Функция *head* возвращает первый элемент списка, функция *tail* возвращает все, кроме первого элемента.

```
head                :: [a] -> [a]
head (x:xs)         = x

tail                :: [a] -> [a]
tail (x:xs)         = xs
```

В отличие от *length*, эти функции не определены для всех возможных значений их аргументов. Ошибка выполнения происходит, когда эти функции применяются к пустому списку.

В случае с полиморфными типами мы видим, что некоторые типы являются в каком-то смысле более общими, чем другие, так как они определяют большее множество значений. К примеру, тип *[a]* более общий, чем *[Char]*. Другими словами, последний тип может быть получен из предыдущего типа подходящей заменой для *a*. В отношении этого обобщения, типовая система Haskell обладает двумя важными свойствами: во-первых, каждое выражение, имеющее правильно определенный тип, гарантированно имеет единственный родительский тип (как будет объяснено ниже), и во-вторых, родительский тип может быть найден автоматически (§ 4.1.3). В сравнении с языком с мономорфными

типами, таким как C, читатель увидит, что полиморфизм улучшает выразительность, а вывод типов уменьшает проблемы, связанные с типами, возникающие у программистов.

Родительским типом выражения или функции является наименее общий тип, который, интуитивно, “содержит в себе все реализации данного выражения”. Например, родительский тип функции *head* - *[a]->a*; в то время как типы *[b]->a*, *a->a*, или даже *a* являются слишком общими, а что-то подобное [Integer]->Integer является слишком узким. Существование единственных родительских типов – признак особенности *типовой системы Хиндли-Милнера*, которая формирует основу систем типов в Haskell, ML, Miranda[4], и нескольких других (главным образом функциональных) языков.

2.2.Определенные пользователем типы

Мы можем определить наши собственные типы в Haskell, используя конструкцию data, с которой мы вас ознакомим с помощью ряда примеров (§ 4.2.1).

Важный встроенный тип в Haskell, который имеет значения истинности:

data Bool = False | True

Здесь определен тип *Bool*, он имеет два значения: True (Истина) и False (Ложь). Тип Bool – пример (нуль-арного) *конструктора типов*, а True и False - (также нуль-арные) *конструкторы данных* (или просто *конструкторы*, для краткости).

Точно так же мы могли бы пожелать определить тип цвет:

data Color = Red | Green | Blue | Indigo | Violet

И Bool и Color – примеры перечисляемых типов, так как они состоят из конечного числа нуль-арных конструкторов данных.

А вот пример типа с только одним конструктором данных:

data Point a = Pt a a

Из-за единственности конструктора, типы, подобные Point, часто называют кортежными типами, так как это по существу есть декартово произведение (в данном случае двоичное) других типов[5]. Напротив, многоконструкторные типы, такие как Bool и Color, называются (непересекающимися) объединяющими или суммирующими типами.

Более важно, однако, то, что *Point* – пример полиморфного типа: для любого типа *t* он определяет тип точек в Декартовом пространстве с координатами типа *t*. Теперь ясно видно, что тип *Point* является унарным конструктором типа, так как из типа *t* он конструирует новый тип *Point t*. (В том же смысле, в примере списка, приведенном ранее, *[]* – также конструктор типа. Для любого типа *t*, мы можем “применить” *[]*, чтобы получить новый тип *[t]*. Синтаксис Haskell позволяет выражение *[] t* записать как *[t]*. Выражение “->” - также конструктор типа: для двух типов *t* и *u, t -> u* – тип функции, отображающий элементы типа *t* в элементы типа *u*.)

---

[4] “Miranda” является торговой маркой Research Software, Ltd

[5] Кортежи являются немного похожими на записи в других языках программирования

Обратите внимание, что тип конструктора двоичных данных - *Pt* есть *a->a->Point a*, и, таким образом, следующие типы также верны:

```
Pt   2.0   3.0          :: Point Float
Pt   'a'   'b'          :: Point Char
Pt True   False         :: Point Bool
```

С другой стороны, выражение такое, как *Pt 'a' 1* неверно, потому что *'a'* и *1* разного типа.

Важно видеть различие между применением *конструкторов данных* для получения *значения*, и применением *конструкторов типов* для получения *типа*; первое происходит во время выполнения программы, и именно так мы проводим вычисления в Haskell, в то время как второе происходит во время компиляции и является частью процесса обеспечения безопасности типов.

[Конструкторы типов, такие как *Point* и конструкторы данных, такие как *Pt* хранятся в отдельных пространствах имен. Это позволяет использовать одно и тоже имя и для конструктора типа и для конструктора данных, как в следующем примере:

```
data  Point a = Point a a
```

И хотя в начале это может показаться немного запутанным, это служит, чтобы сделать связь между типом и его конструктором данных более очевидным.]

2.2.1. Рекурсивные типы

Так же типы могут быть рекурсивными, как в случае с типом *бинарное дерево*:

```
data Tree a      = Leaf a | Branch (Tree a) (Tree a)
```

Здесь мы описали полиморфный тип бинарного дерева, элементы которого являются либо листовыми узлами, содержащими величину типа *a*, либо внутренними узлами ("ветвями"), содержащими (рекурсивно) два поддерева.

При чтении таких объявлений данных, стоит помнить, что *Tree* является конструктором типа, тогда как *Branch* и *Leaf* являются конструкторами данных. Помимо установления связи между этими конструкторами, такое объявление по существу определяет следующие типы для *Branch* и *Leaf*:

```
Branch :: Tree a -> Tree a -> Tree a
Leaf    :: a -> Tree a
```

С помощью этого примера мы определили тип, достаточно мощный для определения некоторых интересных (рекурсивных) функций, которые его используют. К примеру, мы хотим определить функцию *fringe*, которая возвращает список всех элементов листьев дерева слева направо. Обычно бывает полезным сначала написать тип новой функции; в данном случае типом будет *Tree a -> [a]*. Таким образом, *fringe* – полиморфная функция, которая, для любого типа *a*, преобразует деревья типа *a* в списки типа *a*. Далее приведено одно из возможных определений функции:

```
fringe                      :: Tree a -> [a]
fringe (Leaf x)             = [x]
fringe (Branch left right) = fringe left ++ fringe right
```

Здесь ++ - инфиксный оператор конкатенации двух списков (полное определение этого оператора будет приведено в разделе 9.1). Как и в примере с функцией *length*, приведенным ранее, функция *fringe* определена с использованием сопоставления образцов, только в данном случае мы видим образцы, включающие конструкторы, определенные пользователем: *Leaf* и *Branch*. [Обратите внимание на то, что аргументы функции начинаются со строчной буквы.]

2.3.Типовые синонимы

В целях удобства, в Haskell существует способ определения типовых синонимов, т.е. имен для используемых типов. Типовые синонимы создаются с использованием служебного слова *type* (§4.2.2). Ниже приведено несколько примеров:

```
type String   = [Char]
type Person   = (Name, Address)
type Name     = String
data Address = None | Addr String
```

Типовые синонимы не определяют новых типов, а всего лишь дают новые имена существующим типам. К примеру, тип *Person -> Name* эквивалентен типу *(String, Address)-> String*. Чаще всего новые имена короче синонимичных им типов, но это не единственное назначение типовых синонимов: являясь более мнемоническими, они так же могут улучшать читабельность программ; без сомнений, приведенные примеры это доказывают. Так же можно дать новые имена полиморфным типам:

```
Type AssocList a b = [(a, b)]
```

Это тип “ассоциативного списка”, который связывает величины типа *a* с величинами типа *b*.

2.4. Встроенные типы не являются особенными

Ранее мы представили несколько “встроенных” типов (списки, кортежи, целые числа и символы). Так же мы показали, как пользователем могут быть определены новые типы. Помимо специального синтаксиса, являются ли встроенные типы более особенными, чем типы, определенные пользователем? Ответ – нет. Специальный синтаксис существует для удобства и является следствием исторического соглашения, но не играет никакой семантической роли.

Мы можем подчеркнуть этот факт, если представим себе, как бы выглядели объявления встроенных типов, если бы можно было использовать специальный синтаксис для их определения. К примеру, тип *Char* мог быть записан как:

```
data Char                   = ‘a’ | ‘b’ | ‘c’ | ...   -- этот код не действителен
                            | ‘A’ | ‘B’ | ‘C’ | ...   -- в Haskell!
                            | ‘1’ | ‘2’ | ‘3’ | ...
                             ...
```

Такие имена конструкторов не являются синтаксически правильными; для правильной записи нам бы пришлось написать следующее:

```
data Char                = Ca | Cb  | Cc  | ...
                         | CA | CB | CC | ...
                         | C1 | C2  | C3  | ...
                          ...
```

И хотя эти конструкторы более краткие, они являются весьма нешаблонными для представления символов.

В любом случае, код на “псевдо-Haskell” в таком виде помогает нам разобраться в специальном синтаксисе. Мы видим, что *Char* – всего лишь пронумерованный тип, состоящий из большого числа конструкторов с нулевой арностью. Представление типа *Char* в таком виде, делает более очевидным то, что мы можем сопоставлять образцы с символами, так же, как мы сделали бы это с любым конструктором типов.

[Этот пример так же показывает использование *комментариев* в Haskell; символы -- и все последующие символы до конца строки игнорируются. Haskell так же позволяет вложенные комментарии, которые имеют форму {-…-} и могут появляться где угодно (§2.2).]

Аналогично мы могли описать *Int* (целые числа с фиксированной точностью) и *Integer*:

```
data Int     = -65532 | ... | -1 | 0 | 1 | ... | 65532          -- псевдокод
data Integer = ... -2 | -1 | 0 | 1 | 2 ...
```

где *-65532* и *65532* могут быть минимальным и максимальным целым с фиксированной точностью в данной реализации. В типе *Int* это гораздо большее перечисление, чем в типе *Char*, но он все равно конечен! В отличие от него, псевдокод для *Integer* генерирует *бесконечное* перечисление.

Так же типы можно легко определять следующим образом:

```
data (a, b)       = (a, b)                                  -- псевдокод
data (a, b, c)    = (a, b, c)
data (a, b, c, d) = (a, b, c, d)
  .                  .
  .                  .
  .                  .
```

Каждое объявление выше описывает кортеж определенной длины, причем *(...)* играет роль и в синтаксисе выражений (как конструктор данных) и в синтаксисе выражений типов (как конструктор типов). Вертикальные точки после последнего объявления предназначаются для генерирования бесконечного числа таких объявлений, показывая тем самым, что в Haskell разрешены кортежи любых типов.

Со списками так же легко управляться, и что интересно, они рекурсивны:

```
data [a] = [] | a : [a]                                     -- псевдокод
```

Теперь мы ясно видим то, что описали ранее: *[]* – пустой список, и *:* - инфиксный конструктор списков; таким образом, список [1, 2, 3] эквивалентен списку 1: 2: 3: []. (конструктор *:* - право ассоциативный.) Тип пустого списка *[] – [a]*, а тип конструктора *:* - *a->[a]->[a]*.

[То, как здесь определен конструктор *":"*, является синтаксически допустимым – инфиксные конструкторы разрешены в объявлениях *data* и отличаются от инфиксных операторов (в целях сопоставления образцов) тем, что они должны начинаться с *":"*.]

На данном этапе читатель должен отметить разницу между кортежами и списками, которую так ясно показывают приведенные выше определения. В частности, заметьте рекурсивность типа списков, представители которого являются гомогенными и имеют произвольную длину, и нерекурсивность (данного) типа кортежей, представители которого являются гетерогенными и имеют фиксированную длину. Так же теперь становятся ясными правила типизации кортежей и списков:

Для кортежа $(e_1, e_2, ..., e_n)$, $n \geq 2$, если e_i имеет тип t_i, тогда тип кортежа - $(t_1, t_2, ..., t_n)$.

Для списка $[e_1, e_2, ..., e_n]$, $n \geq 0$, все e_i должны иметь тип t, тогда тип списка – $[t]$.

2.4.1. Определители списков и арифметические последовательности

Как и в Lisp, списки широко распространены в Haskell, и, как и в других функциональных языках, существует много синтаксических приемов для их создания. Помимо описанных только что конструкторов списков, Haskell предусматривает выражение, называемое *определителем списка*, которое лучше всего описывается примером:

```
[ f x | x <- xs ]
```

Эта запись может быть прочитана как "список из всех таких *f x*, что *x* взято из *xs*." Сходство с нотацией – не случайность. Структура *x <- xs* называется *генератором*, структур таких может быть несколько:

```
[ (x, y) | x <- xs, y <- ys ]
```

Такой определитель списков формирует декартово произведение двух списков – *xs* и *ys*. Элементы выбираются так, как если бы генераторы были "вложены" справа налево (самый правый генератор будет изменяться быстрее всех); поэтому если *xs* – это *[1, 2]*, а *ys* – это *[3, 4]*, то результатом будет *[(1, 3), (1, 4), (2, 3), (2, 4)]*.

Помимо генераторов, разрешены булевы выражения, называемые *охраной*. Охрана накладывает ограничения на генерируемые элементы. К примеру, ниже приведено краткое определение всеми любимого алгоритма сортировки:

```
quicksort []       = []
quicksort (x: xs) =quicksort [y | y<-xs, y<x]
                              ++ [x]
                              ++ quicksort [y | y<-xs, y>=x]
```

Для большей поддержки использования списков, Haskell имеет специальный синтаксис для *арифметических последовательностей*, который лучше всего описывается рядом примеров:

```
[1 .. 10]      => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 3 .. 10]   => [1, 3, 5, 7, 9]
[1, 3 ..]      => [1, 3, 5, 7, 9, ...          (бесконечная последовательность)
```

Об арифметических последовательностях будет сказано подробнее в разделе 8.2, а о "бесконечных списках" – в разделе 3.4.

2.4.2. Строки

В качестве другого примера синтаксических приемов для встроенных типов заметим, что буквенная строка *"hello"*, вообще говоря, является краткой записью списка символов *['h','e','l','l','o']*. *"hello"* имеет тип *String*, где *String* – заранее определенный типовой синоним (который мы до этого приводили в качестве примера):

```
type String  = [Char]
```

Это означает, что мы можем использовать полиморфные списочные функции на строках. К примеру:

```
"hello" ++ " world"    =>    "hello world"
```

3. Функции

Поскольку Haskell является функциональным языком, мы ожидаем, что функции играют основную роль, и это действительно так.

Для начала, обратите внимание на определение функции, которая складывает два аргумента:

```
add      ::  Integer -> Integer -> Integer
add x y = x + y
```

Это пример *каррированной* функции[6]. Применение функции *add* имеет форму *add* e_1 e_2 и эквивалентно *(add* e_1*)* e_2, так как применение функции имеет *левую* ассоциативность. Другими словами, применение функции *add* к одному аргументу возвращает новую функцию, которая затем применяется ко второму аргументу. Это согласуется с типом *add*, *Integer -> Integer -> Integer,* который эквивалентен *Integer -> (Integer -> Integer);* то есть -> имеет ассоциативность *вправо*. Используя *add*, мы можем описать *inc* по-новому:

```
inc   = add 1
```

Это пример *частичного применения* каррированной функции, и один из случаев, когда функция возвращается как значение. Давайте рассмотрим случай, когда полезно передать функцию как аргумент. Хорошо известная функция *map* – прекрасный пример:

---

[6] Термин *каррирование* происходит от имени человека, который развил данную идею – Хаскелла Карри. Чтобы получить эффект *некаррирорванной* функции можно использовать *кортеж*:

```
add (x, y) = x + y
```

Но тогда мы видим, что такая версия функции *add* – на самом деле функция одного аргумента!

```
map             :: (a -> b) -> [a] -> [b]
map f []        = []
map f (x: xs) = f x : map f xs
```

[Применение функции имеет больший приоритет, чем инфиксный оператор, и поэтому правая часть второго уравнения вычисляется как *(f x) : (map f xs).*] Функция *map* – полиморфна, и ее тип четко указывает, что ее первый аргумент – функция; так же заметьте, что обе переменные, стоящие на месте *a*, должны иметь одинаковый тип (то же относится к *b*). В качестве примера применения *map*, мы можем инкрементировать элементы списка:

```
map (add 1) [1, 2, 3]   => [2, 3, 4]
```

Эти примеры демонстрируют природу первого класса функций, которые, когда используются таким образом, называются функциями *высших порядков*.

3.1.Лямбда-абстракции

Вместо использования уравнений для определения функций, мы так же можем определить их “анонимно” – через *лямбда-абстракцию.* К примеру, функция, эквивалентная *add*, может быть записана как *\ x -> \ y -> x+y.* Вложенные лямбда-абстракции могут быть записаны с использованием эквивалентной сокращенной нотации *\ x y -> x+y.* В действительности, уравнения:

```
inc x           =  x+1
add x y         = x+y
```

являются упрощенным видом:

```
inc             = \ x ->    x + 1
add             = \ x y -> x + y
```

Мы вернемся к этим эквивалентностям позже.

В общем случае, если *x* имеет тип t_1 и *exp* имеет тип t_2, то *\ x -> exp* имеет тип t_1 -> t_2.

3.2.Инфиксные операторы

Инфиксные операторы являются в действительности настоящими функциями и могут быть определены с использованием уравнений. Например, вот определение оператора конкатенации (связывания) списков:

```
(++)                    :: [a] -> [a] -> [a]
[]      ++  ys          = ys
(x:xs)  ++  ys          =x : (xs++ys)
```

[Лексически инфиксные операторы полностью состоят из "символов" в отличии от обычных буквенно-цифровых идентификаторов (§ 2.4). Haskell не имеет префиксных операторов, за исключением минуса (-), который является как инфиксным, так и префиксным.]

В соответствии с другим примером важным инфиксным оператором по функциям является оператор *композиции функции:*

```
(.)          :: (b->c) ->(a ->b) -> (a -> c)
f.g          =\x -> f (g x)
```

3.2.1. Разделения

Так как инфиксные операторы являются в действительности настоящими функциями, то также имеет смысл их частичное применение. В Haskell частичное применение инфиксного оператора называется *разделением.* Например:

```
(x+)     ≡     \y -> x+y
(+y)     ≡     \x -> x+y
(+)      ≡     \x y -> x+y
```

[Скобки обязательны.]

Последний вид разделения, приведенный выше, по существу приводит инфиксный оператор к эквивалентному функциональному значению, и является удобным в том случае, когда инфиксный оператор выступает в роли аргумента функции, как в функции *map* (+) [1,2,3] (читателю следует отметить, что возвращается список функций!). Это также необходимо при задании типа функции, как в примерах с операторами (++) и (.), приведенными ранее.

Сейчас мы можем увидеть, что сложением, определенным ранее, является именно (+), а увеличением на единицу является именно (+1)! Действительно, такие определения были бы совершенно точными:

```
inc          = (+1)
add          = (+)
```

Мы можем привести инфиксный оператор к функциональному значению, а можем ли мы пойти другим путем? Да - мы просто заключаем идентификатор, связанный с функциональным значением, в обратные кавычки. Например, *x `add` y* то же самое, что и *add x y*.[7] Некоторые функции лучше читаются таким образом.

Примером служит предопределенный предикат проверки вхождения в список elem; выражение x `elem` xs может быть интуитивно прочитано, как "x является элементом xs."

[Существуют специальные правила относительно разделений, содержащих префиксный/инфиксный оператор.]

В данный момент читатель может растеряться от такого количества способов определения функции. Решение сделать доступными все эти способы отчасти продиктовано чисто историческими договоренностями, а от части желанием обеспечить ясную последовательность действий (например, при обработке инфикса по сравнению с регулярными функциями).

---

[7] Заметьте, что add заключается в обратные кавычки, а не в апострофы, используемые для символов; т.е. 'f' это символ, тогда как `f` - инфиксный оператор. К счастью, большинство терминалов ASCII различают эти символы намного лучше, чем шрифт, использованный в этом документе.

3.2.2. Задание приоритетов

Приоритеты могут даваться любому инфиксному оператору или конструктору (включая те, что созданы из обычных идентификаторов, например, *`elem`*). Можно задать уровень приоритета от 0 до 9 (9 – наибольшее значение; предполагается, что обычные выражения имеют приоритет 10), и левостороннюю-, правостороннюю- или отсутствие ассоциативности. Например, задание приоритетов для ++ и **.** имеют такой вид:

infixr 5 ++
infixr 9 **.**

Такая запись задает правостороннюю ассоциативность, в первом случае с приоритетом 5, а в другом – 9. Левосторонняя ассоциативность задается с помощью *infixl*, а отсутствие ассоциативности через *infix*. Одной строчкой можно задать приоритет для нескольких операторов. Если конкретному оператору не дан приоритет и не указана ассоциативность, то по умолчанию он задан как *infix 9*. (Смотри § 5.9 - для подробного определения правил ассоциативности.)

3.3. Нестрогие функции

Предположим, что *bot* определена как

bot = *bot*

Другими словами, *bot* является бесконечным выражением. В общем, мы обозначим *значение* бесконечного выражения в виде $\perp$ (читается "основание"). Выражения, которые приводят к некоторым видам ошибок во время выполнения программы, такие как 1/0, также имеют это значение. Такая ошибка неисправима: программы не будут продолжать работу после этих ошибок. Ошибки, с которыми сталкивается система ввода/вывода, например ошибка признака конца файла, устранимы и обрабатываются различными способами. (Такая ошибка ввода/вывода в действительности вообще не является ошибкой, а скорее исключением. Об исключениях будет сказано подробнее в Разделе 7.)

Считается, что функция *f* является *строгой* в случае, если она, будучи примененой к бесконечному выражению, также не закончится. Другими словами, f является строгой, если значение *f bot* есть $\perp$. Для большинства языков программирования *все* функции строгие. Но в Haskell это не так. В качестве простого примера рассмотрим *const1*, функция *const1*, определяемая:

const1 x =*1*

В Haskell значение *const1bot* есть 1. Проще говоря, так как *const1* не “нуждается” в значении своего аргумента, то она никогда не пытается оценить его и, следовательно, никогда не примется за бесконечные вычисления. Исходя из этого, нестрогие функции также называют "ленивыми функциями" и говорят, что они означивают свои аргументы "лениво" или "по необходимости".

Так как ошибка и бесконечные значения в *Haskell* семантически являются одним и тем же, то все рассуждения, приведенные выше, относятся и к ошибкам. Например, *const1* (1/0) также будет 1.

Нестрогие функции чрезвычайно полезны во многих ситуациях. Основным преимуществом является то, что они освобождают программиста от многих забот в отношении порядка оценки. Дорогостоящие в вычислительном плане значения могут быть переданы в функции как аргументы без страха, что вычисления будут производиться в случаях, когда в них нет необходимости. Важным примером этого является возможность *бесконечных* структур данных.

Другим способом объяснения нестрогих функций является то, что Haskell вычисляет используя в большей степени *определения,* чем *присвоения,* встречающиеся в традиционных языках. Например, прочитайте определение

v = *1/0,*

как 'определите *v* как 1/0' вместо 'вычислите 1/0 и сохраните результат в *v*'. Только в случае, когда значение (определение) *v* необходимо, возникнет ошибка ‘деление на ноль’. Само по себе это определение не подразумевает какие-либо вычисления. Программирование с использованием присвоений требует пристального внимания за соблюдением порядка присвоений: смысл программы зависит от порядка, в котором эти присвоения исполняются. Определения, напротив, намного проще: они могут быть представлены в любом порядке без изменения смысла программы.

3.4."Бесконечные" структуры данных

Преимуществом нестрогого характера Haskell является то, что конструкторы данных также не строги. Это не должно вызывать удивление, так как конструкторы в действительности являются специальным видом функций (имеющих отличительную особенность, что они могут использоваться при сопоставлении с образом).

Например, конструктор для списков, (:), является нестрогим.

Нестрогие конструкторы допускают (в принципе) определение *бесконечных* структур данных. Вот пример бесконечного списка единиц *ones*:

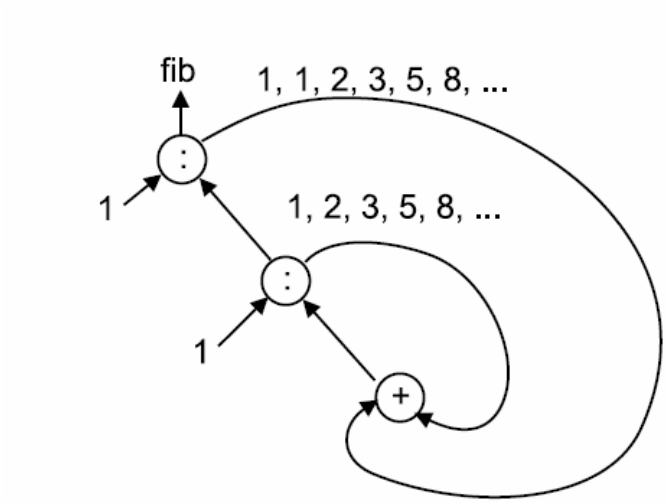


Рисунок 1: Циклическая последовательность Фибоначчи

ones = *1 : ones*

Возможно, более интересна функция *numsFrom*:

```
numsFrom n                    = n : numsFrom (n+1)
```

Таким образом, *numsFrom n* является бесконечным списком последовательных целых чисел, начиная с *n*. Из этого мы можем конструировать бесконечный список квадратов *(squares):*

```
squares                       = map (^2) (numsfrom 0)
```

(Заметьте использование разделения; ^ является инфиксным оператором возведения в степень.)

Конечно, в итоге мы хотим выделить некоторую определенную часть списка для реального вычисления, и в Haskell существуют группы предопределенных функций, которые помогают сделать это: *take, takeWhile, filter* и другие. Определение Haskell включает большой набор встроенных функций и типов - в" *The Standard Prelude*". Полностью "*The Standard Prelude"* включено в Приложение A описания Haskell; см. часть, *Preludelist*, содержащую множество полезных функций, использующих списки. Например, take берет n первых элементов из списка:

```
take 5 squares    =>      [0,1,4,9,16]
```

Определение *ones*, данное выше, является примером циклического списка. Во многих случаях леность оказывает значительное влияние на эффективность, так как такой список действительно может быть представлен в виде циклической структуры, экономя, таким образом, память.

В качестве другого примера применения зацикливания, последовательность Фибоначчи может быть эффективно вычислена как следующая бесконечная последовательность:

```
fib                           =1:1: [a+b / (a, b) <- zip fib (tail fib)],
```

где *zip* - функция "*The Standard Prelude"*, которая возвращает парное чередование аргументов ее двух списков:

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

Отметьте то, как бесконечный список *fib* определен через себя самого, словно он "гоняется за своим хвостом". Действительно, мы можем нарисовать картину вычислений, как показано на Рисунке 1.

Другое применение бесконечных списков смотрите в Раздел 4.4.

3.5. Функция ошибки

В Haskell имеется встроенная функция, называемая *error*, тип которой *String ->a*. Эта функция может показаться лишней. По типу эта функция выглядит так, как будто она возвращает значение полиморфного типа, о котором она ничего не знает, так как она никогда не получает значение этого типа в виде аргумента!

Фактически существует одно значение, "разделяемое" всеми типами: ⊥. Действительно, семантически это именно то, что всегда возвращается *error* (вспомните, что все ошибки возвращают значение ⊥). Однако, мы можем предполагать, что при разумной реализации функции *error* будет передаваться строка в качестве аргумента для диагностических целей.

Таким образом, эта функция полезна, если мы хотим завершить программу в случае, если что-то "пошло не так". Например, реальное определение head, взятое из The Standard Prelude:

```
head (x : xs)                    = x
head   []                        = error "head {Preludelist} : head []"
```

4. Case-выражения и сопоставление с образцами

Ранее мы рассмотрели несколько примеров сопоставления с образцами при определении функций - например, *length* и *fringe*. В этом разделе мы рассмотрим процесс сопоставления с образцом более подробно (§3.17)[8].

Образцы не являются "первоклассными;" существует только фиксированный набор различных видов образцов. Мы уже видели несколько примеров образцов *конструктора данных;* как *length*, так и *fringe*, определенные ранее, используют такие образцы, первая функция построена на использовании конструктора "встроенного" типа (*list*), вторая –на использовании типа, определенного пользователем (*Tree*). Действительно, сопоставление допускается при использовании конструкторов любого типа, определенного пользователем или нет. Включая кортежи, строки, цифры, знаки и т.д. Например, вот хитрая функция, которая производит сопоставление с кортежем "констант":

```
Contrived  ::   ([a], Char, (hit, Float), String, Bool) -> Bool
Contrived       ([],    'b',    (1, 2.0),      "hi",   True)     =  False
```

Этот пример так же демонстрирует, что допускается вложенность образцов (произвольной глубины).

Говоря техническим языком, *формальные параметры*[9] также являются образцами - посто у них *никогда не будет неудачи при сопоставлении со значением.* В качестве "побочного эффекта" успешного сопоставления формальный параметр связывается со значением, с которым его сопоставляют. Из этого следует, что образцам ни в одном уравнении не допускается иметь более одного вхождения одного и того же формального параметра (свойство, называемое линейностью §3.17, §3.3, §4.4.2).

Образцы, такие как формальные параметры, у которых не бывает несовпадения при сопоставлении, называются *неопровержимыми,* в отличие от *опровержимых* образцов, которые могут дать ошибку или частичное сопоставлении.

---

[8] 'Сопоставление с образцом в Haskell отлично от того, что принято в языках логического программирования, например, в Прологе; в частности, оно может рассматриваться как "одностороннее" сопоставление, тогда как Пролог допускает "двустороннее" сопоставление (через унификацию), вместе с возвратом в механизм вычисления.

[9] 'Описание .называет эти переменные.

Образец, использованный выше в *contrived* является опровержимым. Существует три вида неопровержимых образцов, два из которых мы введем сейчас (с другим мы повременим до Раздела 4.4).

As-образцы. Иногда удобно назвать образец для использования с правой стороны уравнения. Например, функция, которая дублирует первый элемент в списке, может быть записана так:

```
f (x : xs)                    =x:x:xs
```

(Вспомните, что ':' ассоциативна справа.). Заметьте, что *x : xs* появляется и в левой части уравнения как образец, и в правой части в выражении. Для улучшения читабелности мы могли бы написать *x : xs* только один раз, что можно сделать с помощью as-образцов следующим образом[10]:

```
f s@(x : xs)                  = x:s
```

Говоря техническим языком, as-образцы всегда дают успешное сопоставление, хотя суб-образцы (в данном случае *x : xs*) могли бы дать несовпадение.

Анонимные переменные. Другой распространенной ситуацией является сопоставление со значением, до которого нам нет, по сути, никакого дела. Например, функции *head* и *tail*, определенные в Разделе 2.1, могут быть переписаны в виде:

```
head (x : _)                  = x
tail    (_: xs)               = xs,
```

в которых мы “разрекламировали” тот факт, что нам нет дела, какова конкретная часть входного параметра . Каждое сопоставление анонимной переменной с чем угодно происходит успешно, но, в отличии от формального параметра, ничего не связывает; по этой причине в уравнении допускается более одной анонимной переменной.

4.1.Семантика сопоставления с образцом

Мы обсудили, как сопоставляются отдельные образцы, некоторые опровержимы, некоторые неопровержимы и т.д. Но что в целом управляет этим процессом? В каком порядке идут сопоставления? Что, если ни одно из них не пройдет успешно? Этот раздел отвечает на эти вопросы.

Сопоставление с образцом может показать совпадение, либо полное или частичное несовпадение. Успешное совпадение связывает все формальные параметры в образце. Частичное несовпадение возникает тогда, когда значение, требуемое образцом, содержит ошибку ($\perp$). Процесс совпадения как таковой происходит «сверху вниз, слева на право». Несовпадение выражения с образцом в любом месте в одном равенстве означает несовпадение всего равенства, тогда пробуется следующее равенство. Если ни одно из равенств не подошло, тогда значение функции $\perp$ и в процессе выполнения появляется сообщение об ошибке.

[10] Другое преимущество этого то, что обычное выполнение могло бы совсем изменить x:xs вместо того, что бы повторно использовать значение, с которым ведется сопоставление.

Например, если [1, 2] сопоставляется с [0, *bot*], тогда 1 не сопоставится с 0, и результат будет “несовпадение с образцом”. (Вспомните, что ранее говорилось, что переменная *bot*, определенная ранее, это переменная, связная с ⊥.) Но если [1, 2] сопоставляется с [*bot*, 0], тогда сопоставление 1 с *bot* даст частичное несовпадение (т.е. ⊥). С другой стороны верхние образцы могут содержать булевский ограничитель, как в данном определении функции, которое формирует условный знак числа:

```
sign x    | x > 0 = 1
          | x == 0 = 0
          | x < 0 = -1
```

Заметьте, что для того же образца может быть дана последовательность ограничителей; как и в случае с образцами, они рассматриваются сверху вниз, и первое, которое даст результат *True*, даст положительный результат сопоставления.

4.2.Пример

Правила сопоставления с образцом могут влиять на значения функций. На пример рассмотрим определение *take*:

```
take    0     _      = []
take    _     []     = []
take    n     (x:xs) = x : take (n-1) xs
```

и эту немного измененную версию (первые два равенства были переставлены):

```
take1   _     []     = []
take1   0     _      = []
take1   n     (x:xs) = x : take1 (n-1) xs
```

Теперь обратите внимание:

```
take   0 bot  ⇒   []
take1  0 bot  ⇒   ⊥
take   bot []  ⇒   ⊥
take1  bot []  ⇒   []
```

Мы видим, что *take* более “определена” по отношению ко второму аргументу, тогда как *take1* более “определена” для первого. В этом случае трудно определить, какое определение лучше. Просто необходимо помнить, что в конкретных приложениях это может иметь значение. (The Standard Prelude включает в себя определение, относящееся к take.)

4.3.Case-конструкции

Сопоставление с образцом предоставляет способ “быстрого контроля”, основанного на структурных свойствах значений.

В некоторых случаях мы бы не хотели определять *функцию* каждый раз, когда это необходимо, но до сих пор мы показали только, как делать сопоставление с образцом в определениях функций. Case-конструкции в Haskell предоставляют способ решить эту проблему. На самом деле, значение сопоставления с образцом в определениях функций

определено в Report в терминах case-конструкций, которые считаются более простыми. В частности, такое определение функции:

$$f\ p_{11} \ldots p_{1k} = e_1$$
$$\ldots$$
$$f\ p_{n1} \ldots p_{nk} = e_n$$

где p_{ij} - это образец, семантически эквивалентно такому:

$$f\ x_1\ x_2 \ldots x_k = case\ (x_1, \ldots, x_k)\ of\ (p_{11}, \ldots, p_{1k}) \rightarrow e_1$$
$$\ldots$$
$$(p_{n1}, \ldots, p_{nk}) \rightarrow e_n$$

где x_i – новые идентификаторы. (Для более подробного определения см. §4.4.2.)

На пример, определение *take*, данное ранее, эквивалентно такому:

```
take m ys           = case (m,ys) of
                        (0,_)       ->    []
                        (_,[])      ->    []
                        (n,x:xs)    ->    x : take (n-1) xs
```

Мы еще не отметили, что для совпадения типов необходимо, чтобы типы правых частей case-конструкции и множества равенств, составляющих определение функции, должны быть одинаковы; более точно, они все должны иметь общий основной тип.

Правила сопоставления с образцом для case-конструкций те же, что мы дали для определения функции, так что ничего нового здесь нет, кроме того, что следует отметить удобство, предлагаемое case-конструкциями. В самом деле, есть один очень распространенный способ применения case-конструкций, для которого существует специальный синтаксис: условные выражения. В Haskell, условные выражения имеют привычную форму:

if e_1 *then* e_2 *else* e_3

что на самом деле является краткой записью:

```
case e1 of      True    ->    e2
                False   ->    e3
```

Из выражения должно быть ясно, что e_1 должно иметь тип *Bool*, а e_2 и e_3 должны иметь одинаковые (но с другой произвольные) типы. Другими словами, если рассматривать *if-then-else* как функцию, она должна иметь тип *Bool -> a -> a -> a*.

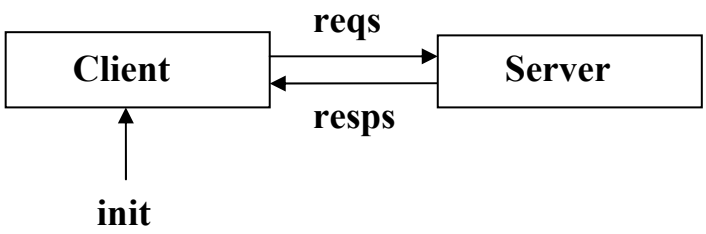
4.4.Ленивые образцы

В Haskell есть еще один тип образцов. Они называются ленивыми образцами и имеют вид *~pat*. Ленивые образцы неопровержимы: сопоставление значения *v* с *~pat* всегда дает полное совпадение, независимо от *pat*. Сразу добавим, что если идентификатор в *pat* далее используется в правой части, то он будет связный с той частью значения, которая получилась бы, если бы *v* совпала с *pat*, и $\perp$ в противном случае.

Ленивые образцы полезны в тех случаях, когда бесконечные структуры данных определены рекурсивно.

На пример, бесконечные списки - отличное средство для написания программ-симуляторов, и в таком случае бесконечные списки называются потоками. Рассмотрим простейший случай симуляции взаимодействия между сервером и клиентом, когда клиент посылает последовательности или запросы серверу, а сервер отвечает как-то на все запросы.

Данная ситуация проиллюстрирована на Рисунке 2. (Обратите внимание, что клиент использует начальное сообщение как аргумент.)



Если использовать потоки для симуляции последовательности сообщений, то код на Haskell, отвечающий диаграмме:

```
reqs                    = client init resps
resps                   = server reqs
```

Эти рекурсивные равенства являются прямой лексической транслитерацией диаграммы.

Теперь давайте предположим, что структура сервера и клиента выглядит как-то так:

```
client    init    (resp:resps)    = init : client (next resp) resps
server            (req:reqs)      = process req : server reqs
```

где мы предполагаем, что *next* это функция, возвращающая ответ сервера и определяющая следующий запрос, а *process* это функция, обрабатывающая запросы от клиента и возвращающая подходящий ответ.

К сожалению, у этой программы есть один серьезный минус: она не даст никакого результата! Проблема в том, что клиент, используемый в рекурсивном задании *reqs* и *resps*, принимается проводить сопоставление списка ответов еще до того, как он отправил свой первый запрос! Другими словами, сопоставление начинается “слишком рано”. Один из способов устранить ошибку – переопределить клиента так:

```
client init resps       = init : client (next (head resps)) (tail resps)
```

Хотя так и будет работать, это решение воспринимается не так просто, как то, которое было дано ранее. Лучше использовать ленивые образцы:

```
client init ~(resp:resps)       = init : client (next resp) resps
```

Из-за того, что ленивые образцы неопровержимы, сопоставление сразу же даст положительный результат, таким образом, обеспечивая “принятие” начального запроса,

позволяя сгенерировать первый ответ; теперь мы “принимаем” этот механизм, а об остальном позаботиться рекурсия.

В качестве примера продемонстрируем работу этой программы, определим:

```
init                    = 0
next resp               = resp
process req             = req+1
```

тогда получается:

take 10 reqs ⇒ *[0,1,2,3,4,5,6,7,8,9]*

В качестве ещё одного пример использования ленивых образцов рассмотрим определение чисел Фибоначчи, данное ранее:

```
fib                     = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

Мы можем попробовать переписать его, используя as-образец:

```
fib@(1:tfib)            = 1 : 1 : [ a+b | (a,b) <- zip fib tfib ]
```

Этот вариант *fib* имеет преимущество (небольшое), что не использует *tail* в правой части, так как это доступно в ”неструктурированной” форме в левой части в виде *tfib*.

[Этот тип равенств называется связывание образцом, потому что это равенство верхнего уровня, в котором левая часть целиком является образцом; т.е. и *fib* и *tfib* становятся связными еще при объявлении.]

Теперь, используя те же рассуждения, что и до этого, мы должны были бы поверить в то, что эта программа не выдает ответа. Удивительно, но она выдает, и причина проста: в Haskell предполагается, что связывания образцом имеют по умолчанию ~ перед собой, именно в этом причина самого обычного поведения, ожидаемого от связывания образцом, и именно это позволяет избегать некоторых аномальных ситуаций, рассмотрение которых выходит за рамки этого курса. Таким образом, мы видим, что ленивые образцы играют важную роль в Haskell, даже если только по умолчанию.

4.5. Локальные определения и ограничение области связывания

Часто бывает удобно создать область видимости связанную с некоторым выражением с целью введения локальных связываний, которые действуют только внутри этой области. В Haskell существуют два способа добиться этого:

Let-конструкции. Let-конструкции в Haskell полезны в тех случаях, когда необходимо ввести множество локальных определений. Рассмотрим простой пример:

```
let y       = a*b
f x         = (x+y)/y
in f c + f d
```

Множество связываний, созданных с помощью let-конструкции, взаимнорекурсивно.

Where-клозы. Иногда удобно ограничивать связывания внутри области с несколькими охранными выражениями, для которых необходимы where-клозы:

```
f x y       | y>z           = ...
            | y==z          = ...
            | y<z           = ...
  where z = x*x
```

Заметьте, что это невозможно сделать при помощи let-конструкций, которые ограничивают область связывания только выражениями, входящими в них. Where-клозы допустимы только на верхних уровнях множества равенств или case-выражений.

В where-клозах используются те же свойства и ограничения, что и в let-конструкциях.

Эти две формы ограничения области связывания кажутся похожими, но не стоит забывать, что let-конструкции - это выражения, тогда как where-клозы – это часть объявления функций и case-выражений.

4.6.Двумерный синтаксис

Возможно читатель задумывался, как программы в Haskell избегают использования разделителей для указания конца равенства, объявления и т.д. Например, рассмотрим let-выражение из предыдущего раздела:

```
let y       = a*b
    f x     = (x+y)/y
in f c + f d
```

Как анализатор понимает, что это нужно разбирать именно таким образом, а, например, не так:

```
let y       = a*b f
    x       = (x+y)/y
in f c + f d
?
```

Дело в том, что Haskell использует двумерный синтаксис, который основан на объявлениях, “выстроенных в колонки”. Обратите внимание, что в вышеприведенном примере, *y* и *f* начинаются в одной колонке. Более детально правила двумерного синтаксиса изложены в Report (§2.7, §B.3), но на практике использование двумерного синтаксиса скорее интуитивное. Главное помнить две вещи:

Во-первых, символ, следующий за одним из ключевых слов where, let или of определяет начало колонки объявления выражений, написанных в where, let или case конструкциях (это правило так же применимо к where, когда оно используется в определении классов и экземпляров классов, речь о которых пойдет в Разделе5). Таким образом, мы можем начать объявление на той же строке, что и ключевое слово, на следующее строке и т.д. (Ключевое слово do, о котором мы расскажем ниже, так же использует двумерный синтаксис).

Во-вторых, необходимо убедиться в том, что начало колонки клоза находится правее, чем начало колонки, связанной с клозом, непосредственно включающим данный (в противном случае он будет трактоваться неоднозначно). Объявление заканчивается, когда что-то появляется слева от начала колонки, ассоциированной с данной областью связывания[11].

В общем, двумерный синтаксис является краткой формой для явного механизма группировки, который заслуживает упоминания, т.к. может быть полезен в конкретных обстоятельствах. Так что пример с let, приведенный выше, можно переписать так:

```
let { y      = a*b
    ; f x = (x+y)/y
}
in f c + f d
```

Обратите внимание на явное указание фигурных скобок и точек с запятой. Такая явная запись полезна в случае, когда более одного определения необходимо расположить на одной строчке; например, следующее выражение является корректным:

```
let y = a*b; z = a/b
  f x = (x+y)/z
in f c + f d
```

Еще один пример расширения двумерного синтаксиса явными разделителями находится в §2.7.

Использование двумерного синтаксиса сильно уменьшает синтаксический хаос, связанный с последовательными объявлениями, т.о. увеличивая читабельность. Его просто выучить, и его использование приветствуется.

5. Классы типов и перегрузка

Существует свойство системы типов в Haskell'е, которое отличает его от других языков программирования. Вид полиморфизма, о котором до сих пор велся разговор, обычно называют параметрическим полиморфизмом. Существует и другой вид, называемый *ad hoc* полиморфизм, более известный, как *перегрузка*. Вот несколько примеров перегрузки:

- Литералы 1, 2, 3 и т.д. используются как для записи целых чисел, так и для записи чисел произвольной точности.
- Арифметические операции, например, сложение — знак "+" используются для работы с данными различных типов.
- Оператор сравнения (в Haskell'е знак двойного равенства — "==") используется для сравнения данных различных типов.

Отметим, что такое перегруженное поведение операций различно для каждого типа данных (зачастую такое поведение вообще может быть неопределенно или определено ошибочно), в то время, как в параметрическом полиморфизме тип данных, вообще говоря, не важен (в функции **fringe**, например, неважно, какого типа элементы находятся в

[11] в Haskell одна табуляция равна 8 пробелам; таким образом, надо соблюдать некоторую осторожность при использовании редакторов, которые, возможно, считают иначе.

листьях дерева). В Haskell'е классы типов представляют структурированный метод контроля *ad hoc* полиморфизма или перегрузки.

Рассмотрим простой, но важный пример: операция сравнения. Существует большое множество типов, для которых можно и целесообразно переопределить операцию сравнения, но для некоторых типов эта операция бесполезна. Например, сравнение функций бессмысленно, функции необходимо вычислять и сравнивать результаты этих вычислений, однако, например, может возникнуть необходимость сравнивать списки[12]. Для этого рассмотрим функцию elem, которая проверяет, присутствует заданный элемент в заданном списке, или нет:

```
x `elem` []          = False
x `elem` (y:ys)      = x == y || (x `element` ys)
```

[в целях стиля была выбрана инфиксная форма для определения elem. Операторы "==" и "||" являются инфиксными операторами, равенства и выбора.]

Здесь видно, что у функции elem тип (a → [a] → Bool), но при этом тип операции "==" должен быть (a → a → Bool). Т.к. переменная типа a может обозначать любой тип (в том числе и список), целесообразно переопределить операцию "==" для работы с любым типом данных.

Более того, как уже отмечалось ранее, даже если бы "==" был определен для всех типов, сравнение двух списков сильно отличается от сравнения двух целых чисел. В этом смысле предполагается, что "==" должен *перегружаться* для того, чтобы продолжить выполнение таких различных задач.

Классы типов являются решением этой проблемы в Haskell'е. Они позволяют объявлять какие типы являются *экземплярами* какого класса, и представлять определение перегружаемых *операций*, связанных с классом. Для того чтобы рассмотреть этот механизм в действии далее определяется класс типов, содержащий оператор равенства:

```
class Eq a where
      (==)    :: a -> a -> Bool
```

Символ "Eq" является именем определяемого класса, а "==" является единственной операцией в классе. Эта запись может быть прочитана следующим образом: "Тип a является экземпляром класса Eq, если для этого типа перегружена операция сравнения "==" соответствующего типа". (Необходимо отметить, что операция сравнения должна быть определена на паре объектов одного и того же типа.)

Ограничение того факта, что тип a должен быть экземпляром класса Eq записывается как Eq a. Поэтому выражение (Eq a) не является описанием типа, но оно описывает ограничение, накладываемое на тип a, и это ограничение называется *контекстом*. Контексты располагаются перед определением типов. Например, результатом объявления класса является присвоение следующего типа для "==":

```
(==)    :: (Eq a) => a -> a -> Bool
```

[12] Здесь речь идет о сравнении значений, а не сравнении указателей, как, например, это сделано в Java.

Эта запись может быть прочитана как “Для каждого типа a, который является экземпляром класса Eq, определена операция “==”, которая имеет тип (a → a → Bool)”. Эта операция должна быть использована в функции elem, поэтому ограничение распространяется и на неё. В этом случае необходимо явно указывать тип функции:

```
elem   :: (Eq a) => a -> [a] -> Bool
```

Эта запись может быть прочитана как “Для каждого типа a, который является экземпляром класса Eq, определена операция elem, которая имеет тип (a → [a] → Bool)”. Этой записью декларируется тот необходимый факт, что функция elem определена не для всех типов данных, но только для тех, для которых определена соответствующая операция сравнения.

Однако теперь возникает проблема определения того, какие типы являются экземплярами класса Eq. Для этого в Haskell’е есть служебное слово instance. Например, для того, чтобы предписать, что тип Integer является экземпляром класса Eq, необходимо написать:

```
instance Eq Integer where
      x == y = x `integerEq` y
```

В этом выражении определение операции “==” называется определением *метода*. Функция integerEq может быть любой (и не обязательно инфиксной), главное, чтобы у нее был тип (a → a → Bool). В этом случае, скорее всего, подойдет примитивная функция сравнения двух натуральных чисел. В свою очередь прочитать написанное выражение можно следующим образом: “Тип Integer является экземпляром класса Eq, и далее приводится определение метода, который производит сравнение двух объектов типа Integer”. Дав такое объявление, теперь можно определить операцию сравнения и для чисел с фиксированной точностью, используя “==”. Подобным образом:

```
instance Eq Float where
      x == y = x `floatEq` y
```

позволяет сравнивать числа с плавающей точкой, используя “==”.

Рекурсивные типы, например Tree, определенные ранее, также могут обрабатываться:

```
instance (Eq a) => Eq (Tree a) where
      Leaf a    == Leaf b           = a == b
      (Branch l1 r1) == (Branch l2 r2)   = (l1 == l2) && (r1 == r2)
      _                 == _              = False
```

Следует отметить, что контекст Eq a в первой строчке – обязателен, т.к. элементы в листьях (типа a) сравнивают во второй строчке. Дополнительным ограничением, по сути, является то, что можно сравнивать деревья элементов **a** до тех пор, пока известно как сравнивать **a**. Если контекст был опущен из объявления экземпляра, то результатом была бы статическая ошибка типа.

Haskell Report, в особенности Prelude, содержит большое число полезных примеров для классов типов. В действительности, класс Eq определяют немного шире, чем он был определен ранее:

```
class Eq a where
        (==), (/=)              :: a -> a -> Bool
        x /= y          = not (x == y)
```

Это пример класса с двумя операциями, равенства и неравенства. Также демонстрируется использование *метода по умолчанию*, в случае операции неравенства /=. Если для конкретной операции метод опущен в объявлении экземпляра, то вместо него используется метод по умолчанию, определенный в объявлении класса, если он там существует. Например, три экземпляра класса Eq, определенные ранее, будут отлично работать с вышеприведенным объявлением класса, приводящим как раз правильное определение неравенства, которое необходимо: логическое отрицание равенства.

Haskell также поддерживает понятие *наследования*. Например, желательно определить класс Ord, который наследует все операции, имеющиеся в Eq, но дополнительно имеет набор операций сравнения, а также функции максимума и минимума:

```
class (Eq a) => Ord a where
        (<), (>), (<=), (>=)    :: a -> a -> Bool
        min, max                        :: a -> a -> a
```

Обратите внимание на контекст в этом объявлении класса. Мы говорим, что Eq является надклассом для Ord (обратно, Ord является подклассом Eq), а любой тип, являющийся экземпляром Ord, должен быть экземпляром Eq. (В следующем разделе дается более полное определение Ord, взятое из Prelude.)

Исключительным преимуществом таких вложений классов являются более короткие контексты: выражение типа для функции, которая использует операции как из Eq, так и из Ord классов, может использовать контекст (Ord a), более правильно, чем (Eq a, Ord a), т.к. Ord “предполагает” Eq. Более важно то, что методы для операций подклассов могут принимать существование методов для операций надклассов. Например, объявление Ord в Prelude содержит такой метод по умолчанию для (<):

```
x < y                           = x <= y && x /= y
```

В качестве примера использования Ord, приведем описание функции **quicksort**, определенной в Разделе 2.4.1:

```
quicksort                       :: (Ord a) => [a] -> [a]
```

Другими словами, quicksort работает только со списками, тип которых принадлежит классу Ord. Такое отнесение к типу для quicksort происходит из-за использования в его определении операторов сравнения < и >=.

Haskell также допускает *множественное наследование*, т.е. классы могут иметь более одного надкласса. Например, объявление:

```
class (Eq a, Show a) => C a where ...
```

создает класс C, который наследует операции как из Eq, так и из Show.

В Haskell’е методы классов обрабатываются как объявления верхнего уровня. Они разделяют то же самое пространство имен, как обычные переменные; не может быть

использовано имя для обозначения, как метода класса, так и переменной или методов в разных классах.

Контексты допускается использовать в объявлении данных; смотри §4.2.1.

Методы класса могут иметь дополнительные ограничения класса по любой переменной типа, за исключением переменной, определяющей текущий класс. Например, в таком классе:

```
class C a where
    m                    :: Show b => a -> b
```

метод m требует, чтобы тип b находился в классе Show. Однако метод m не мог бы накладывать какое-либо ограничение класса на тип a. Эти ограничения должны были бы стать частью контекста в объявлении класса.

До сих пор, мы использовали типы “первого порядка”. Например, конструктор типа Tree до сих пор всегда был в паре с одним аргументом, как в Tree Integer (дерево, содержащее величины Integer) или Tree a (представляющее семейство деревьев, содержащих величины a). Но Tree само является конструктором типа, и по существу принимает тип, как аргумент, а возвращает тип, как результат. В Haskell’е не существует величин, которые имеют такой тип, но такие “более высокого порядка” типы могут быть использованы в *объявлениях классов*.

Рассмотрим нижеописанный класс Functor (взятый из Prelude):

```
class Functor f where
    fmap                 :: (a -> b) -> f a -> f b
```

Функция fmap обобщает функцию map, использованную до этого. Обратите внимание, что переменная типа f применяется к другим типам в f a и f b. Поэтому ожидается, что данное определение должно быть связано с типом, наподобие Tree, который может быть применен к одному аргументу. Экземпляр Functor’а для типа Tree был бы:

```
instance Functor Tree where
fmap f (Leaf x) = Leaf (f x)
fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)
```

Такое объявление экземпляра говорит, что Tree, нежели Tree a, является экземпляром класса Functor. Такая возможность довольно полезна и демонстрирует здесь способность описывать родовые “контейнерные” типы, позволяющие функциям, таким как fmap, работать одинаково с произвольными деревьями, списками и другими типами данных.

[Применение типов описаны таким же образом, как применение функций. Тип T a b анализируется синтаксически в виде (T a) b. Типы, такие как кортежи, использующие специальный синтаксис, могут быть написаны в альтернативном стиле, который допускает каррирование. Для функций (→) является конструктором типа; типы f→g и (→) f g являются одинаковыми. Подобным образом одинаковыми являются типы [a] и [] a. Для кортежей конструкторами типов (также как конструктором данных) являются (,), (,,) и т.д.]

Как известно, система типов обнаруживает ошибки в отнесении к типу, имеющиеся в выражениях. А что по поводу ошибок из-за деформированных выражений типов? Выражение (+) 1 2 3 приводит к ошибке, так как (+) принимает только два аргумента. Подобным образом тип Tree Int Int должна создавать некоторый вид ошибок, так как тип Tree принимает только один единственный аргумент. Итак, как Haskell обнаруживает деформированные выражения типа? Ответом является вторая система типов, гарантирующая корректность типов! Каждый тип имеет соотнесенный *вид*, который гарантирует, что тип использован корректно.

Выражения типа делятся на различные *виды*, которые принимают одну из двух возможных форм:

- Символ * представляет соотнесенный с конкретными объектами данных вид типа. То есть, если значение *v* имеет тип *t*, то вид *v* должен быть *.
- Если k_1 и k_2 – виды, тогда $k_1 \rightarrow k_2$ – вид типов, который принимает тип вида k_1 и возвращает тип вида k_2.

Конструктор типа Tree имеет вид: * → *; тип Tree Int имеет вид *. Члены класса Functor должны иметь вид * → *; ошибка в определении вида была бы получена из объявления, например

instance Functor Integer where ...

так как Integer имеет вид *.

Виды в программах Haskell'а непосредственно не появляются. Компилятор делает заключение по видам до выполнения проверки типа без какой-либо необходимости в "объявлениях вида". Виды остаются на заднем плане в Haskell программе, за исключением того, когда ошибочная запись типа ведет к ошибке типа. Виды достаточно просты, поэтому компиляторы должны предоставлять сообщения с описанием ошибок при появлении конфликтных ситуаций по виду. Смотри §4.1.1 и §4.6 для дополнительной информации по видам.

Другая перспектива. Перед тем как перейти к последующим примерам использования классов типов стоит указать на два других мнения по поводу классов типов в Haskell'е. Первое – по аналогии с объектно-ориентированным программированием (ООП). В следующем заявлении, просто заменяющим *класс* на *класс типов*, а объект на тип, ООП выдает истинное резюме по механизму Haskell'а:

"*Классы* фиксируют общие наборы операций. Частый *объект* может быть экземпляром класса и будет иметь метод, соответствующий каждой операции. *Классы* могут быть размещены иерархически, формируя понятия *надклассов* и *подклассов* и допуская наследование операций/методов"

В отличие от ООП должно быть понятно, что типы не объекты и, в частности, не существует понятие об изменчивом внутреннем состоянии объекта или типа. Преимуществом над некоторыми языками ООП является то, что в Haskell'е методы полностью защищены по типу: любая попытка применить метод к значению, чей тип находится не в требуемом классе, будет обнаруживаться во время компилирования, а не при работе программы. Другими словами методы не "просматриваются" в ходе прогона программы, а просто пропускаются как функции более высокого порядка.

Другая перспектива может быть получена при рассмотрении взаимосвязи между параметрическим и *ad hoc* полиморфизмом. Мы показали, как полезен параметрический полиморфизм при определении семейств типов посредствам универсальной

квантификации по всем типам. Однако, иногда эта универсальная квантификация слишком широка, поэтому хотелось бы квантифицировать по меньшему набору типов, например, по типам, чьи элементы можно сравнить. Типы классов могут быть рассмотрены как обеспечение структурированного способа сделать именно это. По существу, можно считать параметрический полиморфизм также в качестве вида перегрузки. Верно то, что перегрузка имеет место по всем типам без исключения, а не по ограниченному набору типов (т.е. классу типов).

Сравнение с другими языками. Классы, используемые Haskell'ом подобны классам, используемым в других объектно-ориентированных языках, например, в C++ и Java. Однако, существуют несколько существенных отличий:

- Haskell отделяет определение типа от определения метода, соотнесенного с этим типом. Класс в C++ и Java обычно определяет как структуру данных (переменные элементы), так и функции, связанные с этой структурой (методы). В Haskell'е эти определения разделены.
- Определения методов в Haskell'е соответствуют виртуальным функциям C++. Каждый конкретный экземпляр класса должен переопределять методы класса; умолчания класса соответствуют определениям умолчаний для виртуальной функции в базовом классе.
- Классы в Haskell'е больше всего похожи на интерфейсы Java. Как и определение интерфейса, классы в Haskell'е предоставляют протокол использования объекта, вместо определения самих объектов.
- Haskell не поддерживает стиль перегрузки функции, используемый в C++, когда функции с одним и тем же именем получают данные различных типов для обработки.
- Типы объектов в Haskell'е не могут быть выведены неявно. В Haskell'е не существует универсального базового класса, такого как Object, значения которого могут вводиться и выводиться.
- C++ и Java добавляют идентифицирующую информацию (такую как VTable) для представления объекта при работе программы. В Haskell'е такая информация закрепляется логически, а не физически к значениям по всей системе типов.
- Не существует контроля за доступом (нет публичных и защищенных методов), встроенного в систему классов Haskell'a. Вместо этого должна быть использована система модулей для hide or reveal класса.

6 И снова типы

Здесь мы проведем экспертизу некоторых еще более перспективных аспектов объявлений типов.

6.1 Объявление Нового типа

Общей практикой программирования является определение типа, представление которого идентично существующему типу, но который имеет отдельную идентифицирующую особенность в системе типов. В Haskell объявление **newtype** создает новый тип из существующего. Например, натуральные числа могут быть представлены типом **Integer**, используя следующее объявление:

```
newtype Natural  =  MakeNatural Integer
```

Это объявление создает полностью новый тип, **Natural**, чей конструктор только содержит единственный **Integer.** Конструктор **MakeNatural** осуществляет преобразование между **Natural** и **Integer:**

```
toNatural                   :: Integer -> Natural
toNatural x | x < 0         = error "Can't create negative naturals!"
            | otherwise     = MakeNatural x
fromNatural                  :: Natural -> Integer
fromNatural (MakeNatural i)  = i
```

Следующее объявление экземпляра допускает **Natural** в класс **Num:**

```
instance Num Natural where
fromInteger                  = toNatural
x + y                        = toNatural (fromNatural x + fromNatural y)
x – y                        = let r = fromNatural x - fromNatural y in
                                 if r < 0 then error "Unnatural subtraction"
                                          else toNatural r
x * y                        = toNatural (fromNatural x * fromNatural y)
```

Без этого объявления **Natural** не был бы в **Num.** Экземпляры, объявленные для старого типа, не переносятся на новый. Действительно, все назначение этого типа состоит во введении отличающегося экземпляра **Num.** Это было бы невозможно, если бы **Natural** был определен как синоним типа **Integer.**

Все это работает, используя объявление **data** вместо объявления **newtype.** Однако объявление **data** вытекает из дополнительной служебной информации в представлении значений **Natural.** Использование **newtype** исключает дополнительный уровень преобразований (вызванных ленивостью - laziness), что было бы внесено объявлением **data.**
Смотрите раздел 4.2.3 отчета для большего обсуждения связи между объявлениями **newtype, data** и **type.**

[За исключением ключевого слова объявление **newtype** использует тот же синтаксис, как и объявление **data** с единственным конструктором, содержащим единственное поле. Это подходит, так как типы, определенные с использованием **newtype,** почти идентичны тем типам, которые созданы произвольным определением **data.**]

6.2 Метки Полей

Поля внутри Haskell типа данных могут быть доступны либо позиционно, либо посредством использования имени *метки полей*. Рассмотрим тип данных для двухмерной точки:

```
data Point = Pt Float Float
```

Две компоненты Point являются первым и вторым аргументами для конструктора Pt. Функция, такая как

```
pointx              :: Point -> Float
pointx (Pt x _)     = x
```

может быть использована для обращения к первой компоненте точки более пространно описанным способом, но для больших структур становиться утомительным создание таких функций вручную.

Конструкторы в объявлениях **data** могут объявляться сопроводительными *именами полей*, помещенными в фигурные скобки. Эти имена полей идентифицируют компоненты конструктора лучше по имени, чем по позиции. Альтернативный путь для определения **Point** таков:

```
data Point              = Pt {pointx, pointy :: Float}
```

Этот тип данных идентичен данному ранее определению **Point.** В обоих случаях конструктор **Pt** один и тот же. Однако это объявление также определяет два имени полей, pointx и pointy. Эти имена полей могут быть использованы в качестве *селекторных функций* для выделения компонента из структуры. В этом примере, селекторами являются:

```
pointx                  :: Point -> Float
pointy                  :: Point -> Float
```

Функция, использующая такие селекторы, имеет вид:

```
absPoint                :: Point -> Float
absPoint p              = sqrt (pointx p * pointx p +
                                pointy p * pointy p)
```

Метки полей могут быть использованы для составления новых значений (values). Выражение Pt {pointx=1, pointy=2} идентично Pt 1 2. Использование имен полей в объявлении конструктора данных не препятствует позиционному стилю доступа к полю; как Pt {pointx=1, pointy=2}, так и Pt 1 2 допустимы.

При составлении значения с использованием имен полей, некоторые поля могут быть опущены; эти отсутствующие поля не определяют.

Сопоставление с образцом по именам полей использует подобный синтаксис для конструктора Pt:

```
absPoint (Pt {pointx = x, pointy = y}) = sqrt (x*x + y*y)
```

Функция обновления использует значения полей в существующей структуре для заполнения компонент новой структуры. Если p есть Point, то тогда p {pointx=2} является точкой с таким же самым pointy, как и у p, но с pointx, замененным на 2. Это не разрушительное обновление: функция обновления просто создает новую копию этого объекта, заполняя специальные поля новыми значениями.

[Фигурные скобки в сочетании с метками области отчасти являются специальными: обычно синтаксис Haskell допускает опускание фигурных скобок, используя *двумерный синтаксис* (описанное в Разделе 4.6). Однако, фигурные скобки, сопровождаемые именами полей, должны быть поставлены.]

Имена полей не ограничиваются типами с единственным конструктором (обычно называемые типами ‘запись’). В типе с множеством конструкторов, операции выбора или обновления, использующие имена полей, могут при прогоне программы не иметь успеха. Это подобно поведению функции head при ее применении к пустому списку.

Метки полей разделяют пространство имен верхнего уровня с произвольными переменными и методами классов. Имя поля не может быть использовано в нескольких типах данных в области действия. Однако, в рамках типа данных одно и тоже имя поля может быть использовано в нескольких конструкторах, при условии, что оно имеет то же определение типа во всех случаях. Например, в этом типе данных

```
data T = C1 {f :: Int, g :: Float}
       | C2 {f :: Int, h :: Bool}
```

имя поля f применяется для обоих конструкторов в типе **T**. Таким образом, если **x** есть тип **T**, то x {f=5} будут стремиться к значениям, создаваемым любым из конструкторов в типе **T**.

Имена полей не изменяют базовое свойство алгебраического типа данных; они просто являются условным синтаксисом для доступа компонент структуры данных по имени, предпочитая его позиционному стилю. Они создают конструкторов со многими хорошо управляемыми компонентами, так как поля могут быть добавлены или удалены без изменения любой ссылки на конструктора. За детальной информацией по меткам полей и их семантике обратитесь к Разделу 4.2.1.

6.3 Конструкторы ленивых данных

Структуры данных в Haskell *ленивые*: компоненты не оцениваются до тех пор, пока в них не возникнет необходимость. Допускаются структуры, содержащие элементы, которые, при их оценке, вели бы к ошибке или неуспеху в завершении. Ленивые структуры данных повышают выразительность Haskell и являются существенным аспектом стиля программирования Haskell.

Внутренне, по каждому полю ленивого объекта данных дается заключение в структуре, обычно именуемой *thunk* (“переходник” – часть кода, осуществляющая преобразование, например типов), которая инкапсулирует вычисление, определяющее значение поля. Этот thunk не подключается до тех пор, пока не понадобиться значение (видимо, поля); thunks, которые содержат ошибки (⊥), не оказывают влияния на другие элементы структуры данных. Например, кортеж (‘a’, ⊥) является полностью разрешенным значением в Haskell. Это ‘a’ может быть использовано без нарушения других компонент кортежа. Большинство языков программирования являются строгими, а не ленивыми: то есть, все компоненты структуры данных вычисляются до значений, перед их помещением в структуру.

Существует ряд служебных заметок, связанных с thunks: они требуют времени для конструирования и оценки, они занимают пространство в “куче” (область памяти, выделяемая программой для динамического размещения структур данных), а также вызывают “сборщика мусора” для сохранения других структур, необходимых для оценки thunk. Для избегания этих служебных пометок флаги строгого определения в объявлениях данных предусматривают специальные поля конструктора для того, чтобы сразу быть оцененными, выборочно подавляя ленивость. Поле, маркированное ! в объявлении данных, оценивается при создании структуру вместо той, что задержана в thunk.

Существует ряд ситуаций, для которых свойственно использование флагов строгого определения:

- Компоненты структуры, которые обязательно должны быть оценены в некоторой точке в ходе исполнения программы.
- Компоненты структуры, которые просты для оценки и никогда не вызывают ошибок.

- Типы, в которых частично не определенные значения не являются значимыми.

Например, библиотека комплексных чисел определяет тип **Complex** как:

data RealFloat a => Complex a = !a :+ !a

[обратите внимание на инфиксное определение конструктора : +.] Это определение помечает две компоненты, действительную и мнимую части, комплексного числа, как строго определенные. Это более компактное представление комплексных чисел, но это получается за счет создания комплексных чисел с неопределенным компонентом, 1 : + ⊥ например, полностью неопределенный (⊥). Так как нет реальной необходимости в частично определенных комплексных числах, то имеет смысл использовать флаги строгой определенности для достижения более эффективного представления.

Флаги строгой определенности допускается использовать для адресации утечки информации из памяти: структур, находящихся в "сборщике мусора", в которых нет более необходимости для вычислений.

Флаг строгой определенности, !, может появляться только в объявлениях данных. Он не может быть использован в других типах сигнатур или в любых других определениях. Нет соответствующего способа маркировки аргументов функции, в качестве строго определенных, хотя тот же результат может быть получен, используя функции **seq** и **!$.** За более подробной информацией обратитесь к § 4.2.1.

Сложно представить точные руководящие указания по использованию флагов строгой определенности. Они должны быть использованы с осторожностью: ленивость является одним из фундаментальных свойств Haskell'а и добавление флагов строгой определенности может привести к трудностям в нахождении бесконечных циклов или к другим непредсказуемым последствиям.

7. Ввод/вывод

Система ввода/вывода в Haskell чисто функциональная, кроме того, обладает всей силой выразительности (языка), встречающаяся в традиционных языках программирования. В императивных языках программы приступают к выполнению через *действия*, которые проверяют и модифицируют текущее состояние мира. Типовые действия включают чтение и установку глобальных переменных, запись информации в файлы, чтение входных данных и открытие окон. Такие действия также являются частью Haskell'а, но четко отделены от чисто функционального ядра этого языка.

Уже говорилось, что все операции ввода/вывода построены при помощи такого понятия языка Haskell, как монада. В то же время для понимания системы операций ввода/вывода в Haskell'е нет особой необходимости понимать теоретические основы понятия монада. Монады можно рассматривать как концептуальные рамки, в которых содержится система ввода/вывода. Можно сказать, что понимание теории категорий так же необходимо для использования системы операций ввода/вывода в Haskell'е, как и понимание теории групп для выполнения арифметических операций. Детальное объяснение монад находится в Разделе 9.

Монадические (или одноместные ?) операторы, на которых построена система ввода/вывода, также используется для других целей; более глубоко монады будут

рассмотрены позже. А сейчас будем избегать термина монада и сфокусируем свое внимание на использовании системы ввода/вывода. Лучше всего считать монаду ввода/вывода просто абстрактным типом данных.
Действия предпочтительнее определять, чем вызывать языком выражений Haskell'a. Оценивание определения действия реально не служит причиной для осуществления действия. Скорее вызов действий имеет место вне оценивания выражений, которая была рассмотрена до этого момента.

Действия являются либо элементарными, как определены в примитивах системах, либо последовательной композицией других действий. Монада ввода/вывода содержит примитивы которые выстраивают композитные действия, процесс подобен соединению утверждений в последовательном порядке, используя ';' в других языках. Таким образом, монада служит связующим звеном, которое связывает действия в программе.

7.1 Базовые операции ввода/вывода

Каждое действие ввода/вывода возвращает значение. В системе типов возвращенное значение "помечается' типом **IO,** отличающим действия от других значений. Например, тип функции **get Char**:

```
getChar                    :: IO Char
```

IO Char указывает, что функция getChar, при ее вызове, выполняет некоторое действие, которое возвращает значение типа Char. Действия, которые не возвращают никаких вызывающих интерес значений, используют единичный (unit) тип ( ). Например, функция putChar:

```
putChar                    :: Char -> IO ()
```

принимает знак в качестве аргумента, но не возвращает ничего полезного. Единичный (unit) тип подобен **void** (пустому типу) в других языках.

Действия объединяются в последовательности, используя оператор, имеющий достаточно таинственное имя: >>= (или 'связь'). Как известно, вместо этой функции можно использовать служебное слово do, для того, чтобы спрятать эти операторы, создающие последовательности, под синтаксисом более похожим на традиционные языки. Понятие **do** может быть тривиально расширено до >>=, как описано в §3.14.

Ключевое слово **do** вводит последовательность утверждений, которые исполняются по порядку. Утверждение является либо действием, образ, которого связывается с результатом действия, при использовании <-, либо набором локальных определений, вводимых при использовании **let**. Понятие **do** использует размещение таким же образом, как **let** или **where**, так что мы можем опустить скобки и точки с запятой, используя надлежащие отступы. Вот простая программа для чтения и последующей печати знака:

```
main                       :: IO ()
main                        = do c <- getChar
                                 putChar c
```

Использование имени **main** важно: **main** определяют для того, чтобы быть точкой входа в программы Haskell (подобно функции **main** в C) и должно иметь тип **IO,** обычно **IO ().** **(**Имя **main** является специальным только в модуле **Main;** позже мы должны еще

поговорить о модулях.**)** Эта программа выполняет два действия последовательно: во-первых, она изучает знак, связывая результат с переменной **c**, а затем печатает этот знак.

В отличие от выражения **let**, где переменные находятся в сфере действий всех определений, переменные, определенные <- являются в сфере действий только следующих утверждений.

Есть еще один пропущенный кусок. Можно вызывать действия и проверять их результаты, используя **do**, но как возвратить значение из последовательности действий? Например, рассмотрим функцию **ready**, которая читает знак и возвращает **True**, если знаком был '**y**':

```
ready                   :: IO Bool
ready                   = do c <- getChar
                             c == 'y' -- Bad!!!
```

Так не работает, так как второе утверждение в '**do'** является чисто булевым значением, а не действием. Необходимо взять это булево значение и создать действие, которое ничего не делает, но возвращает это булево значение, как его результат. Функция **return** делает как раз это:

```
return                  :: a -> IO a
```

Функция **return** завершает набор примитивов, создающих последовательность. Последнюю строку **ready** следует читать, как **return (c == 'y').**

Сейчас мы готовы рассмотреть более усложненные функции **IO**. Во-первых, функцию **getline:**

```
getLine     :: IO String
getLine     = do c <- getChar
                 если c == '\n'
                       тогда return ""
                       else do l <- getLine
                               return (c:l)
```

Обратите внимание на второе **do** в пункте **else.** Каждое **do** вводит одиночную цепь утверждений. Любая промежуточная конструкция, например, **if,** должна использовать новое **do** для инициации дальнейших последовательностей действий.

Функция **return** признает обычные значения, например, булевое к области (в базе данных) действий **IO**. А что о другом направлении? Можно ли вызвать несколько действий **IO** в рамках одного обычного выражения? Например, как можно сказать **x + print y** в выражении так, чтобы **y** выдавалось на печать? Ответом является то, что это нельзя! Невозможно действовать украдкой в императивной вселенной посередине чисто функционального кода. Любое значение, 'инфицированное' императивным миром, должно быть помеченным, как таковое. Функция, такая как

```
f       :: Int -> Int -> Int
```

абсолютно не может выполнять какие либо действия по **I/O,** так как **IO** не появляется в возвращенном типе. Этот факт часто весьма огорчителен для программистов, привыкших

к размещению утверждений печати свободно по всему своему коду во время отладки. На деле существует несколько опасных функций, пригодных для обхода этих проблем, но лучше оставить их продвинутым программистам. Пакеты отладки (подобные **Trace**) часто устанавливают свободное использование этих ‘запрещенных функций’ полностью безопасным образом.

7.2 Программирование при помощи действий

Действия ввода/вывода являются обычными значениями в терминах Haskell’a. Т.е. действия можно передавать в функции в качестве параметров, заключать в структуры данных и вообще использовать там, где можно использовать данные языка Haskell. Рассмотрим этот список действий:

```
todoList        :: [IO ()]
todoList        = [putChar 'a',
              do putChar 'b'
                 putChar 'c',
              do c <- getChar
                 putChar c]
```

Этот список не возбуждает никаких действий, он просто содержит их описания. Для того чтобы выполнить эту структуру, т.е. возбудить все ее действия, необходима такая функция как sequence_:

```
sequence_           :: [IO ()] -> IO ()
sequence_ []        = return ()
sequence_ (a:as)    = do   a
                           sequence as
```

Это можно упростить, заметив, что do x;y расширяется до x>>y (смотри Раздел 9.1). Этот образец рекурсии охвачен функцией foldr (смотри определение foldr в Prelude); более лучшим определением sequence_ является:

```
sequence_       :: [IO ()] -> IO ()
sequence_       = foldr (>>) (return ())
```

Понятие do является полезным инструментом, но в этом случае основополагающий монадический оператор, >>, является более подходящим. Понимание операторов, по которым строится do, весьма полезно для программиста, пишущего на Haskell.

Эта функция может быть полезна для написания функции putStr, которая выводит строку на экран:

```
putStr          :: String -> IO ()
putStr s        = sequence_ (map putChar s)
```

На этом примере видно явное отличие системы операций ввода/вывода языка Haskell от систем императивных языков. Если бы в каком-нибудь императивном языке была бы функция map, она бы выполнила кучу действий. Вместо этого в Haskell’е просто создается список действий (одно для каждого символа строки), который потом обрабатывается функцией sequence_ для выполнения.

Prelude и библиотеки содержат много функций, которые полезны в организации последовательностей действий ввода/вывода. Обычно они обобщаются в монады; любая функция с контекстом, включающим Monad m => работает с типом IO.

7.3 Обработка исключений

До сих пор мы избегали вопроса исключений в ходе операций ввода/вывода. Чтобы случилось, если бы функция getChar обнаружила конец файла?[13] В этом случае произойдет ошибка. Как и любой продвинутый язык программирования Haskell предлагает для этих целей механизм обработки исключений. Для этого не используется какой-то специальный синтаксис, но есть специальный тип IOError, который содержит описания всех возникших в процессе ввода/вывода ошибок.
Предикаты допускают, чтобы запрашивались значения IOError. Например, функция:

```
isEOFError :: IOError -> Bool
```

определяет была ли ошибка вызвана условием конца файла. Из-за того, что IOError создается абстрактным, в систему могут добовляться ошибки новых видов без значительного изменения в типе данных. Функция isEOFError определена в отдельной библиотеке, IO, и должна быть импортирована в программу в точном виде.

Обработчик исключений имеет тип (IOError → IO a). Функция catch ассоциирует (связывает) обработчик исключений с действием или набором действий:

```
catch           :: IO a -> (IOError -> IO a) -> IO a
```

Аргументами этой функции являются действие и обработчик исключений. Если действие выполнено успешно, то просто возвращается результат без возбуждения обработчика исключений. Если же в процессе выполнения действия возникла ошибка, то она передается обработчику исключений в качестве операнда типа IOError, после чего выполняется сам обработчик. Например, такова версия getChar возвращающая разделитель строк, когда неожиданно появляется ошибка:

```
getChar'        :: IO Char
getChar'        = getChar `catch` (\e -> return '\n')
```

Это достаточно грубо, т.к. он обрабатывает все ошибки одним и тем же образом. Если только должен быть опознан конец файла, то значение должно быть запрошено:

```
getChar'        :: IO Char
getChar'        = getChar `catch` eofHandler
          where eofHandler e = if isEofError e then return '\n' else ioError e
```

Функция ioError, использованная здесь, выбрасывает исключение следующему обработчику исключений. Типом ioError является:

```
ioError         :: IOError -> IO a
```

Это подобно исключению

```
getChar'        :: IO Char
getChar'        = getChar `catch` eofHandler
           where eofHandler e = if isEofError e then return \'n\ else ioError e

getLine' :: IO String
getLine' = catch getLine'' (\err -> return ("Error: " ++ show err))
           where getLine'' = do c <- getChar'
```

[13] Для ⊥ мы используем термин ошибка; условие, которое не может быть восстановлено, например, из не завершения или не удачи в согласовании с образцом. Исключения, с другой стороны, могут быть отловлены и обработаны внутри монады IO.

```
                    id c == '\n' then return ""
                                 else do l <-  getLine'
                                       return (c : l)
```

В этой программе видно, что можно использовать вложенные друг в друга обработчики ошибок. В функции getChar' отлавливается ошибка, которая возникает при обнаружении символа конца файла. Если ошибка другая, то при помощи функции ioError она отправляется дальше и ловится обработчиком, который «сидит» в функции getLine'. Для определённости в Haskell'е предусмотрен обработчик исключений по умолчанию, который находится на самом верхнем уровне вложенности. Если ошибка не поймана ни одним обработчиком, который написан в программе, то её ловит обработчик по умолчанию, который выводит на экран сообщение об ошибке и останавливает программу.

7.4 Файлы, каналы и обработчики

Для работы с файлами Haskell предоставляет все возможности, что и другие языки программирования. Однако большинство этих возможностей определены в модуле IO, а не в Prelude, поэтому для работы с файлами необходимо явно импортировать модуль IO.
Открытие файла порождает обработчик (он имеет тип Handle). Закрытие обработчика инициирует закрытие соответствующего файла. Обработчики могут быть также ассоциированы с каналами, т.е. портами взаимодействия, которые не связаны напрямую с файлами. В Haskell'е предопределены три таких канала — stdin (стандартный канал ввода), stdout (стандартный канал вывода) и stderr (стандартный канал вывода сообщений об ошибках).

Таким образом, для использования файлов можно пользоваться следующими вещами:

```
type FilePath   = String
openFile              :: FilePath -> IOMode -> IO Handle
hClose          :: Handle -> IO ()
data IOMode           = ReadMode | WriteMode | AppendMode | ReadWriteMode


getChar = hGetChar stdin
```

Haskell также позволяет вернуть все содержимое файла, или канала в виде строки:

```
getContents :: Handle -> IO String
```

Далее приводится пример программы, которая копирует один файл в другой:

```
main = do fromHandle <- getAndOpenFile "Copy from: " ReadMode
          toHandle   <- getAndOpenFile "Copy to: " WriteMode
          contents   <- hGetContents fromHandle
          hPutStr toHandle contents
          hClose toHandle
          putStr "Done."
```

```
getAndOpenFile          :: String -> IOMode -> IO Handle

getAndOpenFile prompt mode =
      do putStr prompt
         name <- getLine
         catch (openFile name mode)
               (\_ -> do putStrLn ("Cannot open "++ name ++ "\n")
                         getAndOpenFile prompt mode)
```

Здесь использована одна интересная и важная функция — hGetContents, которая берёт содержимое переданного ей в качестве аргумента файла и возвращает его в качестве одной длинной строки.

8 Стандартные Классы Haskell

В этом разделе вводятся предписанные в Haskell стандартные типы классов. Мы несколько упростили эти классы, опустив несколько менее интересных методов в этих классах; отчет по Haskell'у содержит более полное описание. Также некоторые из этих стандартных классов являются частью стандартных Библиотек Haskell; они описаны в Отчете Библиотек Haskell.

8 Стандартные Классы Haskell

В этом разделе мы познакомимся с предопределенными в Haskell стандартными классами. Мы будем рассматривать их немного в упрощенном виде, опуская не самые интересные методы в этих классах; Report содержит более полное описание. Так как некоторые из этих стандартных классов являются частью стандартных библиотек Haskell, то они описаны в Library Report.

8.1 Классы Eq и Ord

Мы уже обсуждали классы **Eq** и **Ord**. Определение **Ord** в Prelude является более сложным, чем упрощенная версия **Ord,** описанная выше. В частности, обратите внимание на метод **compare**:

```
data Ordering           = EQ | LT | GT
compare                 :: Ord a => a -> a -> Ordering
```

Метода **compare** достаточно для определения всех других методов (по умолчанию) в этом классе и он является лучшим способом создания экземпляров класса **Ord**.

8.2 Класс Enum

Класс **Enum** содержит набор операций, в основе которых лежит синтаксический сахар (выражения, введенные Питером Ландиным) математических последовательностей; например, арифметическая последовательность **[1,3..]** обозначается **enumFromThen 1 3**. Можно показать, что арифметические последовательности могут использоваться для формирования списков любого типа, являющегося экземпляром класса **Enum.** Это относится не только к большинству числовых типов, но также и к **Char**, так например, ['a'..'z'] обозначает список из строчных букв в алфавитном порядке. Более того, определенные пользователем нумерованные типы, подобные **Color**, могут быть легко объявлены как экземпляр класса **Enum**. Если так, то

[Red .. Violet]) => **[Red, Green, Blue, Indigo, Violet]**

Обратите внимание, что такая последовательность является арифметической в том смысле, что шаг между значениями постоянен, хотя эти значении и не числа. Большинство типов в **Enum** может быть отображено в целые числах фиксированной точности; для таких типов, fromEnum и toEnum осуществляют преобразование между типом Int и типом в Enum.

8.3 Классы Read и Show

Экземпляры класса **Show** являются теми типами, которые могут быть преобразованы в строки (типично для **I/O**). В классе **Read** реализованы операции для синтаксического анализа строк, чтобы можно получить те значения, которые они могут представлять. Простейшей функцией в классе **Show** является **show**:

```
show :: (Show a) => a -> String
```

Достаточно естественно, что **show** принимает любое значение подходящего типа и возвращает его представление в виде строки (списка символов), так например **show (2+2)** даст результат **“4”**. Это все отлично до того момента, пока нам не потребуется создавать более сложные строки, содержащие представления многих значений, например:

```
"The sum of " ++ show x ++ " and " ++ show y ++ " is " ++ show (x+y) ++ "."
```

а потом, все что было сконкатенировано, окажется нужным. Давайте рассмотрим функцию для представления бинарных деревьев из раздела 2.2.1 в виде строки с соответствующими метками для обозначения вложенности деревьев и выделения левой и правой ветвей (при условии, что тип элемента представим в виде строки):

```
showTree :: (Show a) => Tree a -> String
showTree (Leaf x) = show x
showTree (Branch l r) = "<" ++ showTree l ++ "|" ++ showTree r ++ ">"
```

Так как **(++)** имеет сложность, прямопропорциональную длине левого аргумента, то **showTree,** таким образом, может иметь сложность пропорциональную квадрату размера дерева.

Для того, что бы вернуться к линейной сложности, воспользуемся функцией **shows**:

```
shows :: (Show a) => a -> String -> String
```

в **shows** поступает значение, которое можно распечатать и строку, и возвращает эту строку с прикрепленным в начало строковым представлением значения. Второй аргумент используется как некоторый строковый аккумулятор, и теперь **show** мы можем представить как **shows** с пустым аккумулятором. По умолчанию **show** в классе **Show** представляется так:

```
show x = shows x ""
```

Можно использовать **shows** для определения более эффективной версии **showTree,** которая также имеет строковый аккумулятор в качестве аргумента:

```
showsTree                   :: (Show a) => Tree a -> String -> String
showsTree (Leaf x) s        = shows x s
showsTree (Branch l r) s    = '<' : showsTree l ('|' : showsTree r ('>' : s))
```

Это решает нашу проблему эффективности (**showsTree** имеет линейную сложность), но представление этой функции (и других, подобных ей) может быть улучшено. Во-первых, создадим синоним типа:

```
type ShowS                  = String -> String
```

Это тип функции, которая возвращает строчное представление того, за чем следует строковый аккумулятор. Во-вторых, можно избежать перетаскивания аккумулятора с места на место, а также избежать накопления круглых скобок в правом конце длинных конструкций, используя композицию функций:

```
showsTree                   :: (Show a) => Tree a -> ShowS
showsTree (Leaf x)          = shows x
showsTree (Branch l r)      = ('<':) . showsTree l . ('|':) . showsTree r . ('>':)
```

При этой трансформации произошло нечто более важно, чем простое украшательство кода: то, что мы подняли представление с *объектного уровня* (в этом случае, строки) до *функционального уровня*. Мы можем говорить о распечатке, как об отображении дерева в функцию распечатки. Функции, подобные ('<' :) или ("a string" ++), являются примитивными функциями распечатки, а мы построим более сложные функции посредством композиции функции.
Теперь, когда мы можем преобразовывать деревья в строки, давайте обратимся к проблеме инверсии. Основная идея - анализатор для типа **a**, который является функцией, принимающей строку и возвращающей список пар (a, String) [9]. В Prelude есть синоним типа для таких функций:

```
type ReadS a                = String -> [(a,String)]
```

Обычно анализатор возвращает одноэлементный список, содержащий значение типа **a,** прочитанное из входной строки, и оставшуюся часть строки. Если ни один синтаксический разбор не был возможен, результат – пустой список, и если существует более одного возможного синтаксического разбора (однозначность), результирующий список содержит более одной пары. Стандартная функция **reads** является анализатором для любого экземпляра **Read:**

```
reads                       :: (Read a) => ReadS a
```

Мы можем использовать эту функцию для определения анализирующей функции для строчного представления бинарных деревьев, созданных **showsTree**. Вложения списков дают нам удобную идиому для конструирования таких анализаторов[14]:

```
readsTree                   :: (Read a) => ReadS (Tree a)
readsTree ('<':s)           = [(Branch l r, u) | (l, '|':t) <- readsTree s,
                                                 (r, '>':u) <- readsTree t ]
readsTree s                 = [(Leaf x, t) | (x,t) <- reads s]
```

Давайте потратим немного времени на подробное исследование этого определения функции. Существует два основных случая для рассмотрения:

Если первый знак строки, подлежащий разбору **'<'** то мы должны получить представление ветви; в противном случае, мы имеем лист. В первом случае, при вызове остатка входной строки **s**, каждый возможный разбор должен быть деревом **Branch l r** с оставшейся строкой **u,** при допущении следующих условий:

1. Дерево l можно анализировать с начала строки **s**
2. Остаток строки (идущий за представлением 1) начинается с **'|'**. Вызовите хвост этой строки **t.**
3. Дерево **x** можно анализировать, начиная с **t.**
4. Остаток строки с **'>'** и **u,** является хвостом.

Обратите внимание на выразительную силу, которую получаем от комбинации сопоставления с образцом и определителей списков: форма результирующего синтаксического разбора задается основным выражением определителя списка, первые два условия, указанные выше, выражены первым генератором (‘**(l, '|':t)** взят из списка синтаксических разборов **s**’), а оставшиеся условия выражены вторым генератором.

Второе определяющее уравнение, приведенное выше, справедливо говорит о том, что для выполнения синтаксического разбора представления листа, мы выполняем синтаксический разбор типа элемента дерева и применяем конструктор **Leaf** к таким образом полученным значениям.

Допустим, что в классе **Read** (и **Show**) существует экземпляр типа **Integer** (среди множества других типов), при условии, что функция **reads** ведет себя в соответствии с предполагаемым ею действием, например:

```
(reads "5 golden rings")          :: [(Integer,String)])  => [(5, " golden rings")]
```

При таком понимании читателю следует проверить следующие вычисления:

```
readsTree "<1|<2|3>>")      =>    [(Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3)), "")]
readsTree "<1|2" )          =>      []
```

Существует пара изъянов в нашем определении **readsTree.** Первый состоит в том, что анализатор достаточно жесткий, не допускающий никаких пробелов перед и между элементами представления дерева; другой состоит в том, что способ, которым анализируем символы пунктуации, весьма отличен от того способа, что используется при анализе значений листьев и поддеревьев, этот недостаток единообразия ведет к созданию трудного для чтения определения функции. Обе эти проблемы может решить использование лексического анализатора, предоставляемого Prelude:

```
lex                               :: ReadS String
```

lex обычно возвращает одноэлементный список, содержащий пару строк: первую лексему во входной строке и остаток входной строки. Лексическими правилами такие же, как в Haskell, включая комментарии, которые **lex** пропускает, вместе с пробелами. Если входная строка пуста или содержит только пробелы и комментарии, **lex** возвращает [("","")]; если вход не пуст, но также не начинается с действительной лексемы, то **lex** возвращает [].

Используя лексический анализатор, синтаксический анализатор нашего дерева теперь выглядит следующим образом:

```
readsTree                         :: (Read a) => ReadS (Tree a)
readsTree s                       = [(Branch l r, x) | ("<", t) <- lex s,
```

```
                                    (l, u) <- readsTree t,
                                    ("|", v) <- lex u,
                                    (r, w) <- readsTree v,
                                    (">", x) <- lex w          ]
                ++
                [(Leaf x, t) | (x, t) <- reads s                ]
```

Теперь можно использовать **readsTree** и **showTree** для объявления **(Read a) => Tree a** как экземпляра класса **Read** и **(Show a) => Tree a** как экземпляра класса **Show**. Это позволило бы нам использовать перегруженные функции из Prelude для синтаксического анализа (разбора) и отображения деревьев. Кроме того, мы могли бы тогда автоматически анализировать и отображать многие другие типы, содержащие деревья в качестве компонентов, например, **[Tree Integer]**. Как оказывается, **readsTree** и **showsTree** являются практически правильными типами для того, чтобы быть методами **Show** и **Read**. Методы **showsPrec** и **readsPrec** являются параметризированными версиями **shows** и **reads**. Дополнительным параметром является уровень приоритетности, используемый для надлежащего заключения в скобки выражений, содержащих инфиксные конструкторы. Для таких типов, как **Tree**, приоритетность можно игнорировать.

Экземплярами классов **Show** и **Read** для **Tree** являются:

instance **Show a => Show (Tree a)**, где
showsPrec _ x = showsTree x
экземпляр **Read a => Read (Tree a),** где
readsPrec _ s = readsTree s

В качестве альтернативы, экземпляр класса **Show** мог бы быть определен на основе **showsTree:**

```
instance Show a => Show (Tree a) where
    show t = showTree t
```

Однако, это будет менее эффективно, чем версия **ShowS.** Обратите внимание на то, что класс **Show** определяет предопределенные методы как для **showsPrec**, так и для **show**, позволяя пользователю определить какого-то одного из них в объявлении экземпляра. Поскольку эти определения взаимно рекурсивны, то объявление экземпляра, которое не определяет ни одну из этих функций, будет вызывать при обращении зацикливание.
За подробностями о классах **Read** и **Show** отсылаем интересующегося читателя к **§D**.
Можно проверить экземпляры классов **Read** and **Show**, применением (**read . show**) (результат должен быть равен аргументу) к какому-нибудь дереву, где **read** есть конкретизация **reads**:

```
read                    :: (Read a) => String -> a
```

Эта функция не работает, если нет уникального синтаксического анализа (разбора) или если вход не содержит ничего кроме представления одного значения типа **a** (и, возможно, комментарии и пробелы).

8.4 Производные экземпляры

Вспомните экземпляр **Eq** для деревьев, который был представлен в Разделе 5; такое объявление является простым – но нудным – для его создания: необходимо, чтобы тип элемента в листьях был бы типом Eq; тогда два листа эквивалентны, если они содержат

эквивалентные элементы, а две ветви эквивалентны, если их левые и правые поддеревья эквивалентны, соответственно. Любые другие два дерева – неэквивалентны:

```
instance (Eq a) => Eq (Tree a), where
        (Leaf x)      == (Leaf y) = x == y
        (Branch l r)  == (Branch l' r') = l == l' && r == r'
        _             == _              = False
```

К счастью, нет необходимости проходить через это утомительное действие каждый раз, когда необходимы операторы эквивалентности для нового типа; экземпляр **Eq** может быть *выведен автоматически* из объявления данных, если так задано:

```
data Tree a                     = Leaf a | Branch (Tree a) (Tree a) deriving Eq
```

Оператор **deriving** полностью создает объявление экземпляра **Eq**, точно подобное объявлению, приведенному в Разделе 5. Экземпляры классов **Ord, Enum**, **Ix, Read**, и **Show** могут также быть сформированы оператором **deriving**. [Может быть задано несколько имен классов, в любом случае список имен должен заключаться в круглые скобки, а имена отделены запятыми.]

Выведенный экземпляр **Ord** для **Tree** несколько более сложен по сравнению с экземпляром **Eq**:

```
instance (Ord a) => Ord (Tree a) where
    (Leaf _) <= (Branch _) = True
    (Leaf x) <= (Leaf y) = x <= y
    (Branch _) <= (Leaf _) = False
    (Branch l r) <= (Branch l' r') = l == l' && r <= r' || l <= l'
```

Это задает *лексикографический порядок*: Конструкторы располагаются в порядке их появления объявлении данных, а аргументы конструктора сравниваются слева на право. Вспомним, что встроенный тип списка семантически эквивалентен обычному двухконструкторному типу. Фактически, это и есть полное объявление:

```
data [a]        = [] | a : [a] deriving (Eq, Ord)        -- pseudo-code
```

(Списки также имеют экземпляры **Read** и **Show**, которые не являются выведенными.) Выведенные для списков экземпляры **Eq** и **Ord** являются обычными; в частности, строки знаков, в виде списков знаков, упорядочиваются, как определено основополагающим типом **Char**, исходным сравнением подстроки, меньшей в сравнении с более длинной строкой; например, "cat" < "catalog". Практически, экземпляры **Eq** и **Ord** почти всегда выведенные, а не определенные пользователем. На деле следует предоставить собственные определения предикатов равенства и упорядочивания, соблюдая некоторую осторожность для сохранения ожидаемых алгебраических свойств соотношений эквивалентности и общих порядков расстановки. Нетранзитивный предикат (==), например, мог бы быть пагубным, вводя в заблуждение читателей программы и путаного руководства или автоматических преобразований программы, которые надеются на то, что предикат (==) представляет приближение к равенству, заданному определением. Тем не менее, иногда необходимо предоставить экземпляры **Eq** и **Ord,** отличные от тех, которые были бы выведены; вероятно, наиболее важным примером является пример абстрактного типа данных, в котором различные конкретные значения могут представлять те же самые абстрактные значения.

Тип перечислений может иметь выводимый экземпляр **Enum,** и опять при этом упорядочивание такое же, как у конструкторов в объявлении данных. Например:

```
data Day = Sunday | Monday | Tuesday | Wednesday
         | Thursday | Friday | Saturday deriving (Enum)
```

Вот несколько некоторых простых примеров использования выведенных экземпляров для этого типа:

```
[Wednesday .. Friday] ) [Wednesday, Thursday, Friday]
[Monday, Wednesday ..] ) [Monday, Wednesday, Friday]
```

Выведенные экземпляры класса **Read (Show)** возможны для всех типов, чьи типы компонентов также имеют экземпляры **Read (Show).** (Экземпляры **Read** и **Show** для большинства стандартных типов предоставлены в Prelude. Некоторые типы, например тип функции (->), имеют экземпляр **Show,** но не соответствующий **Read.)** Текстуальное представление, определяемое выведенным экземпляром **Show,** находится в соответствии с появлением выражений постоянных Haskell типа, о котором идет речь. Например, если добавляем **Read** и **Show** к оператору **deriving** для вышеприведенного типа **Day**, то получим:

```
show [Monday .. Wednesday]  => "[Monday,Tuesday,Wednesday]"
```

9 Монады

Многие новички в Haskell бывают озадачены понятием монад. Монады довольно часто встречаются в Haskell: система ввода/вывода построена на использовании монад, есть специальный синтаксис для монад (выражения do), стандартные библиотеки содержат целые модули, посвященные монадам. В этом разделе мы рассмотрим программирование с использованием монад более детально.

Возможно, этот раздел наиболее сложный и подробный. Мы обращаемся не только к возможностям Haskell использования монад, но так же постараемся рассмотреть и более общие вопросы: почему монады так важны и как они используются. Нет способа объяснить монады так, что бы поняли абсолютно все; некоторые объяснения можно найти на сайте haskell.org. Можно порекомендовать хорошую книгу по практическому программированию с использованием монад: Wadler, “Monads for Functional Programming” [10].

9.1 Монадические классы

В файле **Prelude** есть несколько классов, описывающих монады, используемые в Haskell. Эти классы основаны на теории категорий. Названия для монадических классов и операций основаны на терминологии теории категорий, но нет необходимости вдаваться глубоко в абстрактную математику, что бы получить интуитивное представление о том, как использовать монадические классы.

Монады являются полиморфными типами, которые представляют собой экземпляры одного из следующих монадических классов: **Functor**, **Monad** и **MonadPlus**.

Монадические классы не наследуются. В модуле **Prelude** определены три монады: **IO**, **[]** и **Maybe**.

Математически монада определяется через набор правил, которые связывают операции, производимые над монадой. Такая идея характерна не только для монад: в Haskell существуют и другие операции, определяемые, по крайней мере неформально, через правила. На пример, **x /= y** и **not (x == y)** должны быть одним и тем же для любых типов сравниваемых значений. Однако, нет никакой уверенности в том, что и == и /= не связаны в классе **Eq**, но и нет ни какой возможности убедиться в том, что == и =/ связаны. Точно так же, монадические правила, о которых мы говорим, не применяются в Haskell явно, но им должны подчиняться любые сущности монадических классов. Монадические правила дают глубокое понимание внутренней структуры монад: рассматривая эти правила, мы надеемся объяснить, как используются монады.

Класс **Functor**, обсуждавшийся в Главе 5, определяет единственную операцию: **fmap**. Функция **map**, применяющая операцию ко всем объектам внутри контейнера (мы можем рассматривать полиморфные типы как контейнеры для значений других типов), возвращает контейнер той же формы. Применим эти правила к **fmap** в классе **Functor**:

```
fmap id       = id
fmap (f . g)  = fmap f . fmap g
```

Эти правила гарантируют, что функция **fmap** не изменит формы контейнера и что порядок элементов контейнера после применения этой операции останется прежним.

Класс **Monad** определяет два основных оператора: **>>=** (связывание) и return.

```
infixl 1 >>, >>=
class Monad m where
    (>>=)    :: m a -> (a -> m b) -> m b
    (>>)     :: m a -> m b -> m b
    return   :: a -> m a
    fail     :: String -> m a
    m >> k   = m >>= \_ -> k
```

Операции связывания, **>>** и **>>=**, комбинируют два монадических значения, тогда как операция return возвращает значение в монаду (контейнер). Сигнатура **>>=** помогает нам понять эту операцию: **ma >>= \v -> mb** комбинирует монадическое значение **ma**, содержащее значения типа **a** и функцию, применяемую к значению **v** типа **a**, возвращая монадическое значение **mb**. Смысл операции – скомбинировать **ma** и **mb** в монадическое значение, содержащее **b**. Функция **>>** используется, когда функции не нужно значение, выдаваемое первым монадическим оператором.

Более точное значение связывания зависит, конечно же, от самой монады. На пример, в монаде ввода/вывода, **x >>= y** выполняет два действия последовательно, передавая результат первого во второе.

Для остальных встроенных монад, списков и **Maybe**, эта монадическая заключена в передаче нуля, или более значений от одного вычисления следующему. Рассмотрим несколько примеров.

Синтаксис **do** предоставляет короткий способ записи цепочек монадических операций. Два основных правила для **do**:

```
do e1 ; e2        =     e1 >> e2
do p <- e1; e2    =     e1 >>= \p -> e2
```

Когда образец во второй строке опровержим, сравнивание с образцом не проходит и вызывается операция **fail**. Это может привести к ошибке (как в случае с монадой ввода/вывода) или ничего не вернуть (как в случае со списком).

Таким образом, в более общем виде **do** выглядит так

```
do p <- e1; e2          =       e1 >>= (\v -> case v of p -> e2; _ -> fail "s")
```

где "**s**" – строка, определяющая положение **do** для того, что бы можно было выдать сообщение об ошибке. Например, в монаде ввода/вывода такое действие, как **'a' <- getChar** не выполнится, если будет набран не символ **'a'**. Это, в свою очередь, завершит программу, так как функция **fail** монады вызовет **error**.

Правила, которым подчиняются **>>=** и **return**:

```
return a >>= k                  =       k a
m >>= return                    =       m
xs >>= return . f               =       fmap f xs
m >>= (\x -> k x >>= h)         =       (m >>= k) >>= h
```

Класс **MonadPlus** используется для монад, имеющих нулевой элемент и операцию ***plus***:

```
class     (Monad m) => MonadPlus m where
      mzero     :: m a
      mplus     :: m a -> m a -> m a
```

Нулевой элемент подчиняется следующим правилам:

```
m >>= \x -> mzero     = mzero
mzero >>= m           = mzero
```

Для списков, нулевое значение - **[]**, пустой список. Монада ввода/вывода не имеет нулевого элемента и не относится к этому классу.

Правила, которым подчиняется оператор **mplus** такие:

```
m `mplus` mzero       = m
mzero `mplus` m       = m
```

Оператор **mplus** является обычной конкатенацией для монады списка.

9.2 Встроенные монады

Что мы можем построить, если нам дать монадические операции и правила, управляющие этими операциями? Мы уже детально обсудили встроенную монаду ввода/вывода, продолжим обсуждать две другие встроенные монады.

Для списков монадическое связывание включает в себя объединение набора вычислений для каждого значения в списке. При использовании со списками, **>>=** становиться:

```
(>>=) :: [a] -> (a -> [b]) -> [b]
```

Это значит, что дан список значений типа a и функция, отображающая значение типа **a** в список тип **b**, связывание применяет эту функцию к каждому значению типа **a** во

входном списке и возвращает конкатенированный список значений типа **b**. Функция **return** создает одноэлементное множество. Эти операции уже должны быть вам знакомы: определители списков могут быть легко выражены с использованием монадических операций, определенных для списков. Следующие три выражения являются разной записью для одного и того же:

```
[(x,y) | x <- [1,2,3] , y <- [1,2,3], x /= y]
```

```
do      x <- [1,2,3]
        y <- [1,2,3]
        True <- return (x /= y)
        return (x,y)
```

```
[1,2,3] >>= (\ x -> [1,2,3] >>= (\y -> return (x/=y) >>=
  (\r -> case r of True   -> return (x,y)
                   _ -> fail "")))
```

Это определение выглядит так, потому что **fail** в данной монаде определен как пустой список. Важно, что каждый **<-** генерирует набор значений, который передается (проходит) в оставшиеся монадические вычисления.

Таким образом, **x <- [1,2,3]** вызовет оставшуюся часть монадических вычислений три раза, по одному разу для каждого элемента списка. Возвращаемое значение, **(x,y)**, будет оценено по всем возможным комбинациям связывания, которые его окружают. Если рассуждать так, то монада списков может рассматриваться как функция многих переменных. Например, эта функция:

```
mvLift2          :: (a -> b -> c) -> [a] -> [b] -> [c]
mvLift2 f x y    = do   x' <- x
                        y' <- y
                        return (f x' y')
```

превращает обычную функцию двух переменных (**f**) в функцию над множественными значениями (списки аргументов), возвращающую значение для каждой возможной комбинации из двух входных аргументов. Например,

```
mvLift2 (+) [1,3] [10,20,30]          ⇒    [11,21,31,13,23,33]
mvLift2 (\a b->[a,b]) "ab" "cd"       ⇒    ["ac","ad","bc","bd"]
mvLift2 (*) [1,2,4] []                ⇒    []
```

Эта функция похожа на функцию **LiftM2** в библиотеке **monad**. Можно подумать, что это перетаскивание функции **f** извне монады списка в эту монаду, в которой вычисления производятся на множественных значениях.

Монада, определенная для **Maybe**, аналогична монаде списков: значение **Nothing** аналогично **[]**, а **Just x** аналогично **[x]**.

9.3 Использование монад

Простое объяснение монадических операторов и их правил, в общем-то, бесполезно для понимания того, зачем вообще нужны монады. Самое главное, для чего нужны монады - модульность. Это достигается определением операции монадически - мы

можем спрятать механизм таким образом, что бы новые элементы можно было включать в монаду легко и прозрачно. В статье Wadler'a [10] можно найти отличные примеры того, как монады могут использоваться для построения модульных программ. Мы начнем с монады, взятой непосредственно из этой статьи, монады состояния, и построим более сложную монаду с аналогичной структурой.

Кратко, монада состояния, построенная из состояния типа **S**, выглядит так:

```
data SM a = SM (S -> (a,S)) -- The monadic type

instance Monad SM where
        -- defines state propagation
        SM c1 >>= fc2       = SM (\s0 -> let (r,s1) = c1 s0
                                             SM c2 = fc2 r in
                                             c2 s1)
        return k            = SM (\s -> (k,s))

        -- extracts the state from the monad
readSM                      :: SM S
readSM                      = SM (\s -> (s,s))

-- updates the state of the monad
updateSM                    :: (S -> S) -> SM () -- alters the state
updateSM f                  = SM (\s -> ((), f s))

-- run a computation in the SM monad
runSM                       :: S -> SM a -> (a,S)
runSM s0 (SM c)             = c s0
```

В этом примере определен новый тип, **SM**, который представляет собой процесс вычисления, явно использующий состояние типа **S**. Далее, процесс вычисления типа **SM t** определяет (выдает результат) значение типа **t** в зависимости от состояния типа **S**. Определение **SM** очень простое: оно состоит из функций, которые получают на вход состояние и возвращают значение (любого типа) и новое состояние. Здесь мы не можем использовать синонимы типа: нам нужно имя типа вроде **SM**, которое может быть использовано в объявлении экземпляра класса. Здесь обычно используется определение **newtype** вместо **data**.

Такое объявление определяет основу монад: композицию двух вычислений и определение пустого (тождественного) вычисления. Упорядочивание (оператор **>>=**) определяет вычисление (заданное с помощью конструктора **SM**), которое передает начальное состояние **s0**, в вычисление **c1**, затем передает результат данного вычисления **r** в функцию, которая возвращает следующее вычисление **c2**. И, наконец, состояние, выработанное вычислением **c1**, передается в вычисление **c2** и результатом композиции является результат **c2**.

Определение **return** легче: **return** не меняет состояния вообще; он нужен только для того, что бы передавать значение в монаду.

Кроме основных упорядочивающих операций **>>=** и **return**, нам необходимы некоторые монадические примитивы. Монадический примитив является попросту операцией, которая используется внутри монады и является рабочим винтиком в системе, которая заставляет монаду работать. Например, в монаде ввода/вывода операторы вроде **putChar** являются примитивами, так как имеют дело только со внутренним устройством монады ввода/вывода.

Аналогично, наша монада состояния использует два примитива: **readSM** и **updateSM**. Они полностью зависят от внутренней структуры монады – изменение определения типа **SM** потребовало бы изменения этих примитивов.

Определение **readSM** и **updateSM** простые: **readSM** выдает состояние монады для просмотра, а **updateSM** позволяет пользователю изменять состояние в монаде. (Мы так же могли использовать в качестве примитива **writeSM**, но более естественно изменять состояние).

И наконец, нам нужна функция, которая запускала бы вычисления внутри монады, **runSM**. Она получает начальное состояние, выполняемое вычисление и выдает результат этого вычисления и выработанное новое состояние. В общем, мы пытаемся определить все вычисления как серию шагов (функций с типом **SM a**), упорядоченных при помощи **>>=** и **return**. Эти шаги могут влиять на состояние (при помощи **readSM** или **updateSM**) или в могут никак с ним не взаимодействовать. Тем не менее, использование (или не использование) состояние спрятано: мы вызываем или упорядочиваем наши вычисления вне зависимости от того, используют они **S** или нет.

Вместо того, что бы приводить здесь пример использования простой монады состояния, мы рассмотрим более сложный пример, включающий монаду состояния. Мы определим маленький вложенный язык ориентированных на ресурсы вычислений. Для этого мы построим специальный язык, реализованный как набор типов и функций Haskell. Такие языки используют стандартные средства Haskell, функции и типы, для построения библиотеки типов и операций, скроенной специально для интересующей нас области.

Полагаем, что в этом примере вычисления требуют ресурсы некоторого типа. Если ресурсы доступны, то вычисления выполняются; когда ресурсы недоступны, вычисления откладываются. Мы используем тип **R**, что бы обозначить вычисления, использующие ресурсы, контролируемые нашей монадой. Определение **R** следующее:

```
data R a = R (Resource -> (Resource, Either a (R a)))
```

Каждое вычисление есть функция из доступных ресурсов в оставшиеся ресурсы, объединенные или с результатом типа **a**, или с отложенным вычислением типа **R a**, продолжающая работать до того момента, когда ресурсы заканчиваются.

Определение **R** как экземпляра класса **Monad** выглядит так:

```
instance Monad R where
R c1 >>= fc2            = R (\r -> case c1 r of
                              (r', Left v)        -> let R c2 = fc2 v in
                                                     c2 r'
                              (r', Right pc1) -> (r', Right (pc1 >>= fc2)))
return v                = R (\r -> (r, (Left v)))
```

Тип **Resource** используется точно так же, как и **state** в монаде состояния. Это определение следует читать так: что бы последовательно выполнить два ‘ресурсоемких’ вычисления **c1** и **fc2** (функция, производящая **c2**), нужно передать начальные ресурсы в **c1**. Возможен один из двух результатов:

- значение **v** и оставшиеся ресурсы, которые используются, что бы определить следующее вычисление (вызов **fc2 v**),
- отложенное вычисление **pc1** и ресурсы, оставшиеся на момент приостановки.

При приостановке нужно принимать во внимание второе вычисление: **pc1** приостанавливает только первое вычисление, **c1**, так что мы должны связать **c2** с **pc1**, что бы приостановить весь процесс вычислений целиком. Определение **return** оставляет ресурсы нетронутыми, помещая **v** в монаду.

Объявление экземпляра определяет основную структуру монады, но определяет, как используются ресурсы. Эта монада могла бы использоваться для контроля многих типов ресурсов или для реализации разнообразных типов использования ресурсов. Покажем на примере очень простое определение ресурсов: мы хотим, что бы **Resource** был типа **Integer** и представлял собой возможное количество вычислений:

```
type Resource           = Integer
```

Эта функция выполняет шаги вычисления до тех пор, пока они не кончатся:

```
step                    :: a -> R a
step v                  = c where
                                c = R (\r -> if r /= 0   then (r-1, Left v)
                                                         else (r, Right c))
```

Конструкторы **Left** и **Right** являются частями типа **Either**. Эта функция продолжает вычисления в **R** , возвращая **v** пока доступен хоть один вычислительный шаг. Если не осталось ни одного шага, функция **step** приостанавливает текущее вычисление (эта приостановка содержится в **c**) и передает это отложенное вычисление обратно в монаду.

Таким образом, у нас есть средства для определения последовательности "ресурсоемких" вычислений (монады) и мы можем выразить форму использования ресурсов, используя **step**. Наконец, нам нужно выяснить как выражаются вычисления в этой монаде.

Рассмотрим инкрементирующую функцию в нашей монаде:

```
inc                     :: R Integer -> R Integer
inc i                   = do    iValue <- i
                                step (iValue+1)
```

Здесь инкремент является одним шагом вычислений. Символ **<-** необходим для того, что бы вытолкнуть значение аргумента из монады; тип **iValue** – **Integer**, а не **R Integer**. Это определение не совсем удовлетворительное по сравнению со стандартным определением функции инкремента. Можем ли мы вместо этого изменить существующие операции, такие как **+**, что бы они заработали в нашем “монадическом мире”? Начнем с функций **lifting**. Эти функции наполняют монаду функциональностью. Рассмотрим определение **lift1** (немного отличается от определения **liftM1**, находящегося в библиотеке **Monad**):

```
lift1                   :: (a -> b) -> (R a -> R b)
lift1 f                 = \ra1 -> do   a1 <- ra1
                                       step (f a1)
```

Здесь берется функция одного аргумента **f** и создается функция в **R**, выполняющая встраиваемую функцию за один шаг. Используя **lift1**, функцию **inc** можно представить так

```
inc                     :: R Integer -> R Integer
inc i                   = lift1 (i+1)
```

Так лучше, но не совсем идеально. Для начала добавим **lift2**:

```
lift2                   :: (a -> b -> c) -> (R a -> R b -> R c)
```

```
lift2 f                 = \ra1 ra2 -> do    a1 <- ra1
                                            a2 <- ra2
                                            step (f a1 a2)
```

Заметьте, что эта функция явно устанавливает порядок вычислений во встраиваемой функции: вычисление, вырабатывающее **a1**, оказывается раньше вычисления для **a2**.
Используя **lift2**, можно создать новую версию **==** в монаде R:

```
(==*)                   :: Ord a => R a -> R a -> R Bool
(==*)                   = lift2 (==)
```

Мы использовали немного другое имя для этой новой функции, так как **==** уже использовано, но в некоторых случаях мы можем использовать одно и тоже имя для встроенных и невстроенных функций. Такое объявление экземпляра позволяет использовать в **R** все операторы из **Num**:

```
instance Num a => Num (R a) where
(+)                     = lift2 (+)
(-)                     = lift2 (-)
negate                  = lift1 negate
(*)                     = lift2 (*)
abs                     = lift1 abs
fromInteger             = return . fromInteger
```

Функция **fromInteger** применяется неявно ко всем целым константам в программе (см. раздел 10.3); это определение позволяет целым константам иметь тип **R Integer**. Теперь мы, наконец-то, можем написать инкремент в том же виде, что и обычно:

```
inc                     :: R Integer -> R Integer
inc x                   = x + 1
```

Заметьте, что мы не можем встроить операторы класса **Eq** так же, как операторы класса **Num**: сигнатура **==*** несовместима с допустимыми перегружениями **==,** так как результатом **==*** будет **R Bool** вместо **Bool**.

Что бы выразить более сложные вычисления в **R**, нам потребуются условный оператор. Так как мы не можем использовать **if** (так как этому оператору необходимо значение типа **Bool**, а не **R Bool**), назовем функцию **ifR**:

```
ifR                     :: R Bool -> R a -> R a -> R a
ifR tst thn els          = do   t <- tst
                                if t then thn else els
```

Теперь мы можем написать полноценную программу в монаде **R**:

```
fact                    :: R Integer -> R Integer
fact x                  = ifR (x ==* 0) 1 (x * fact (x-1))
```

Теперь это уже не самая обычная функция модуля, но все еще читаемая. Смысл введения новых определений для существующих операторов, таких как **+** или **if** является основной частью создания встроенных языков в Haskell. Монады особенно важны для инкапсуляции семантики этих встроенных языков в прозрачном и модульном виде.

Теперь мы, в общем-то, готовы запустить некоторые программы. Эта функция запускает программу в **M**, возвращая максимальное количество потребовавшихся шагов:

```
run                 :: Resource -> R a -> Maybe a
run s (R p)         = case (p s) of
                          (_, Left v) -> Just v
                          _ -> Nothing
```

Мы используем тип **Maybe**, что бы избежать проблем, связанных с тем, что вычисления могут не закончится за выделенное количество шагов. Теперь мы можем вычислять

```
run 10 (fact 2)     ⇒   Just 2
run 10 (fact 20)    ⇒   Nothing
```

Теперь мы можем добавить еще немного функциональности в нашу монаду. Рассмотрим следующую функцию:

```
(|||)               :: R a -> R a -> R a
```

Эта функция запускает два вычисления параллельно, возвращая значение того вычисления, которое раньше закончится. Одно из возможных определений этой функции следующее:

```
c1 ||| c2                   = oneStep c1 (\c1' -> c2 ||| c1')
        where
                oneStep :: R a -> (R a -> R a) -> R a
                oneStep (R c1) f =
                        R (\r -> case c1 1 of
                                        (r', Left v) -> (r+r'-1, Left v)
                                        (r', Right c1') -> -- r' must be 0
                                        let R next = f c1' in
                                                next (r+r'-1))
```

Она выполняет один шаг вычисления **c1**, возвращает значение его значение, или, если оно породило отложенное вычисление (**c1'**), вычисляет выражение **c2 ||| c1'**. Функция **oneStep** принимает один шаг в качестве аргумента, или возвращая вычисленное значение, или передавая остаток вычисления в **f**. Определение функции **oneStep** простое: она передает в **c1** в качестве ресурса **1**. Если итоговое значение достигнуто, то оно и возвращается, при этом изменяется количество оставшихся ресурсов. Если вычисление откладывается, то исправленное количество ресурсов передается в продолжающуюся функцию.

Теперь мы можем вычислять выражения вида run **100 (fact (-1) ||| (fact 3))** без зацикливания, так как два вычисления чередуются . (Определение функции fact зацикливается для -1). Можно построить много различных вариантов на этой общей структуре. К примеру, можно расширить определение состояние так, чтобы подсчитывалось количество шагов. Мы также можем встроить эту монаду в стандартную монаду ввода/вывода **IO**, позволяя вычислениям в **M** взаимодействовать с внешним миром.

Возможно, этот пример более продвинут, чем все остальные в данном введении, он служит для демонстрации силы монад как средства определения основной семантики систем. Мы так же представили этот пример как модель небольшого Языка Определения

Проблемной Области(DSL) – того, что достаточно хорошо реализуется в Haskell. Множество других DSL's было реализовано в Haskell; см. haskell.org для примеров. Большой интерес представляют Fran,− язык реактивной анимации, и Haskore,− язык компьютерной музыки.

10 Числа

В Haskell существует множество числовых типов, основанных на числовых типах языка Scheme [7], основанных, в свою очередь, на Common Lisp [8]. (Эти языки, однако, реализуют динамическую типизацию). Стандартные типы включают целые числа с фиксированной и плавающей точкой, рациональные числа, формируемые из чисел целого типа и действительных чисел с одинарной и двойной точностью и комплексные числа. Здесь мы упомянули об основных характеристиках структур числовых классов, более подробно это описано в §6.4.

10.1 Структура числовых классов

Числовые классы (класс **Num** и те классы, которые являются его потомками) составляют большинство стандартных классов Haskell'а. Мы также отметим, что **Num** является потомком класса **Eq**, но не **Ord**; это из-за того, что для комплексных чисел не определено отношение упорядоченности. Однако потомок класса **Num**, класс **Real,** является также потомком класса **Ord**.

В классе **Num** реализовано несколько основных операций общих для всех числовых типов; они включают среди прочих такие, как сложение, вычитание, отрицание, умножение, и взятие модуля:

```
(+), (-), (*)          :: (Num a) => a -> a -> a
negate, abs            :: (Num a) => a -> a
```

[**negate** – эта функция используется в Haskell'е только в качестве префиксного оператора; мы не можем назвать ее (-), потому что это функция вычитания, так что название **negate** используется вместо этого. Например, **−x*y** эквивалентно **negate(x*y)**. (Префиксный минус имеет тот же самый синтаксический приоритет, что и инфиксный минус, который, конечно, выполняется после умножения.)]

Обратите внимание, что класс **Num** не реализует операцию деления; две различные операции деления реализуются в двух непересекающихся подклассах **Num**:

В классе **Integral** реализуется операция целочисленного деления и операция получения остатка от деления. Стандартный экземпляр класса **Integral** – тип **Integer** (неограниченные целые числа также известные, как "bignums") и тип **Int** (ограниченное, машинное целое число в диапазоне от -2^{31} до $2^{31}-1$). Специфические реализации Haskell'а могут обеспечить другие целочисленные типы в дополнение к этим. Отметим, что **Integral** - это потомок класса **Real**, а не непосредственно класса **Num**; это означает, что не обеспечиваются гауссовы целые числа.

Все другие числовые типы находятся в классе **Fractional**, в котором реализована обычная операция деления (/). Следующий класс **Floating** содержит тригонометрические, логарифмические и экспоненциальные функции.

Для **RealFrac** – потомка класса **Fractional** и **Real** - реализованы функция **properFractional**, декомпозирующая число на его целую и дробную части, и множество функций, округляющих до целых значений по различным правилам:

```
properFraction          :: (Fractional a, Integral b) => a -> (b,a)
truncate, round,
floor, ceiling:         :: (Fractional a, Integral b) => a -> b
```

RealFloat – потомок классов **Floating** и **RealFrac** реализует несколько специальных функций для эффективного доступа к компонентам чисел с плавающей точкой, показателям степени и значащей части числа. Стандартные типы **Float** и **Double** являются экземплярами класса **RealFloat**.

10.2 Построение чисел

Стандартные числовые типы: **Int**, **Integer**, **Float** и **Double** являются примитивными. Другие типы строятся из них с помощью конструкторов типов.

Complex (находящийся в библиотеке **Complex**) является конструктором типов, который создает тип комплексных чисел в классе **Floating** из типа **RealFloat**:

```
data (RealFloat a) => Complex a = !a :+ !a deriving (Eq, Text)
```

Символ ! – это флаг, обозначающий строгий компонент данных; обсуждавшийся в разделе 6.3. Обратите внимание на запись **RealFloat a**, которая ограничивает тип аргумента; таким образом, стандартные комплексные типы - **Complex Float** и **Complex Double**. При объявлении классов также можно увидеть, что комплексное число записывается x :+ y; аргументы являются декартовыми действительной и мнимой частями соответственно. Таким образом, : + является конструктором данных, который можно использовать при сопоставлении с образцом:

```
conjugate           :: (RealFloat a) => Complex a -> Complex a
conjugate (x:+y)    = x :+ (-y)
```

Точно также конструктор типов **Ratio** (находится в библиотеке **Rational**) строит тип рациональных чисел в классе **RealFrac** из экземпляра класса **Integral**. (**Rational** – это синоним **Ratio Integer.**) **Ratio**, однако, является конструктором абстрактного типа данных. Вместо конструктора данных, такого как : +, рациональные числа используют функцию "%", чтобы сформировать дробь из двух целых чисел. Вместо сопоставления с образцом используются функции извлечения компонент:

```
(%)                         :: (Integral a) => a -> a -> Ratio a
numerator, denominator      :: (Integral a) => Ratio a -> a
```

Почему такое различие? Комплексные числа в декартовой форме являются уникальными – не имеется существует нескольких представлений , использующих : +, одного и того же комплексного числа. С другой стороны, рациональные числа не уникальны, но они имеют каноническую (несократимую) форму, выполнение которой должен поддерживать абстрактный тип данных; не всегда, к примеру, что numerator (x%y) равен x, в то время как действительная часть x : + y всегда есть x.

10.3 Приведение числовых типов и перегруженные символы

В стандартном файле Prelude и в библиотеках реализовано несколько перегруженных функций, которые служат для явного приведения типов:

```
fromInteger     :: (Num a) => Integer -> a
fromRational            :: (Fractional a) => Rational -> a
toInteger       :: (Integral a) => a -> Integer
toRational      :: (RealFrac a) => a -> Rational
fromIntegral    :: (Integral a, Num b) => a -> b
fromRealFrac    :: (RealFrac a, Fractional b) => a -> b
fromIntegral     = fromInteger . toInteger
fromRealFrac    = fromRational . toRational
```

Две из них явно используются для обеспечения перегруженных числовых символов: целые нумералы (далее нумералы − именно числовые символы, например, введенные с клавиатуры), фактически эквивалентные применению **fromInteger** к значению, например, типа **Integer**. Точно также, нумералы с плавающей точкой расцениваются как приложение **fromRational** к значению, например, типа **Rational**. Это означает, что 7 имеет тип **(Num a) => a**, а 7.3 имеет тип **(Fractional a) => a**. Это означает, что можно использовать числовые литералы в обычных числовых функциях, например:

```
halve           :: (Fractional a) => a -> a
halve x         = x * 0.5
```

Этот косвенный путь перегрузки нумералов имеет преимущество, заключающееся в том, что метод интерпретации нумералов как чисел данного типа, может быть, конкретизирован при объявления экземпляра **Integral** или **Fractional** (так как **fromInteger** и **fromRational** операторы этих классов соответственно). Например, экземпляр **Num (RealFloat a) => Complex a** содержит этот метод:

```
fromInteger x           = fromInteger x :+ 0
```

Это значит, что экземпляр **Complex fromInteger** определен, для того, чтобы создать комплексное число, реальная часть которого снабжена соответствующим экземпляром **RealFloat fromInteger**. Таким образом, даже для определенных пользователем числовых типов (например, кватернионы) могут пригодиться перегруженные нумералы.

Другой пример, относящийся к первому определению **inc** в разделе 2:

```
inc             :: Integer -> Integer
inc n           = n+1
```

Игнорируя сигнатуру типа, наиболее общий тип функции **inc** – **(Num a) => a- >a**. Явное приписывание типа является верным, однако, таким образом, оно более специфично, чем родительский тип (приписывание более общего типа вызовет статическую ошибку). Приписывание типа имеет эффект своеобразного ограничения типа функции **inc** и в этом случае было бы причиной того, что **inc (1::Float)** вызвало бы ошибку проверки типов.

10.4 Стандартные числовые типы

Рассмотрим определение следующих функций:

```
rms             :: (Floating a) => a -> a -> a
rms x y         = sqrt ((x^2 + y^2) * 0.5)
```

Функция возведения в степень (^) (одна из трех стандартных операторов возведения в степень с различными типами, смотреть §6.8.5) имеет тип **(Num a, Integral b) => a -> b -> a** и так как **2** имеет тип **(Num a) => a**, тип **x^2 - (Num a, Integral b) => a**. Это является проблемой; нет никаких способов определить тип переменной **b**, так как он определяется как из контекста, так и из сигнатуры функции. По существу, программист определил, что x должно быть возведено в степень 2, но не определил, должно ли число **2** быть значением типа **Int** или **Integer**. Конечно, можно это уладить:

```
rms x y         = sqrt ((x ^ (2::Integer) + y ^ (2::Integer)) * 0.5)
```

Однако такая запись очень скоро разрастается в размерах.
Этот вид двусмысленности перегрузки не является специфическим только для чисел:

```
show (read "xyz")
```

В качестве какого типа будет определена эта строка? Этот случай более серьезен, чем двусмысленность при возведении в степень, потому что там можно использовать любой экземпляр класса **Integral**, в то время как здесь можно ожидать различного поведения функции в зависимости от того, какой именно экземпляр **Text** использовал для разрешения двусмысленности.
Из-за различий между числовыми и общими случаями двусмысленности при перегрузки, Haskell обеспечивает решение, которое относится к числам: каждый модуль может содержать определение приоритета экземпляров, состоящее из ключевого слова **default** и заключенного в скобки списка числовых монотипов (типы без переменных) разделенных запятыми. Когда обнаруживается неоднозначная переменная типа (такая как **b**, приведенная в тексте выше), то если, по крайней мере, один возможных классов числовой и все эти классы стандартные, то просматривается список приоритетов и первый тип из списка, который удовлетворит контексту переменной типа, используется. Например, если использовалась декларация **default (Int, Float)**, то приведенный выше неоднозначный образец будет иметь тип **Int**. (Более подробно это описано в §4.3.4 .)
Стандартным приоритетом экземпляров является **(Integer, Double)**, но **(Integer, Rational, Double)** может также быть таковым. Очень осторожные программисты могут предпочесть описание **default()** , которое обеспечивает безошибочность.

11 Модули

Программы на Haskell’е состоит из множества модулей. Модули в Haskell’е служат двойной цели – управление пространствами имен и создание абстрактных типов данных.

Высший уровень модуля содержит любую из тех деклараций, которые были обсуждены: декларации типов и данных, декларации классов и экземпляров, приписывание типов, определения функций и связь с образцом. Если бы не тот факт, что декларация импорта должна появиться первой, то декларации могли бы появляться в любом порядке.

Структура Haskell’овских модулей относительно консервативна: пространство имен полностью плоское и модули ни в каком случае не могут быть «первичными

классами». Имена модулей состоят из букв и цифр и должны начинаться с прописной буквы. Не имеется формальной связи между Haskell'овскими модулями и файловой системой, которая поддерживала бы их. В частности, не имеется никакой связи между именами модулей и именами файлов, и более чем один модуль может находиться в одном файле (один модуль может быть даже в нескольких файлах). Конечно, скорее всего, при реализации будет принято соглашение делать связь между модулями и файлами более строгой.

Технически говоря, модуль – это действительно только одна большая декларация, которая начинается с ключевого слова **module**; приведем пример модуля, который называется **Tree:**

```
module Tree ( Tree(Leaf,Branch), fringe ) where
data Tree a                 =Leaf a | Branch (Tree a) (Tree a)
fringe :: Tree a -> [a]
fringe (Leaf x)             = [x]
fringe (Branch left right)  = fringe left ++ fringe right
```

Тип **Tree** и функция **fringe** должны быть знакомы; они были даны в качестве примеров в разделе 2.2.1.

Этот модуль явно экспортирует **Tree, Leaf, Branch** и функцию **fringe**. Если список экспорта, следующий за ключевым словом **module**, опущен, то все названия, находящиеся на верхнем уровне модуля экспортировались бы. (В приведенном выше примере все явно экспортируется, так что результат будет тот же самый.) Отметим, что название типа и его конструктора, сгруппированы вместе, как в **Tree(Leaf, Branch)**. С помощью короткой записи, можно также записать **Tree(..)**. Экспортирование подмножества конструкторов также возможно. Имена в списке экспорта не должны быть локальными в модуле экспорта; любое название по возможности может быть внесено в список экспорта.

Модуль **Tree** может теперь быть импортирован в некоторый другой модуль:

```
module Main (main) where
import Tree ( Tree(Leaf,Branch), fringe )

main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

Различные элементы, импортируемые в и экспортируемые из модуля называются объектами. Отметим, что явный список импорта находится в декларации импорта; в случае его отсутствия будут импортированы все объекты, экспортируемые модулем **Tree**.

11.1 Классификационные имена

Имеется очевидная проблема с импортированием имен непосредственно в пространство имен модуля. Что если два импортируемых модуля содержат различные объекты с одинаковыми именами? Haskell решает эту проблему, используя *классификационные имена*. Декларация импорта может использовать ключевое слово **qualified** для того, чтобы импортируемые имена сопровождались именами модуля, экспортирующего их. За этими префиксами (приставками) следует символ ' . ', без пробелов. [Классификаторы – это часть лексического синтаксиса. Таким образом, **A.x** и **A . x** весьма различны: первое классификационное имя, а второе использует инфиксную функцию '.'.] Пример использования модуля **Tree**, приведенного выше:

```
module Fringe(fringe) where
import Tree(Tree(..))
fringe :: Tree a -> [a]                -- A different definition of fringe
fringe (Leaf x) = [x]
fringe (Branch x y) = fringe x
module Main where
import Tree ( Tree(Leaf,Branch), fringe )
import qualified Fringe ( fringe )
main = do print (fringe (Branch (Leaf 1) (Leaf 2)))
          print (Fringe.fringe (Branch (Leaf 1) (Leaf 2)))
```

Некоторые программисты на Haskell'е предпочитают использовать классификаторы для всех импортируемых объектов, делая источник из каждого имени объекта при каждом его использовании. Другие предпочитают короткие имена и используют классификаторы, когда это абсолютно необходимо. Классификаторы используются для решения конфликтов между различными объектами, которые имеют одинаковые имена. Но что если один и тот же объект импортируется в более чем один модуль? К счастью, такое совпадение имен разрешено: объект может быть импортирован по различным маршрутам без конфликтов. Компилятор знает, действительно ли объекты из разных модулей одинаковые.

11.2 Абстрактные типы данных

Кроме управления пространствами имен, модули обеспечивают единственный способ построения абстрактных типов данных (ADTs) в Haskell'е. Например, характерная особенность ADT – это то, что представляемый тип скрыт; все операции с ADT выполняются на абстрактном уровне, который не зависит от представления. Например, хотя тип **Tree** является достаточно простым, чтобы мы не могли сделать его абстрактным, подходящий ADT включает следующие операции:

```
data Tree a          -- just the type name
leaf                 :: a -> Tree a
branch               :: Tree a -> Tree a -> Tree a
cell                 :: Tree a -> a
left, right          :: Tree a -> Tree a
isLeaf               :: Tree a -> Bool
```

Поддерживающий это модуль:

```
module TreeADT (Tree, leaf, branch, cell,
                left, right, isLeaf) where
data Tree a          = Leaf a | Branch (Tree a) (Tree a)
leaf                 = Leaf
branch               = Branch
cell (Leaf a)        = a
left (Branch l r)    = l
right (Branch l r)   = r
isLeaf (Leaf _)      = True
isLeaf _             = False
```

Заметим, что в списке экспорта, тип с названием **Tree** появляется один (то есть без его конструкторов). Таким образом, **Leaf** и **Branch** не экспортируются, и единственный путь построить или разбить (разделить) деревья вне модуля – это использование различных (абстрактных) операций. Конечно, преимуществом этого сокрытия информации является то, что далее можно изменять представляемые типы без воздействий пользователя на тип.

11.3 Дополнительные вопросы

Здесь приведен краткий обзор некоторых других аспектов системы модулей. Более подробно это описано в Report.

- Декларация **import** может выборочно скрывать объекты, использующие пункт **hiding** в декларации импорта. Это полезно для явного исключения имен, которые используются в других местах без необходимости использования классификаторов при импорте этих имен из других модулей.

- **import** может содержать пункт **as**, для определения классификаторов, отличных от имен экспортирующих модулей. Это может быть использовано для сокращения классификаторов модулей с длинными именами или для легкого приспособления к изменениям в имени модуля без изменения всех классификаторов.

- Программы неявно импортируют модуль Prelude. Явный импорт Prelude переопределяет все имена Prelude. Таким образом,

  import Prelude hiding length

  не будет импортировать **length** из Standard Prelude, позволяя имя **length** определить по-другому.

- Объявление экземпляров явно не указывается в списке экспорта или импорта. Каждый модуль экспортирует все свои объявления экземпляров, и каждый модуль импортирует все экспортируемые экземпляры.

- Методы класса могут быть названы в той же манере, как и конструкторы данных, заключены в круглые скобки или как обычные переменные.

Хотя Haskell'овская система модулей относительно консервативна, но все же имеет много правил относительно импортируемых и экспортируемых значений. Большинство из них очевидно – например, это недопустимый импорт двух различных объектов, имеющих одинаковые имена в одной и той же области видимости. Другие правила не так очевидны – например, данные тип и класс не могут иметь более чем одно объявление **instance** для этой комбинации типа и класса где-нибудь в программе. Читатель должен прочитать Report для более подробного описания.

12 Ловушки при типизации

Данный короткий раздел дает интуитивное описание некоторых общих проблем, которые могут возникнуть у новичков при работе с системой типов Haskell'а.

12.1 Let-Связанный полиморфизм

Любой язык, использующий систему типов Хиндли-Милнера, имеет так называемый *let-связанный полиморфизм*, так как идентификаторы, не связанные конструкциями **let** или **where** (или находящиеся на верхнем уровне модуля), ограничены относительно своего полиморфизма. В частности, *связанное лямбда-выражение* (т.е., функция, передающаяся в качестве аргумента другой функции) не может быть представлена двумя разными способами. К примеру, следующая программа является неверной:

```
let f g = (g [], g 'a')              -- неверный тип выражения
in f (\x -> x)
```

потому что лямбда-абстракция, чей тип имеет вид *a -> a*, подставляемая вместо **g**, используется внутри функции **f** двумя разными способами: с типом *[a] -> [a]*, и с типом *Char -> Char*.

12.2 Перегрузка нумералов

Временами легко забыть, что нумералы *перегружены,* и *не настолько безоговорочно подчиняются* многочисленным числовым типам, как во многих других языках. Более общие числовые выражения иногда не могут быть достаточно обобщающими. Достаточно распространенная ошибка при указании типа числового выражения выглядит примерно так:

```
average xs = sum xs / length xs        -- неверно!
```

Операция *(/)* требует дробных аргументов, но результат функции **length** имеет тип **Int**. Несоответствие типов должно быть исправлено явным указанием типа:

```
average :: (Fractional a) => [a] -> a
average xs = sum xs / fromIntegral (length xs)
```

12.3 Ограничение мономорфизма

Система типов Haskell'а имеет ограничение, связанное с классами типов, которые нельзя найти в обыкновенных системах типов Хиндли-Миллера: *ограничение мономорфизма*. Причина данного ограничения связана с едва уловимой неопределенностью типов и в полной мере объясняется в Report (§4.5.5).

Согласно ограничению мономорфизма, любой идентификатор, ограниченный связыванием образцов (что включает и связывание отдельных идентификаторов), и не имеющий явного указания типа, должен быть *мономорфным*. Идентификатор является мономорфным, если либо не перегружен, или же перегружен, но при этом используется не более чем в одной отдельной перегрузке и не экспортируется.

Результатом нарушения этого ограничения является статическая ошибка типа. Простейший способ избежать проблему – явное присвоение типа. Хотя *не любое* приписывание типа подойдет (оно должно быть верным).

Распространенное нарушение данного ограничения происходит в случае с функциями, определенными на высшем уровне, как в случае определения функции **sum** из Standard Prelude:

```
sum = fold1 (+) 0
```

Такое описание приведет к статической ошибке типа. Ошибку можно исправить добавлением приписания типа:

```
sum :: (Num a) => [a] -> a
```

Заметьте так же, что данная проблема не возникла, если бы мы написали:

```
sum xs = fold1 (+) 0 xs
```

так как ограничение применяется только к связыванию образцов.

13 Массивы

В идеале, массивы в функциональном языке рассматривались бы просто как функции, которые преобразуют индексы в значения. Однако на практике с тем, чтобы обеспечить эффективный доступ к элементам массива мы должны убедиться в том, что мы можем воспользоваться преимуществом особенных свойств областей определения этих функций, которые являются изоморфными по отношению к конечным смежным подмножествам целых чисел. Поэтому Haskell рассматривает массивы не как обычные функции с операцией аппликации, а как абстрактный тип данных с операцией над индексами.

Различают два основных подхода к построению функциональных массивов: *возрастающее* и *единое* определение. В случае возрастающего определения, мы имеем одну функцию, которая строит пустой массив заданного размера и другую функцию, которая принимает массив, индекс и значение и выдает новый массив, отличающийся от входного только в заданном индексе. Очевидно, что наивная реализация такой семантики массива будет недопустимо неэффективной, либо требуя новую копию массива для каждого возрастающего переопределения или занимая линейное время на поиск элемента в массиве. Таким образом, при более тщательной попытке использования данного подхода применяют сложный статический анализ и искусные механизмы времени выполнения для избежания чрезмерного копирования. С другой стороны, единый подход строит весь массив одновременно, без обращения к промежуточным значениям массива. Хотя в Haskell’е и есть оператор возрастающего построения массивов, главным образом используются средства единого подхода.

Массивы не являются частью Standard Prelude – стандартная библиотека содержит операторы над массивами. Любой модуль, работающий с массивами, должен импортировать модуль **Array**.

13.1 Индексные типы

Библиотека **Ix** определяет класс типа индексов массива:

```
class (Ord a) => Ix a where
        range :: (a, a) -> [a]
        index :: (a, a) a -> Int
```

inRange :: (a, a) -> a -> Bool

Определения экземпляров предусмотрены для типов **Int**, **Integer**, **Char**, **Bool**, а так же для кортежей типов **Ix** длиной не более 5. В дополнение, экземпляры могут быть автоматически получены для перечислимых типов и типов кортежей. Примитивные типы понимаются как векторы индексов, а кортежи – как индексы многомерных декартовых массивов. Заметьте, что первым аргументом всех операций класса **Ix** является пара индексов; обычно они являются *границами* (первый и последний индекс) массива. К примеру, границами вектора из 10 элементов, индексы типа **Int** которого начинаются с нуля, будет **(0, 9)**. В то же время матрица размером 100 на 100 с началом координат в единице будет иметь границы **((1, 1), (100, 100))**. (Во многих других языках данные границы будут записаны в виде **1: 100, 1:100**, однако настоящая форма лучше вписывается в систему типов, так как каждая границы имеет тот же тип, что и обычный индекс.)

Операция **range** принимает пару, состоящую из значений границ, и строит список индексов, лежащих между этими границами, в порядке возрастания. К примеру,

range (0, 4) => [0, 1, 2, 3, 4]
range ((0, 0), (1,2)) => [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]

Предикат **inRange** определяет, лежит ли индекс между заданной парой границ. (Для кортежей данная проверка производится покомпонентно.) Наконец, операция **index** позволяет обратиться к определенному элементу массива: принимая в качестве входных значений границы и входящий в них индекс, операция выдает порядковый номер индекса в данном интервале, считая с нуля; к примеру:

index (1, 9) 2 => 1
index ((0, 0), (1, 2)) (1, 1) => 4

13.2 Создание массива

Функция единого создания массива в Haskell'е формирует массив из пары границ и списка пар вида (индекс, значение) (*ассоциативный список*):

array :: (Ix a) => (a, a) -> [(a, b)] -> Array a b

Далее, к примеру, приводится определение массива квадратов чисел от 1 до 100:

squares = array (1, 100) [(i, i*i) | i <- [1 .. 100]]

Использование определителей списков в ассоциативных списках при записи выражений для определения массивов является распространенным; кстати, такая запись приводит к определениям массивов, больше похожим на *определители массивов* языка Id [6].

Индексация массивов производится с помощью инфиксного оператора **!**, а границы массива могут быть получены с помощью функции **bounds**:

squares ! 7 => 49
bounds squares => (1, 100)

Данный пример можно обобщить, параметризуя границы и применяя функцию к каждому индексу:

```
mkArray :: (Ix a) => (a -> b) -> (a, a) -> Array a b
mkArray f bnds = array bnds [(i, f i) / i <- range bnds]
```

Таким образом, мы можем определить **squares** как **mkArray (\i -> i*i) (1, 100)**.

Многие массивы определяются рекурсивно, т.е. в случае, когда некоторые элементы зависят от значений других элементов. Далее, к примеру, приводится функция, возвращающая массив чисел Фибоначчи:

```
fibs :: Int -> Array Int Int
fibs n = a where a = array (0, n) ([(0, 1), (1, 1)] ++
                                   [(i, a! (i-2) + a! (i-1)) | i <-[2 .. n]])
```

Другой пример такой рекуррентности – матрица *фронта волны* размера *n* на *n*, элементы первой строки и первого столбца которой равны *1*, а остальные элементы равны сумме своих соседей по направлению запада, северо-запада и севера:

```
wavefront :: Int -> Array (Int, Int) Int
wavrfront n = a where
                a = array ((1, 1), (n, n))
                    ([ ((1, j), 1) | j <- [1 .. n] ] ++
                     [ ((i, 1), 1) | i <- [2 .. n] ]  ++
                     [ ( (i, j), a! (i, j-1) + a! (i-1, j-1) + a! (i-1, j ) )
                                   | i <-  [2 .. n], j <- [2 .. n] ] )
```

Матрица фронта волны названа так потому, что при параллельной реализации рекуррентность указывает на то, что вычисления могут начинаться с первой строки и столбца в параллели и продолжаться как клинообразная волна, движущаяся из северо-запада на юго-восток. Однако необходимо отметить, что в ассоциативном списке никак не определен порядок вычислений.

Во всех приведенных нами на данный момент примерах, для каждого индекса массива было задана однозначная связь, и все эти индексы лежали в пределах границ массива. Действительно, это необходимо делать с тем, чтобы массив был полностью определен. Результатом определение связи с индексом, лежащим вне границ массива, будет ошибка; если индекс отсутствует или появляется чаще, чем один раз, не последует немедленной ошибки, но значение массива в данной ячейке будет не определено и таким образом попытка индексации массива приведет к ошибке.

13.3 Накопление

Мы можем ослабить ограничение о появлении индекса в ассоциативном списке не более одного раза, определив способ объединения различных значений, связанных с единственным индексом; результат – так называемый *накопительный массив*:

```
acumArray :: (Ix a) -> (b -> c -> b) -> b -> (a, a) -> [Assoc a c] -> Array a b
```

Первый аргумент **acumArray** – *накапливающая функция*, второй – начальное значение (одинаковое для всех элементов массива), а оставшиеся аргументы – границы и ассоциативный список, как и в функции **array**. Обычно накапливающей функцией является **(+)**, а начальным значением – ноль; к примеру, такая функция принимает границы и список значений (типа индексов) и выдает гистограмму, т.е. таблицу с количеством попаданий каждого значения в пределах границ:

```
hist :: (Ix a, Integral b) => (a, a) -> [a] -> Array a b
hist bnds is = accumArray (+) 0 bnds [(i, 1) | is, inRange bnds i]
```

Предположим, у нас имеется набор измерений на интервале *[a, b)*, и нам необходимо разделить интервал на десятки и сосчитать число измерений в каждом из них:

```
decades :: (RealFrac a) => a -> a -> [a] -> Array Int Int
decades a b = hist (0, 9) . map decade
                where decade x = floor ((x-a) * s)
                      s = 10 / (b-a)
```

13.4 Возрастающее построение

В дополнение к функциям единого создания массивов, в Haskell'е так же имеется функция возрастающего построения массивов, обозначающаяся как инфиксный оператор //; простейший случай, когда в массиве **a** элемент **i** меняется на **v**, запишется как **a // [(i, v)]**. Причина использования квадратных скобок состоит в том, что левым аргументом оператора **(//)** является ассоциативный список, обычно содержащий требуемое подмножество индексов массива:

```
(//) :: (Ix a) => Array a b -> [(a, b)] -> Array a b
```

Как и в случае функции **array**, индексы в ассоциативном списке не должны повторяться для значений, которые будут определены. К примеру, ниже приводится функция обмена двух строк матрицы:

```
swapRows :: (Ix a, Ix b, Enum b) => a -> a -> Array (a, b) c -> Array (a, b) c
swapRows i i' a = a // ( [ ((i, j), a! (i', j)) | j <- [jLo .. jHi] ]  ++
                         [ ((i', j), a! (i, j))  | j <- [jLo .. jHi] ] )
                  where ((iLo, jLo), (iHi, jHi)) = bounds a
```

Конкатенация двух различных определителей списков, определенных на одном и том же списке индексов **j** в данном примере является немного неэффективной; это похоже на написание двух циклов, в то время как в процедурном языке хватит и одного. В Haskell'е мы так же можем произвести оптимизацию слиянием циклов:

```
swapRows i i' a = a // [assoc | j -< [jLo .. jHi],
                                assoc <- [((i, j), a! (i', j)),
                                          ((i', j), a! (i, j))] ]
                  where ((iLo, jLo), (iHi, jHi)) = bounds a
```

13.5 Пример: перемножение матриц

Мы завершим наше знакомство с массивами в Haskell'е известным примером перемножения матриц, используя преимущества перегрузки для определения очень общей функции. Так как в данном случае привлекается только перемножение и сложение типов элементов матриц, мы получаем функцию, которая перемножает матрицы любых числовых типов. Дополнительно, если мы будем осторожны и используем только **(!)** и операции класса **Ix**, которые применяются к индексам, мы получим универсальность и по классам индексов и, кстати, все четыре типа индексов строк и столбцов необязательно все должны быть одинаковыми. Однако для простоты, мы требуем, чтобы левые индексы

столбцов и правые индексы строк имели один и тот же тип и, более того, чтобы их границы совпадали:

```
matMult :: (Ix a, Ix b, Ix c, Num d) =>
           Array (a, b) d -> Array (b, c) d -> Array (a, c) d
matMult x y = array resultBounds
                   [((i, j), sum [x! (i, k) * y! (k, j) | k <- range (lj, uj)])
                             | i <- range (li, ui),
                               j <- range (lj', uj') ]
      where ( (li, lj), (ui, uj) ) = bounds x
            ( (li', lj'), (ui', uj') ) = bounds y
            resultBounds
              | (lj, uj) == (li', ui') = ((li, lj'), (ui, uj'))
              | otherwise              = error "matMult: несовместимые границы"
```

Мы так же можем определить **matMult**, используя **accumArray**. Результатом будет алгоритм, который имеет больше сходства с обычным процедурным языком:

```
matMult x y = accumArray (+) 0 resultBounds
                         [ ((i, j), x! (i, k) * y! (k, j))
                                 | i <- range (li, ui),
                                   j <- range (lj', uj'),
                                   k <- range (lj, uj)  ]
      where ( (li, lj), (ui, uj) ) = bounds x
            ( (li', lj'), (ui', uj') ) = bounds y
            resultBounds
              | (lj, uj) == (li', ui') = ((li, lj'), (ui, uj'))
              | otherwise              = error "matMult: несовместимые границы"
```

Мы можем произвести дальнейшее обобщение, делая функцию функцией более высокого порядка, просто заменив **sum** и **(*)** на параметры функции:

```
genMatMult :: (Ix a, Ix b, Ix c) =>
              ([f] -> g) -> (d -> e -> f) ->
          Array (a, b) d -> Array (b, c) e -> Array (a, c) g
genMatMult sum' star x y = array resultBounds
                   [((i, j), sum' [x! (i, k) `star` y! (k, j) | k <- range (lj, uj)])
                             | i <- range (li, ui),
                               j <- range (lj', uj') ]
      where ( (li, lj), (ui, uj) ) = bounds x
            ( (li', lj'), (ui', uj') ) = bounds y
            resultBounds
              | (lj, uj) == (li', ui') = ((li, lj'), (ui, uj'))
              | otherwise              = error "matMult: несовместимые границы"
```

Поклонники APL оценят полезность таких функций, как следующие:

```
genMatMult maximum (-)
genMatMult and (==)
```

В первом случае аргументами являются числовые матрицы, а *(i, j)*-м элементом результата будет максимум разницы между соответствующими элементами *i*-й строки и *j*-го столбца входных матриц. Во втором случае, аргументами являются матрицы любых

эквивалентных типов, а результатом будет Булева матрица, в которой *(i, j)*-й элемент равен истине (**True**) тогда и только тогда, когда *i*-я строка первого аргумента равна *j*-му столбцу второго.

Заметьте, что требование об одинаковости типов элементов в **genMatMult** необязательно, однако целесообразно для параметра функции **star**. Мы можем продолжить обобщение, опуская требование о том, что типы индексов первого столбца и второй строки должны совпадать; несомненно, две матрицы две матрицы будут совместимыми, если длина столбцов первой совпадает с длиной строк второй. Читатель, возможно, захочет реализовать данную более обобщенную версию. (**Подсказка:** Используйте операцию **index** для определения длин.)

14 Следующий уровень

В Интернете на сайте **haskell.org** представлен огромный набор ресурсов по Haskell'у. Здесь вы найдете компиляторы, демонстрационные примеры, статьи и множество ценной информации о Haskell'е и функциональном программировании. Компиляторы и интерпретаторы Haskell'а работают практически на всем оборудовании и операционных системах. Система Hugs является одновременно маленькой и портативной – это отличное средство для изучения Haskell'а.