Введение в Haskell 98
(черновой перевод)

Paul Hudak John Peterson Joseph H.Fasel*

11 сентября 2006 г.

Содержание

*Translation from english by Anthony Akentiev (Перевод с английского выполнен Антоном Акентьевым)

1 Предисловие переводчика

Данный текст является лишь наброском (иными словами - черновиком) перевода. Прошу не судить меня слишком строго, если вам встретятся не устоявшиеся термины, а мои собственные (например, "гарды"). Прошу все пожелания, а также исправления посылать на специально открытый для этих целей почтовый ящик haskell-trans@yandex.ru. Приветствуется любая помощь, имена помогавших людей будут перечислены в финальном выпуске перевода.

2 Введение

Целью данного руководства является вовсе не обучение программированию и тем более не обучение функциональному программированию. Настоящее руководство служит как дополнение к Haskell Report [1], который сложно назвать компактным. Считаем своей задачей „открытие двери“ в Haskell для человека, который работал до этого, по крайней мере, с одним языком программирования (желательно функциональным или „полуфункциональным“, вроде ML или Scheme). Если читатель желает глубже изучить функциональное программирование, очень советуем ему текст Бёрда (Bird) под названием „Введение в функциональное программирование“ (*Introduction to Functional Programming*) [1] или „Введение в функциональное программирование на Haskell “ (*An introduction to Functional Programming Systems Using Haskell*)[2], написанное Дэйви (Davie). Ещё одним полезным источником в изучении принципов, на которых основан Haskell , может служить [3].

Язык Haskell претерпел существенные изменения с момента его первого появления в 1987 году. Настоящая статья охватывает версию 1998 года. Ранние версии языка теперь считаются устаревшими (obsolete); теперь любые пользователи Haskell вынуждены иметь дело лишь с Haskell 98. Существует множество расширений этого языка, которые достаточно хорошо распространены. Они не являются темой данной статьи.

В данном руководстве мы старались следовать нашему главному принципу: необходимо мотивировать идею, привести примеры, а уже позже привести ссылку на Haskell Report . Тем

[1] Здесь и далее будем приводить название данного документа в изначальной форме - Прим.пер.

не менее, мы рекомендуем читателю не вникать в детали до тех пор, пока он не дочитает данное творение до конца. Стандартная *прелюдия*[2] Haskell (см. Дополнение A Haskell Report и „Library Report“) содержит огромное количество полезных примеров. После прочтения настоящего руководства можно переходить к вдумчивому чтению исходного текста этого модуля. Это не только поможет читателю разобраться, что же предетавляет из себя настоящий код Haskell , но и поможет в изучении встроенных (заранее написанных) функций и типов.

Напоследок не забывайте про веб-сайт Haskell - **http://haskell.org**, который содержит множество информации о языке и его различных реализациях.

[Конечно же, мы не хотели перегружать текст излишне строгими синтаксическими правилами, а скорее пожелали вводить их постепенно по мере необходимости. Обратите внимание, что мы будем заключать их в такие квадратные скобки, как этот абзац. Поэтому данное руководство сильно отличается от Haskell Report , но именно последний является полноценным источником деталей, так что указатели (например, „§ 2.1“) будут указывать на главы Haskell Report .]

Haskell является строго типизированным языком[3]: на типах основано абсолютно всё, но часто новички пугаются обилию возможностей. Для тех, кто имел дело лишь с „нетипизированными“ языками вроде Perl,Tcl,Scheme, это покажется вдвойне сложным. Для тех же, кто работал с Java,C,Modula,ML, это не будет чем-то особенно новым, но в любом случае придётся многому учиться, так как система типов Haskell чуть другая и, как правило, выглядит богаче. В конце концов, „строгая типизация“ Haskell является той неизбежностью, которой не избежать.

3 Значения,Типы и многое другое

Так как Haskell является т.н. „чистым“ функциональным языком программирования, все вычисления организованы с помощью *выражений* (синтаксические термы), которые вычисляют *значения* (абстрактные сущности, очень похожие на простые ответы). Каждое значение имеет определённый *тип*. (Можно считать, что типы являются множествами, содержащими значения). Примерами выражений являются атомарные величины, вроде `5`, символа `'a'`, функции `\ x -> x+1`, а также структурированные величины, например список `[1,2,3]` или т.н. пара `('b',4)`.

Так же, как выражения обозначают непосредственно сами значения, типовыражения[4] (type expressions) являются синтаксическими термами, которые „отображают“ типы значений (или просто *типы*). Примерами типовыражений являются атомарные типы вроде `Integer` (целые бесконечной точности), `Char` (символы), `Integer → Integer` (функция, отображающая `Integer` в другой `Integer`), а также структурированные типы, например `[Integer]` (однородный список из целых чисел) или `(Char,Integer)` (пара символ/целое).

Все значения в Haskell являются „первоклассными“ (first-class values) сущностями, что означает, что они могут передаваться функциям в виде аргументов, возвращаться в качестве результатов, могут быть помещены в различные структуры, и т.д. С другой стороны, типы

[2] Позволю себе такой перевод названия модуля 'Prelude'. Обратите внимание на то, что название приводится с прописной буквы - Прим.пер.

[3] Luca Cardelli предложил использовать термин *typeful*

[4] Позволю себе несколлько вольный перевод - Прим.пер.

Haskell не являются первоклассными. Типы описывают значения, а ассоциацию значения с типом называют *типизацией* (typing). Можно описать типы для вышеприведённых значений таким образом:

$$\begin{array}{lll} 5 & :: & Integer \\ 'a' & :: & Char \\ inc & :: & Integer \rightarrow Integer \\ [1,2,3] & :: & [Integer] \\ ('b',4) & :: & (Char, Integer) \end{array}$$

Двойное двоеточие (::) можно прочитать просто как „имеет тип“.

Функции в Haskell обычно определяются с помощью серии *уравнений* (eqauation). Для примера, функция *inc* может быть описана следующим уравнением (единственным):

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

Уравнение является примером *объявления* (declaration). Другим видом объявлений являются *объявления сигнатуры типа* (type signature declarations) (§ 4.4.1), с помощью которых можно явно указать тип функции **inc**:

```
inc :: Integer → Integer
```

Более подробно объявления функций будут рассмотрены в главе 3.

Когда мы хотим указать, что выражение e_1 вычисляется (evaluates) или „редуцируется“ (reduces) в выражение e_2, мы записываем это следующим образом:

$$e_1 \Rightarrow e_2$$

Например:

$$inc\ (inc\ 3) \Rightarrow 5$$

Система статических типов Haskell устанавливает формальную связь между типами и значениями (§ 4.1.3). Она гарантирует, что программы, написанные на Haskell , являются *типобезопасными* (type safe), т.е. что программист нигде не ошибся (касаемо типов). Например, мы не можем просто сложить два символа: выражение `'a' + 'b'` не согласовано с типом (ill-typed). Главное преимущество статически типизированных языков хорошо известно: все ошибки, связанные с типами, обнаруживаются во время компиляции. Конечно, не все ошибки могут быть обнаружены системой типизации (или просто - системой типов): выражения, такие как $1/0$ являются валидными, но их вычисление приведёт к ошибке времени выполнения. Тем не менее система типов помогает находить многие ошибки уже во время компиляции, а также помогает компилятору генерировать более эффективный код (что не требуется никаких проверок или тэгов во время исполнения).

Система типов гарантирует, что указанные пользователем (программистом) сигнатуры типов корректны. Вообще, система типов Haskell позволяет не указывать сигнатур типов[5]; мы говорим, что система типов автоматически *выводит* (infers) корректные типы. Тем не менее, наличие в программе явно указанных типов считается „хорошим тоном“, помогающим документировать программы и легче находить ошибки.

[5] С некоторыми исключениями, описанными далее

[Надеемся, читатель обратил внимание на то, что идентификаторы типов записаны с прописной буквы (`Integer, Char`), а идентификаторы, означающие значения - со строчной буквы (`inc`). Это не простая условность: Haskell требует, чтобы это было так. Не забывайте, в Haskell `foo, fOo, fOO` являются различными идентификаторами.]

3.1 Полиморфные типы

Система типов Haskell включает и *полиморфные* (polymorphic) типы - типы, которые стоят над „обычными типами“. Полиморфные типы описывают семейства (families) типов. Для примера, $(\forall a)[a]$ является семейством типов, которое состоит, для любого `a`, из списка, в котором элементами являются `a`. Список целых чисел (например, `[1,2,3]`), список символов (например, `['a','b','c']`), список списков целых чисел (и т.д.) - всё это члены этого семейства. (Обратите внимание на то, что `[2,'b']` *не является* членом, так как не существует единого типа для 2 и $'b'$[6])

[Идентификаторы, такие как `a` называют *переменными типов* (type variables) и записываются с прописной буквы, чтобы отличать их непосредственно от конкретных типов (например, `Int`). Так как в Haskell имеются только типы, стоящие под квантором $\forall$, нет нужды явно указывать его. Это означает, что мы можем просто использовать, допустим, `[a]`. Другими словами, все переменные типов стоят под квантором $\forall$.]

Списки являются наиболее часто используемыми структурами данных в функциональных языках. Очень удобно показать основы полиморфизма с их помощью. Список `[1,2,3]` является сокращением для `1:(2:(3:[]))`, где `[]` является пустым списком, а `:` является инфиксным оператором, который добавляет его первый аргумент в голову своего второго аргумента (списка). Так как `:` является правоассоциативным, можно написать тоже самое так: `1:2:3:[]`.

В качестве примера определения функции, которая работает со списком, приведём функцию, которая подсчитывает количество элементов списка: Это определение говорит само за

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

себя. Можно просто прочитать эти уравнения так: „Длина пустого списка равна нулю; длина списка, первым элементом которго является `x`, а вторым `xs`, равна еденице плюс длине `xs`.“ (Обратите внимание, что название идентификатора `xs` является множественным числом от `x`.)

Этот пример открывает нам важную особенность Haskell : *сопоставление с образцом* (pattern matching). Левые стороны части уравнений содержат образец (например `[]` или `s:xs`). При вызове функции эти образцы сопоставляются с параметрами функции. (`[]` успешно подойдёт к пустому списку, `x:xs` подойдёт к списку из хотя бы одного элемента). При этом параметры „привязываются“ (binding to) к образцам, т.е. `x` привязывается к первому элементу, а `xs` к остальной части списка (возможно пустой). Если сопоставление прошло успешно, то вычисляется соответствующая правая часть и возвращается результат. Если сопоставление не прошло успешно, то проверяется следующее уравнение. Если же ни одна левая часть не подошла к аргументу, то это приводит к ошибочной ситуации.

Определение функций, использующих сопоставления с образцом, часто применяется в Haskell , так что программист должен уметь применять различные варианты. Мы вернёмся

[6] Другими словами - такой список не является гомогенным - Прим.пер.

к этому в главе 4.

Функция `length` является примером полиморфной функции. Она может быть применена к любому гомогенному списку, например `[Integer],[Char],[[Integer]]`:

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

Приведём пример двух часто используемых функций, работающих со списками: `head` возвращает первый элемент списка (голову), а `tail` - всё остальное (хвост).

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

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

В отличии от функции `length`, эти функции не определены для всех возможных аргументов. При вызове функции с пустым списком происходит ошибка (времени выполнения).

В случае с полиморфными типами, некоторые типы более „универсальны“, чем другие: тип `[a]` определён для большего числа типов, чем, допустим, `[Char]`. Другими словами, первый тип может стать производным с помощью подстановки первого. В Haskell система типов имеет два основных свойства: во-первых, любое правильно типизированное выражение гарантированно имеет основной тип (principal type)[7]; во-вторых, основной тип может быть выведен (be inferred) автоматически (§ 4.1.3). В сравнении с мономорфнно типизированными языками (такими как C) полиморфизм повышает т.н. „описательные возможности“ (expressiveness), а выведение типов облегчает работу программиста.

Основной тип выражения или функции это наименее общий (least general) тип, который „отвечает всем экземплярам выражения“. Например, основным типом для `head` является `[a]->a; [b]->a,a->a`, и даже `a` является корректным типом, но он слишком универсален. А `[Integer]->Integer` является слишком специфичным. Существование основного типа является отличительной чертой *системы типов Хиндли-Милнера* (Hindley-Milner type system), которая образует базовую систему типов Haskell , ML, Miranda[8], и некоторых других (главным образом функциональных) языков.

3.2 Пользовательские типы

В Haskell , конечно же, существует возможность определять свои собственные типы, используя объявления `data`. Приведём несколько примеров (§4.2.1).

Важным предопределённым типом Haskell является `Bool`:

```
data Bool = False | True
```

[7] Объяснение даётся далее

[8] „Miranda“ является торговой маркой Research Softwate, Ltd

Тип, определённый здесь, имеет два значения: `True` и `False`. Тип `Bool` является примером пустого (nullary) *конструктора типов* (type constructor), а `True` и `False` являются примерами двух пустых *конструкторов данных* (data constructors). Последние обычно называют просто *конструкторами* (constructors).

Точно так же мы можем определить тип `Color`:

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

И `Bool`, и `Color` есть перечисляемые типы, так как состоят из пустых конструкторов данных.

Приведём пример типа с одним конструктором данных:

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

Из-за наличия лишь одного конструктора, такой тип имеет название *тип-кортеж*[9] (tuple type), так как это, по сути, декартово произведение (в данном случае - двух аргументов) других типов. В противоположность этому, типы с многочисленными конструкторами, такие как `Bool` и `Color` называют (неперекрывающимися) объеденениями или суммами типов.

Более интересно то, что `Point` является примером полиморфного типа: для любого типа t оно определяет точку на Декартовой плоскости, которая использует t в качестве типа координат. Тип `Point` имеет унарный конструктор типа, так как для любого t он конструирует новый тип `Point t`. (Таким образом, `[]` является конструктором типа. Для любого типа t он конструирует новый тип $[t]$. Синтаксис Haskell позволяет записать `[] t` как $[t]$. Точно так же, $->$ является конструктором типа: для любых типов t и u конструируется тип $t -> u$ - т.е. тип функции, отображающей аргумент типа t в аргумент типа u.)

Обратите внимание, что бинарный конструктор данных `Pt` имеет тип `a->a->Point a`, что, в свою очередь, утверждает валидность следующих типов:

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

Выражения, такие как `Pt 'a' 1` не могут быть верными, так как `'a'` и `1` имеют разные типы.

Очень важно различать *конструктор данных*, который конструирует *значение* от *конструктора типов*, который конструирует *тип*. Первые работают во время выполнения, когда мы вычисляем что-либо, в то время как последние работают во время компиляции для обеспечения типобезопасности.

[Конструкторы типов, такие как `Point`, и конструкторы данных, такие как `Pt`, находятся в различных пространствах имён (namespaces). Это означает, что можно давать им одинаковые имена:

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

На первый взгляд это мешает, но часто это служит для того, чтобы связь между типом и его конструктором данных чувствовалась лучше.]

[9] Кортежи очень походят на структуры из других языков программирования

3.3 Рекурсивные типы

Типы могут быть рекурсивными, например в случае бинарных деревьев:

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

Здесь определён полиморфный тип бинарного дерева, элементами которого являются либо листья, т.е. узлы, содержащие значения типа a, либо внутренние узлы (ветви), содержащие (рекурсивно) два поддерева.

Когда вы встретите объявления данных такого вида, вспомните, что `Tree` является конструктором типа, в то время как `Branch` и `Leaf` являются конструкторами данных. Помимо создания связи между этими конструкторами, вышеприведённое определение создаёт следующие типы для `Branch` и `Leaf`:

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

В этом примере мы создали уже довольно сложный тип, который позволяет нам написать рекурсивные функции, использующие его. Например, представьте, что необходимо создать функцию `fringe`[10], которая вернёт список листьев (слева направо). Как правило, сначала полезно описать тип функции. Видно, что типом является `Tree a -> [a]`, т.е. `fringe` - есть полиморфная функция, которая для любого типа a[11] отображает деревья из a в список из a. Сразу же следует естественное определение:

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

Здесь ++ есть инфиксный оператор, который соединяет (конкатенирует) два списка (его полноценное описание будет дано в разделе 9.1). Как и в примере с `length`, функция `fringe` определена с использованием сопоставления с образцом. Единственная разница заключается в том, что здесь используется сопоставление с образцом определённых пользователем конструкторов: `Leaf` и `Branch`.

[Формальные параметры легко отличить, так как они записываются со строчной буквы.]

3.4 Синонимы типа

Для удобства Haskell имеет способ определения *синонимов типа* (type synonyms), т.е. имён, которые используются в качестве идентификаторов типа. Они создаются с помощью декларации `type` (§4.2.2). Вот несколько примеров:

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

Синонимы типа не определяют (не создают) новые типы, они просто задают новые имена для уже существующих. Например, тип `Person -> Name` является точным эквивалентом

[10] Переводится как "граница"или "кайма Прим.пер.

[11] Стоит заметить, что существуют определённые ограничения на типы аргументов, которые определяются используемыми самой функцией операциями. На данный момент читателю полезно считать, что может быть использован *любой* тип. Особенности системы типов будут разобраны чуть позже - Прим.пер.

`(String,Address) -> String`. Обычно новые имена короче исходных, но это не единственное предназначение синонимов - они могут повысить „читабельность“ программ. Вышеприведённые примеры хорошо показывают это. Мы можем дать новые имена даже полиморфным типам:

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

Это тип „ассоциативного списка“, который „ассоциирует“ значения типа a со значениями типа b.

3.5 Встроенные типы - не исключение

Ранее мы рассмотрели несколько „встроенных“ типов: списки, кортежи, целые числа,а также символы. Мы рассмотрели, *как* можно создать новые типы. Может возникнуть вопрос: являются ли „встроенные“ типы чем-то особенным? Ответом будет *нет*. Специальный синтаксис, используемый в некоторых примерах, был введён лишь для удобства и для согласования с исторически сложившимися соглашениями (конвенциями), но не имеет каких-либо семантических следствий.

Интересно, как бы выглядели объявления типов для этих встроенных типов, если бы мы имели возможность использовать специальный синтаксис для их определения. Например, тип `Char` мог бы быть записан так:

```
data Char  = 'a' | 'b' | 'c' | ...     Это неверный
           | 'A' | 'B' | 'C' | ...     код Haskell
           | '1' | '2' | '3' | ...     –
           ...                         –
```

Эти имена конструкторов синтаксически неверны; для того, чтобы преодолеть это, необходимо написать что-то вроде следующего:

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

Такие конструкторы не очень подходят для описания символов.

В любом случае, написание „псевдо-Haskell “ кода таким образом помогает изучить специальный синтаксис. Видно, что `Char` является простым перечисляемым типом, содержащим большое количество пустых конструкторов. Если представлять `Char` так, то становится понятно, что мы можем сопоставлять их с символами (которые являются образцом) точно так же, как и в случае с любым другим конструктором обычного типа.

[В этом примере используются *комментарии*. Символы `--` и всё остальное (до конца строки) игнорируется. В Haskell имеются *вложенные* (nested) комментарии, которые имеют следующий вид: `{-...-}` (§ 2.2).]

Можно определить тип `Int` (целые числа фиксированной точности) и `Integer` (целые числа бесконечной точности) следующим образом (с использованием псевдокода[12]):

[12] Псевдокод не является верным исходным кодом Haskell ! - Прим.пер.

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

где -65532 и 65532 являются нижней и верхней границами для целых с фиксированной точностью в данной реализации. `Int` имеет гораздо более объемное объявление, нежели `Char`, но оно тоже конечно! В противоположность этому, `Integer`, записанный псевдокодом, призван описать *бесконечное* перечисление.

Кортежи тоже легко определить таким обазом (псевдокод):

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

Каждое из приведённых объявлений определяет кортеж определённой длины, в которых скобки (...) играют роль выражения и выражения типа одновременно (конструктора типа). Стоит отметить, что многоточие означает лишь то, что таких определений дано бесконечное множество. Это означает, что в Haskell можно использовать кортежи любой длины.

Можно легко определить и списки, которые являются рекурсивными типами (псевдокод):

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

Хорошо видно то, о чём мы говорили раньше: `[]` является пустым списком, а `:` является инфиксным конструктором. `[1,2,3]` должно являться эквивалентным для `1:2:3:[]`. (так как `:` правоассоциативен.) Типом `[]` является `[a]`, а типом `:` является `a->[a]->[a]`.

[На самом деле, определение „:“ является валидным синтаксисом - инфиксные конструкторы разрешены в объявлениях `data` и отличаются от инфиксных операторов (для сопоставления с образцом) тем, что они начинаются с „:“ (достаточно и одного „:“).]

К этому моменту читатель должен понимать различия между кортежами и списками. Обратите внимание на рекурсивную натуру типа списка. Он содержит любое число элементов одинакового типа. Обычный кортеж имеет нерекурсивный тип и содержит фиксированное количество элементов (возможно) различных типов. Правила типизации для списков и кортежей должны быть понятны:

- Для $(e_1, e_2, \ldots, e_n),\ n \geq 2$ типом кортежа является $(t_1, t_2, \ldots, t_n)$, где t_i является типом e_i
- Для $[e_1, e_2, \ldots, e_n],\ n \geq 0$ типом списка является $[t]$, где все e_i имеют одинаковый тип t

3.5.1 Составление списков и арифметические последовательности

Как и в Lisp, списки являются одним из главных средств в Haskell . Специально для удобства работы с ними было добавлено достаточно много „синтаксического сахара“. Помимо конструкторов списков существуют так называемые составления списков (list comprehensions):

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

Это выражение можно прочитать как „список из всех `f x`, таких, что `x` берётся[13](is drawn) из `xs`“. Выражения типа `x <- xs` называют *генераторами* (generators). Их может быть несколько:

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

Здесь мы указываем, что создаётся список из Декартова произведения двух других списков - `xs` и `ys`. Элементы выбираются так, как будто конструкторы были вложены слева направо (самый правый генератор извлекает элементы „чаще“). Это означает что для `xs` равного $[1,2]$ и `ys` равного $[3,4]$ результат будет выглядеть так: `[(1,3),(1,4),(2,3),(2,4)]`.

Помимо генераторов существуют особые булевы выражения, названные гардами (guards). Гарды накладывают ограничения на генерируемые элементы. Вот пример определения, наверное самого используемого метода сортировки:

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

Помимо этого, для поддержки списков Haskell имеет специальный синтаксис для *арифметических последовательностей* (arithmetic sequences), который лучше всего понять из примера:

```
[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)
```

Арифметические последовательности будут более подробно рассмотренны в разделе 8.2, а „бесконечные списки“ в разделе 3.4.

3.5.2 Строки

Другим примером „синтаксического сахара“ для встроенных типов являются строки: строка „`hello\` является сокращением от `['h','e','l','l','o']`. Конечно же, типом „`hello\` является `String`, где `String` есть предопределённый синоним (синонимы были рассмотрены ранее):

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

Это означает, что мы вправе использовать функции, работающие с полиморфными списками, для обработки строк. Например:

```
"hello" ++ " world"   ⇒   "hello world"
```

[13] Извлекается, вытягивается - Прим.пер.

4 Функции

Так как Haskell является функциональным языком, можно ожидать, что функции играют в нём самую главную роль. В этом разделе мы рассмотрим некоторые аспекты функций Haskell .

Во-первых, представьте себе такое определение функции, которая складывает два аргумента:

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

Это пример *функции Карри*[14] (curried function). Аппликация (application) этой функции имеет форму `add` e_1 e_2, что эквивалентно `(add` e_1`)` e_2, так как вызовы функций левоассоциативны. Другими словами, вызов `add` с одним аргументом „создаёт“ функцию[15], которая затем вызывается со вторым аргументом. Это прекрасно согласуется с типом `add`: `Integer -> Integer -> Integer`, который эквивалентен `Integer -> (Integer -> Integer)`[16], т.е. $->$ правоассоциативен. Получается, что можно объявить `inc` с помощью `add` в таком виде:

```
inc = add 1
```

Это пример *частичной аппликации* функции Карри (partial application), и это один из способов, которым функция может быть возвращена как значение. Давайте рассмотрим пример того, когда полезно передавать функцию в качестве аргумента. Функция `map` является хорошим кандидатом для изучения:

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

[Аппликация функции имеет больший приоритет, чем любой инфиксный оператор, а значит второе уравнение будет разобрано как `(f x) : (map f xs)`] Функция `map` является полиморфной, а её тип явно отражает тот факт, что её первый аргумент есть функция. Обратите внимание на то, как связаны одинаковые типы a и одинаковые типы b (в выражении типа два вхождения каждого из них). В качестве примера использования `map` увеличим все элементы списка:

```
map (add 1) [1,2,3] ⇒ [2,3,4]
```

Рассмотренные примеры показывают первоклассную сущность функций, которые при таком использовании обычно называются *функциями высшего порядка* (higher-order functions).

[14] Человека, который распространил идею таких функций звали Хаскелем Карри (Haskell Curry). Для того, чтобы получить не-Карри функцию, необходимо использовать кортеж:

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

Теперь видно, что `add` на самом деле является функцией всего одного аргумента!

[15] Не забывайте, что функции в Haskell являются „первоклассными“ - Прим.пер.

[16] Функция принимает один `Integer` и возвращает функцию, принимающую один `Integer` и возвращающую один `Integer`. Обратите внимание на скобки - Прим.пер.

4.1 Лямбда-функции

Вместо того, чтобы использовать для определения функций уравнения, можно определить безымянные или „анонимные“ функции, которые называют „лямбда-функциями“. Например, функция, эквивалентная `inc`, может быть записана в виде `\x -> x+1`. Функция, эквивалентная `add`, записывается как `\ x -> \ y -> x+y`. Вложенные лямбда-функции можно записать и другим образом: `\x y -> x+y`. Вообще, уравнения:

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

на самом деле являются сокращенными вариантами

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

Мы вернёмся к этому чуть позже.

В общем случае, если `x` имеет тип t_1, а `exp` имеет тип t_2, тогда `\x->exp` имеет тип $t_1 -> t_2$.

4.2 Инфиксные операторы

Инфиксные операторы являются обычными функциями, которые можно определить с помощью уравнений. Вот пример оператора конкатенации списка:

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

[Лексически инфиксные операторы состоят только из „символов“ в противоположность обычным идентификаторам, которые могут состоять из букв и цифр (§ 2.4). В Haskell нет префиксных операторов, кроме минуса (-), который является одновременно и инфиксным, и префиксным.]

В качестве ещё одного примера инфиксного оператора, рассмотрим оператор, используемый для *композиции функций* (function composition):

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

4.2.1 Секции

Так как инфиксные операторы являются обычными функциями, необходимо иметь возможность частично применять их. В Haskell частичная аппликация инфиксных операторов называется *секцией* (section). Например:

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

[Скобки обязательны.]

Последний вид секции приводит инфиксный оператор к эквивалентному функциональному виду. он удобен при передаче инфиксного оператора в функцию, например так: `map (+) [1,2,3]` (читатель должен убедиться в том, что это вернёт список функций!). Такая секция применяется и когда необходимо определить для функции сигнатуру типа, как было показано ранее в примерах с (++) и (.).

Теперь видно, что `add`, определённый ранее, является простым `(+)`, а `inc` это просто `(+1)`! Вот такие определения отлично сработают:

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

Теперь у нас есть возможность привести инфиксный оператор к функциональному значению, но можно ли сделать наоборот? Да - простым заключением идентификатора в одинарные кавычки (backquots). Например, `x ‘add‘ y` это то же самое, что и `add x y`[17]. Некоторые функции удобнее записывать именно таким обазом. Допустим, предикат[18], определяющий, принадлежит ли элемент списку `elem`, можно использовать как `x ‘elem‘ xs`. Попробуйте прочитать это вслух, ведь сделать это так просто: „x является элементом xs“.

[Существуют особые правила, касающиеся секций с префиксным/инфиксным оператором `-`; см. (§3.5,§3.4).]

К этому моменту читатель должно быть уже запутался от такого количества способов описания функций! Решение включить эти механизмы в язык было основано на многочисленных соглашениях, принятых достаточно давно, и частично отражает желание сделать язык монолитным (например, при сравнении инфиксных и регулярных функций).

4.2.2 Объявления устойчивости

Объявление устойчивости (fixity declaration) может быть дано для любого инфиксного оператора или конструктора (включая и те, которые образованы из обыкновенных идентификаторов, например, `‘elem‘`). Такое объявление указывает уровень предшествования (precedence level): от 0 до 9, где 9 - наибольший возможный уровень; подразумевается, что обычная аппликация имеет 10 уровень. Существует неассоциативные, лево-, и правоассоциативные уровни. Например, объявления устойчивости для `++` и `.` могут быть заданы так:

```
infixr 5 ++
infixr 9 .
```

Оба объявления определяют правоассоциативность с уровнем в 5 и 9 соответственно. Левоассоциативность задаётся с помощью `infixl`, а задать неассоциативность можно с помощью `infix`. Имеется возможность задавать устойчивость сразу нескольких операторов/конструкторов. По умолчанию, задаётся 9 уровень устойчивости. (см. §5.9)

4.3 Нестрогие функции

Представьте, что функция `bot` определена следующим образом:

```
bot = bot
```

[17] Обратите внимание, что `add` заключена не в *апострофы*, т.е. `'f'` это символ, а `‘f‘` это инфиксный оператор. К счастью, большинство терминалов отображают различие лучше, чем данный текст.

[18] Напомним, что предикатом называется функция, возвращающая булево значение - Прим.пер.

Другими словами, `bot` является незавершающимся выражением (non-terminating expression). Мы обозначаем *значение* незавершающегося выражения как $\perp$ (произносится „основание“ (bottom)). Выражения, которые приводят к ошибке времени выполнения (например, `1/0`), имеют это значение. Такая ошибка является невосстановимой (non-recoverable): программа не может продолжить выполнение. Ошибки системы ввода-вывода, такие как EOF, являются восстановимыми и обрабатываются другим образом (Вообще-то, такая ошибка I/O[19] не является „ошибкой“, а скорее исключением. Более подробно об этом - в разделе 7.)

Функция `f` называется *строгой*, если при аппликации к незавершающемуся выражению она тоже не завершается. Другими словами, f является строгой, если значением `f bot` является *bot*. В большинстве языков программирования *все* функции являются строгими. Но это не так в Haskell . Вот пример константной функции:

```
const1 x = 1
```

Значением `const1 bot` в Haskell является 1. Так как `const1` не нуждается в аргументе, она и не пытается вычислить его, т.е. никогда не вовлекается в незавершающиеся вычисления. По этой причине нестрогие функции часто называют „ленивыми функциями“ (lazy functions), так как вычисляют свои аргументы „лениво“ или „по необходимости“.

Так как ошибки и незавершающиеся значения в Haskell семантически равнозначны, значение `const1 (1/0)` выдаёт верный ответ - 1.

Нестрогие функции очень полезны в различных ситуациях. Главное их преимущество состоит в том, что они освобождают программиста от многих проблем, связанных с порядком вычислений. Вычислимо дорогие значения могут передаваться в функции без опаски: если они не потребуются, то никогда не будут вычислены. Представьте себе *бесконечную* структуру данных.

Легче представить нестрогие функции так, что Haskell производит вычисления с помощью *определений* (definitions), а не *присваиваний* (assignments), как в традиционных языках программирования. Следующее определение

```
v = 1/0
```

произносится как „объявить `v` как $1/0$“, а не как „вычислить $1/0$ и записать результат в `v`“. Только когда потребуется узнать *значение* `1/0`, возникнет ошибка деления на нуль. Само по себе определение не влечёт за собой никаких вычислений. Программирование с помощью присваиваний требует особой внимательности к порядку присваиваний: смысл программы чрезвычайно сильно меняется от порядка, в котором выполняются присваивания. Работать с объявлениями, в противоположность, проще: порядок не имеет никакого значения.

4.4 „Бесконечные“ структуры данных

Одно из преимуществ Haskell кроется в том, что конструкторы данных тоже являются нестрогими. Это не должно шокировать, так как конструкторы являются по сути просто специальным видом функций (отличительная особенность которых заключается в том, что они участвуют в сопоставлении с образцом). Например, конструктор списков (:) является нестрогим.

[19] Здесь и далее вместо В/В используется англоязычное сокращение - I/O - Прим.пер.

Нестрогие конструкторы разрешают определять (концептуально) *бесконечные* (infinite) структуры данных. Вот бесконечный список целых единиц:
<рискунок отсутвует>

```
ones = 1 : ones
```

Более интересный пример:

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

Т.е. `numFrom n` является бесконечным списком, начинающимся с `n`. Можем построить бесконечный список квадратов:

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

(Обратите внимание на использование секции; `(^)` является инфиксным оператором возведения в степень.)

В конечном счете, мы извлекаем лишь конечные части списка для вычислений. В Haskell существует огромное количество функций, которые делают это: `take`, `takeWhile`, `filter`, и т.д. Haskell содержит множество встроенных типов и функций - это назыают „Стандартной Прелюдией“[20] (Standart Prelude). Полностью „прелюдия“ разбирается в дополнении A Haskell Report . Посмотрите на `PreludeList` - там содержится множество функций, использующих списки. Например, `take` берёт[21] первые `n` элементов из списка:

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

Объявление `ones` являет собою пример *цикличного списка* (circular list). Обычно ленивые вычисления сильно влияют на производительность, да и список может быть реализован как настоящая цикличная структура, сохраняющая память.

Другим примером цикличности служит функция, возвращающая последовательность чисел Фибоначчи:

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

в которой `zip` это стандартная функция из прелюдии, возвращающая парами элементы её двух аргументов (списков):

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

Обратите внимание на то, как `fib`, т.е. бесконечный список, определён через самого себя, как будто бы он „гонится за своим хвостом“. Можно представить это таким образом: см. рис. 1.

Другой пример использования бесконечных списков см. в разделе 4.4.

[20] Называемо на протяжении всего повествования просто „прелюдией“ - Прим.пер.
[21] В оригинале - "удаляет"(removes) - Прим.пер.

4.5 Обработка ошибок

Haskell имеет встроенную функцию `error`, которая имеет тип `String->a`. Это довольно странная функция: из её сигнатуры типа можно сказать, что она возвращает значения полиморфного типа, о котором ничего не известно. Более того, она даже не принимает аргумента с таким типом!

Вообще *существует* значение, которое имеют любые типы: $\perp$. Поэтому, семантически это и есть то самое значение, которое возвращает функция `error` (ведь все ошибки имеют $\perp$ значение). Тем не менее, можно ожидать, что нормальная реализация должна выводить сообщение об ошибке на экран. Эта функция используется, если что-то пошло не так. Например, настоящее определение `head` выглядит так:

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

5 Case-выражения и сопоставление с образцом

Ранее мы привели несколько примеров сопоставления с образцом в определении функций - например `length` и `fringe`. В этом разделе мы рассмотрим процесс сопоставления более детально (§ 3.17)[22].

Образцы (шаблоны) не являются „первоклассными“ сущностями; существует лишь конечное количество их различных видов. Мы уже встречали примеры шаблонов *конструкторов данных*; и `length`, и `fringe` использовали такие шаблоны: первая - в конструкторе „встроенного“ типа (списка), вторая - в пользовательском типе (`Tree`). Сопоставление разрешено в конструкторах любых типов(пользовательских, или нет): кортежи, строки, числа, символы и т.д. могут быть использованы. Вот пример функции `contrived`[23], которая сопоставляется с кортежем констант:

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

В этом примере встречается *вложенные* (nested patterns) шаблоны (разрешена любая глубина вложенности).

Говоря техническим языком, *формальные параметры*[24] тоже являются шаблонами - просто они *всегда сопоставляются со значением* (never fail to match a value). В качестве „побочного эффекта“ успешной подстановки формальные параметры привязываются к значениям. По этой причине, шаблоны в любом уравнении не должны иметь более одного вхождения одинакового формального параметра (свойство, называемой *линейностью* (linearity) §3.17, §3.3, §4.4.2).

[22] Сопоставление с образцом в Haskell отличается от того, что применяется в логическом программировании (Prolog); его можно рассматривать как "одностороннее сопоставление“ (one-way matching), в то время как Prolog имеет и „двустороннее сопоставление“ (two-way matching) (через унификацию), а также бэктреккинг (backtracking)

[23] Её название переводится как „сложная“ или „хитрая“ и говорит само за себя - Прим.пер.

[24] Haskell Report называет их „переменными“

Шаблоны, формальные параметры которых всегда сопоставляются, называются *неопровержимыми* (irrefutable), в противоположность *опровержимым* (refutable). Шаблон из `contrived` является опровержимым. Существует 3 типа неопровержимых шаблонов, два из которых мы рассмотрим сейчас (остальной подождёт до раздела 4.4).

As-Шаблоны. Иногда удобно дать имя шаблону для того, чтобы использовать его справа от знака равенства. Например, можно написать функцию, которая дублирует первый элемент списка так:

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

(Вспомните, что „:“ правоассоциативен.)Обратите внимание на то, что `x:xs` присутствует с двух сторон. Можно написать `x:xs` всего один раз для улучшения кода: [25].

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

Такой шаблон всегда приводит к успешному сопоставлению, а его „подшаблон“ (в нашем случае это `x:xs`), конечно же, может и не приводить.

Подчеркивания. Часто случается, что мы проводим сопоставление для значения, которое нас на самом деле не интересует. Для примера: функции `head` и `tail` из раздела 2.1 можно записать следующим образом:

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

Где мы „указываем“, что нас не волнуют определённые части. Каждое подчеркивание отдельно сопоставляется хоть с чем, но, в отличии от формального параметра, оно ни с чем не *связывается*. Именно поэтому одно уравнение может содержать несколько таких подчеркиваний.

5.1 Семантика сопоставления с образцом

Мы обсудили то, как происходит сопоставление, что такое опровержимые шаблоны, и неопровержимые... Но мы до сих пор не знаем, *как* происходит сопоставление. В каком порядке происходит связь? Что происходит, если процесс не завершился успешно? Этот раздел призван объяснить всё это.

Сопоставление с образцом может пройти либо *успешно* (succeed) , либо *неуспешно* (fail), а может *дивергировать* (diverge). Успешное сопоставление связывает формальные параметры. Дивергенция возникает, если значение, требуемое образцу, содержит ошибку ($\perp$). Сам процесс сопоставления проходит „сверху вниз, слева направо“. Неуспешное сопоставление в любой части шаблона приводит к неуспешному сопоставлению всего шаблона, а значит проверяется следующее выражение. Если все уравнения прошли процесс сопоставления неуспешно, то значение аппликации функции равно $\perp$, что приводит к ошибке времени выполнения.

Например, $[1, 2]$ проходит сопоставление с $[0, bot]$: единица не может быть сопоставлена с нулём, а значит результат - неуспешное сопоставление. (Вспомните, что *bot*, определённая ранее, является переменной, связанной с $\perp$.) Но если $[1, 2]$ сопоставляется с $[bot, 0]$, то сравнение 1 с *bot* приводит к дивергенции (т.е. $\perp$).

[25] Помимо этого, некоторые реализации Haskell будут работать эффективнее

Шаблоны высшего уровня (top-level) могут иметь *гарды*, как в следующем определении функции, которая возвращает знак числа[26]:

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

Обратите внимание, что один шаблон может иметь несколько гардов. Как и с шаблонами, они сканируются сверху вниз, слева направо.

5.2 Пример

Сопоставление с образцом порой является подводным камнем функций. Например, представьте себе следующее определение `take`:

```
take0   0 _          =    []
take0   _ []         =    []
take0    n (x : xs)  =   x :  take (n-1) xs
```

а вот её немного изменённая версия (два первых уравнения поменялись местами):

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

А теперь посмотрите на это:

```
take0 0 bot ⇒ []
take1 0 bot ⇒ ⊥

take0 bot [] ⇒ ⊥
take1 bot [] ⇒ []
```

Видно, что `take0` объявлена „лучше“ по отношению к своему второму аргументу, а `take1` - наоборот. Трудно сказать, какое определение лучше. Просто помните, что это так. (прелюдия включает `take0`.)

5.3 Case-выражения

Сопоставление с образцом является т.н. „диспетчеризацией“, основанной на структурных свойствах значений. Обычно мы не хотим определять *функцию* каждый раз, когда нам необходимо ветвление. Тем не менее, мы привели лишь этот способ. В Haskell существуют case-выражения, решающие эту проблему. На самом деле, в Haskell Report сопоставление с образцом объясняется после case-выражений, которые считаются более простыми. Определение функции следующего вида:

$$\begin{aligned} f\ p_{11}\ \ldots\ p_{1k} &= e_1 \\ &\ldots \\ f\ p_{n1}\ \ldots\ p_{nk} &= e_n \end{aligned}$$

где p_{ij} является шаблоном, семантически эквивалентно:

где `xi` - новые идентификаторы. (Более обобщенная реализация, включая гарды, приведена в §4.4.2.) Например, определение `take0`, приведённое выше, эквивалентно следующему:

[26] Функция „сигнум“, как ясно и из поставновки, и из её определения - Прим.пер.

```
f x1 x2 ... xk = case (x1, ..., xk) of   (p11, ..., p1k) -> e1
                                         ...
                                         (pn1, ..., pnk) -> en

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

Мы не говорили ранее, что для типобезопасности все типы правых частей case-выражений или уравнений (определяющих функцию) должны быть одинаковыми; если быть более точным - они должны иметь одинаковый основной тип[27].

Правила сопоставления для case-выражений такие же, как и в определениях функций, так что здесь нет ничего особенного (кроме синтаксиса case-выражений). Существует особенный вид case-выражений, которые используются повсеместно: *условные выражения* (conditional expressions). В Haskell условные выражения имеют общеизвестную форму:

```
if  e1 then e2  else  e3
```

которые на самом деле являются сокращением следующего case-выражения:

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

Хорошо видно, что выражение e_1 должно иметь тип `Bool`, а e_2 и e_3 должны иметь одинаковые типы (но могут быть произвольными). Другими словами, если рассматривать `if-then-else` как функцию, то она имеет следующий тип: `Bool->a->a->a`.

5.4 Ленивые образцы (шаблоны)

В Haskell существует ещё один тип шаблонов. Им дано название *ленивые шаблоны* (lazy patterns), и они имеют следующий вид: `~pat`. Такие шаблоны являются неопровержимыми: если значение v сопоставляется с `~pat`, то сопоставление проходит успешно независимо от *pat*. Если идентификатор в *pat* позже „используется“ в правой части, он будет привязан либо к *pat*, либо к $\perp$, если сопоставление прошло неуспешно.

<рисунок отсутсвует>

Ленивые шаблоны используются, если структуры данных определены рекурсивно. Например, бесконечные списки являются идеальной основой для написания *симуляторов* (simulation programs), и в этом контексте бесконечные списки обычно называют *потоками* (streams). Представьте, что мы хотим смоделировать простейшее взаимодействие между серверным процессом (`server`) и клиентским процессом (`client`),где `client` шлёт *запрос* (request) к `server`, а `server` отвечает на каждый запрос определённым *откликом* (response). Такое взаимодействие показано на рис.2. (Учтите, что `client` принимает инициализирующее сообщение в качестве аргумента.) Вот соответствующий код:

Эти рекурсивные уравнения являются прямым отображением рисунка.

[27] Об *основных типах* (principal types) рассказывается чуть позже - Прим.пер.

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

Давайте предположим, что структура клиента и сервера выглядит примерно так:

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

Предполагается, что `next` - это функция, которая по отклику сервера вычисляет следующий запрос, а `process` - это функция, которая обрабатывает запрос клиента и вычисляет соответствующий отклик.

К сожалению, эта программа имеет огромную проблему: она не будет работать! Проблема заключается в том, что `client`, как показано в рекурсивном определении `reqs` и `resps`, пытается произвести сопоставление со списком отклика до того, как отправит первый запрос! Другими словами, сопоставление „производится слишком рано“. Конечно, можно переопределить `client` следующим образом:

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

Хотя это и работает, но такая запись читается гораздо хуже. Вместо этого можно использовать ленивый шаблон:

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

Так как ленивые шаблоны являются неопровержимыми, сопоставление произойдёт сразу же, позволяя начальному запросу быть „отправленным“, что, в свою очередь, генерирует первый ответ. Система „завязана“, и рекурсия сделает всё остальное.

Чтобы показать работу программы, можно определить:

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

Затем мы получаем:

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

Мы можем использовать ленивые шаблоны и для объявления чисел Фибоначчи. Вот старый вариант:

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

А вот новый:

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

Эта версия `fib` имеет одно (небольшое) преимущество, которое заключается в том, что она не использует `tail` в правой части, так как он находится в разрушенном виде (destructed form) слева (в качестве `tfib`).

[Такой тип уравнений называется *привязкой шаблона* (pattern binding), так как это уравнение высшего уровня, в котором вся левая часть является шаблоном, т.е. и `fib`, и `tfib` связываются в области видимости этого объявления.]

Принимая во внимание наши доводы, мы должны считать, что программа не выводит ничего. Тем не менее, она *выводит* текст на консоль по простой причине: в Haskell привязки шаблонов имеют впереди как бы неявный `~`, отражающий стандартное поведение и призванный решить некоторые проблемы, оставшиеся за рамками этого документа. Видно, что ленивые шаблоны (хотя и не явно) играют важную роль в Haskell .

5.5 Область видимости и вложенные формы

Порой необходимо образовать в выражении вложенную область видимости. Это полезно для создания связок (bindings), которые нигде более не видимы - т.е. для создания как бы „блочной структуры программы“. В Haskell для этого есть 2 способа:

Let-выражения (let-expressions). В качестве простого примера приведём следующее объявление:

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

Набор (множество) из связок, созданных `let`-выражениями является *взаимно рекурсивными* (mutually recursive) и неявно ленивыми (т.е. имеют неявный в начале). В определениях `let`-выражений разрешены только *сигнатуры типов* (type signatures), *связки функций* (function bindings), *привязки шаблонов* (pattern bindings).

Where-выражения (where clauses)[28]. Иногда удобно использовать гарды для связок. Для этого можно применить *Where-выражение*:

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

Обратите внимание, что так нельзя сделать с помощью `let`-выражений, которые видимы только из выражений, которые они „окружают“. `where`-выражения разрешены только на высшем уровне набора уравнений или `case`-выражений. Они обладают такими же свойствами и ограничениями, как и `let`-выражения.

Эти две формы вложенной области видимости очень похожи, но помните, что `let`-выражения являются полноценными выражениями, а `where`-выражения - это всего лишь дополнительный синтаксис для объявления функций и `case`-выражений.

[28] Здесь и далее будет использоваться такой перевод, хотя они и не являются полноценными выражениями, а лишь входят в состав других - Прим.пер.

5.6 Разметка

Читатель должно быть давно задаётся вопросом, как же Haskell обходится без точки с запятой (или другого типа разделителей) для обозначения окончания уравнения, объявления и т.д. Например, посмотрите на `let`-выражение из последнего раздела:

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

Почему же разборщик не понимает это как:

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

?

Ответ заключается в том, что Haskell использует двумерное расположение синтаксиса, называемое *разметкой* (layout), которая основана на том, что объявления „разделены на несколько колонок“. В предыдущем примере `y` и `f` находятся в одной колонке. Правила разметки обсуждаются в Haskell Report (§2.7,§B.3), но вообще они обычно усваиваются достаточно легко на практике. Просто запомните две вещи:

Во-первых, следующий символ, идущий за `where,let,of` открывает объявление этих конструкций и определяет начальную колонку (это относится и к `where` в объявлениях классов и экземпляров, о которых идёт речь в разделе 5). Таким образом, мы можем начинать объявление на этой же строке с ключевого слова (keyword), символа новой строки, и т.д. (`do`, рассматриваемый далее, тоже использует разметку)

Во-вторых, всегда проверяйте, находится ли первая колонка текущего уровня вложенности правее первой колонки его окружения (иначе возникнет двусмысленность). „Окончание“ объявления происходит, когда что-то встречается с тем же отступом (или находится левее)[29].

Слово „разметка“ встречается как сокращение для *явного* механизма группировки, который заслуживает внимания, так как в некоторых случаях является очень полезным. Пример с `let` эквивалентен следующему:

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

Обратите внимание на фигурные скобки и точку с запятой. Можно располагать несколько объявлений на одной строке, например так:

Другой пример явной разметки приведён в §2.7.

[29] Haskell использует соглашение, что табуляция состоит из 8 пробелов, поэтому будьте внимательны со своим редактором, который может иметь другое соглашение

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

Разметка такого типа сильно уменьшает „синтаксический мусор“, а значит улучшает вид кода. Её легко принять и приятно использовать.

6 Классы типов и перегрузка

Существует ещё одно свойство (особенность) системы типов Haskell , которое выделяет его среди других языков программирования. Вид полиморфизма, о котором мы так много говорили, называется *параметрическим* (parametric) полиморфизмом. Существует и другой вид полиморфизма, названный *специальным* (ad hoc) полиморфизмом:

- Литералы (например, 1 и 2) обычно отображают как целые с фиксированной точностью, так и целые с произвольной точностью числа.
- Числовые операторы (например, +) обычно работают с различными видами чисел.
- Оператор равенства (equality operator) (в Haskell это ==) обычно работает с числами и многими другими (но не со всеми) типами.

Обратите внимание, что поведение операторов (и др.) меняется от типа к типу. Иногда поведение является неопределённым (undefined behavior) или ошибочным, в то время как с параметрическим полиморфизмом тип не играет никакой роли (`fringe` без разницы, какой тип у элементов дерева). В Haskell специальный полиморфизм (или перегрузка (overloading)) основан на *классах типов* (type classes).

Приступим с простого, но очень важного, примера: равенства. Существует огромное количество типов, для которых мы желаем определить равенство, но есть и те, для которых этого не нужно. Например, трудно представить себе сравнение функций, в то время как сравнение двух списков очень желательно[30]. Чтобы лучше понять это, рассмотрим определение функции `elem`, которая проверяет список на наличие в нём определённого элемента:

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

[Из соображений, описанных в разделе 3.1, мы описываем эту функцию в инфиксной форме. `==` и `||` являются, соответственно, оператором сравнения и логическим „или“.]

Как видно отсюда, тип `elem` „должен быть“ таким: `a->[a]->Bool`. Но это означает, что `==` имеет тип `a->a->Bool`, хотя мы только что согласились, что `==` не должен быть объявлен для всех типов.

Более того, даже если бы `==` был объявлен для всех типов, то сравнение двух списков всё равно сильно отличалось бы от сравнения двух чисел. Поэтому ожидается, что `==` будет

[30] Равенство, о котором мы говорим, является "равенством значений а не "равенством указателей как в случае с оператором == из Java. Равенство указателей не имеет "ссылочной прозрачности"(referential transparency), а значит не может присутствовать в чисто функциональном языке.

перегружен для различных типов.

Классы типов[31] служат для этих двух целей. Они позволяют нам указать, какие типы будут являться *экземплярами* (instances) каких классов, а также дают возможность нам описать перегруженные *операции* (operations), связанные с классом. Например, давайте опишем класс, содержащий оператор сравнения:

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

Здесь `Eq` - это имя класса, который мы определили, а `==` - это единственная операция класса. Можно озвучить это определение следующим образом: „тип `a` является экземпляром класса `Eq`, если существует такая (перегруженная) операция, как `==` определённого вида.“ (Обратите внимание на то, что `==` определена только для пар объектов одинакового типа.)

Запись `Eq a` означает, что `a` должен являться экземпляром класса `Eq`. Это означает, что `Eq a` не является типовыражением, а есть некое ограничение, называемое *контекстом* (contex). После контекста идёт типовыражение. Например, предыдущее выражение присваивает (assign) следующий тип к `==`:

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

Это можно озвучить так: „Для каждого типа `a`, который является экземпляром класса `Eq`, операция `==` имеет тип `a->a->Bool`“. Этот тип и использовался в примере с `elem`. Контекст налагает ограничение, так что `elem` будет иметь следующий основной тип:

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

Это звучит так: „Для любого типа `a`, который является экземпляром класса `Eq`, функция `elem` имеет тип `a->[a]->Bool`“.Это то, что нам и нужно - мы утверждаем, что `elem` определена не для всех типов, а лишь для тех, которые имеют оператор сравнения.

Но как указать, какие типы являются экземплярами какого класса? Для этого существуют *объявления экземпляров* (instance declarations). Вот пример:

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

Определение `==` называют просто *методом* (method). Функция `integerEq`, по видимому, является функцией, которая сравнивает целые числа на равенство между собой[32], но вообще возможно любое выражение в правой части (как для обычных функций). Объявление говорит: „Тип `Integer` является экземпляром класса `Eq`, а вот определение метода, соответствующего `==`“. Благодаря этому определению, мы можем сравнивать целые числа на равенство с помощью `==`. Точно так же:

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

[31] Далее просто классы (не путать с классами ООП/ООД) - Прим.пер.

[32] Это не является правильным словосочетанием, но оно хорошо прижилось в мире программиования - Прим.пер.

Что позволяет нам сравнивать и действительные числа с плавающей запятой, используя ==.

Рекурсивные типы, такие как `Tree`, тоже можно использовать:
`instance (Eq a) => Eq (Tree a) where`

```
Leaf a          == Leaf b          = a == b
(Branch l1 r1)  == (Branch l2 r2)  = (l1==l2)  &&  (r1==r2)
_               == _               = False
```

Обратите внимание на контекст `Eq a` в первой строке - это необходимо, так как элементы листьев (тип - `a`) сравниваются (во второй строке). Дополнительное ограничение указывает, что мы можем сравнивать деревья из элементов типа `a`, если мы можем сравнивать между собой сами элементы типа `a`. Если бы мы опустили контекст из объявления экземпляра, то возникла бы ошибка типизации.

Haskell Report , а особенно прелюдия включает огромное количество примеров классов. Например, класс `Eq` определён немного по-другому:

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

Это пример класса с двумя операциями. Он также знакомит читателя с *методами по-умолчанию* (default methods), в данном случае `/=`. Если метод для конкретной операции опущен в объявлении экземпляра, то используется метод по-умолчанию, объявленный в объявлении класса (если доступен). Для примера, три экземпляра `Eq`, объявленные выше, будут отлично работать с вышеприведённым определением класса, в том числе и операция неравенства `/=`, которая просто является логическим отрицанием равенства.

Haskell поддерживает и *расширение классов* (class extension). Допустим, нам нужно объявить класс `Ord`, который *наследует* (inherits) все операции класса `Eq` и имеет дополнительный набор операций сравнения и функции нахождения минимума и максимума:

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

Обратите внимание на контекст в декларации класса. `Eq` является *суперклассом* (superclass) класса `Ord` (соответственно, `Ord` является *подклассом* (subclass) класса `Eq`), и любой тип, являющийся экземпляром `Ord`, должен быть экземпляром `Eq`. (В следующем разделе мы дадим более полное определение `Ord`, взятое из прелюдии.)

Одним из плюсов такого подхода является более короткий контекст: типовыражение для функции, которая требует операции из `Eq` и `Ord` может использовать лишь контекст `(Ord a)`, а не `(Eq a, Ord a)`, так как `Ord` „подразумевает“ `Eq`. Важнее то, что методы для операций подкласса могут быть уверены в существовании методов для операций суперкласса. Например, в прелюдии для `Ord` определён метод по-умолчанию для (<):

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

В качестве примера использования `Ord` посмотрим на основной тип функции `quicksort` из раздела 2.4.1:

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

Другими словами, `quicksort` оперирует только со списками из элементов, которые могут быть упорядочены.

В Haskell также разрешено *множественное наследование* (multiple inheritance), т.е. класс может иметь несколько суперклассов. Вот пример:

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

Такой класс наследует операции из `Eq` и `Show`.

Методы класса рассматриваются как объявления высшего уровня. Они находятся в одном пространстве имён с обычными переменными. Поэтому одно имя не может принадлежать методу класса одновременно с переменной или другим методом другого класса.

В `data` объявлениях разрешены контексты. См. §4.2.1.

Методы класса могут иметь дополнительные ограничения, наложенные на любую переменную, кроме той, что описывает данный класс. Например, в следующем классе:

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

метод `m` требует, чтобы тип `b` относился к классу `Show`. Тем не менее, метод `m` не может наложить никаких других ограничений на тип `a`. Это должно быть сделано в контексте объявления класса.

Мы рассмотрели „первоклассные“ типы. Конструктор типа `Tree` всегда был представлен нам с аргументом, например так: `Tree Integer` (дерево с `Integer` элементами) или `Tree a` (семейство деревьев с элементами типа `a`). Но сам по себе `Tree` является лишь конструктором типа, а значит он принимает тип в качестве аргумента и возвращает тип в качестве результата. В Haskell не существует таких значений, но такие „типы высшего порядка“ (higher-order types) могут быть использованы в объявлениях класса.

Начнём с примера класса `Functor` (взят из прелюдии):

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

Функция `fmap` есть обобщение функции `map`. Обратите внимание на то, что переменная типа применяется к другим типам в `f a` и `f b`. Поэтому, можно ожидать, что она будет привязана к типу подобному, например, `Tree`, который может быть задан аргументу. Экземпляр `Functor` для типа `Tree` будет выглядеть так:

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

Это объявление экземпляра объявляет, что `Tree`, а не `Tree a`, является экземпляром `Functor`. Такая возможность очень полезна, а в примере использовалась для описания типов „общих“ (generic) „контейнеров“, позволяющих функциям (таким как `fmap`) работать в одном стиле с любыми деревьями, списками и другими типами.

[Аппликация типов записывается точно так же, как и аппликация функций. Тип `T a b` разбирается как `(T a) b`. Типы, такие как кортежи, могут быть записаны другим способом, который вновь напоминает нам о Хаскеле Карри . Для функций (->) является конструктором типов; тип `f -> g` эквивалентен `(->) f g`. Точно также, типы `[a]` и `[] a` эквивалентны. Для кортежей `(,)`, `(,,)`... являются конструкторами типа (как и конструктором данных).]

Как известно, система типов находит ошибки типизации в выражениях. А что же с ошибками в типовыражениях? Выражение `(+)1 2 3` приводит к ошибке, так как `(+)` принимает всего два аргумента. Точно так же, тип `Tree Int Int` должен приводить к ошибке, так как `Tree` имеет всего один аргумент. Так как Haskell обнаруживает ошибочные типовыражения? Ответ кроется во второй системе типизации, которая проверяет типы на корректность! Каждый тип имеет соответствующий *вид* (kind), от которого и зависит правильность.

Типовыражения подразделяются на различные *виды*, которые могут принимать одну из двух следующих форм:

- Символ $*$ означает, что тип связан с конкретными объектами данных. Т.е. если значение v имеет тип t, то вид v должен быть $*$.
- Если k_1 и k_2 являются видами, то $k_1 \rightarrow k_2$ это вид типов, который получает вид k_1 и возвращает вид k_2.

Конструктор типов `Tree` имеет вид $* \rightarrow *$; тип `Tree Int` имеет вид $*$. Все члены `Functor` должны иметь вид $* \rightarrow *$; произойдёт ошибка, если будет дано следующее объявление:
`instance Functor Integer where ...` т.к. `Integer` имеет вид $*$.

Виды отсутствуют в коде программы. Компилятор автоматически (без каких-либо объявлений) выводит виды перед проверкой типов. Виды скрываются на заднем плане до тех пор, пока они не приводят к ошибкам. Виды просты настолько, что обычно компилятор может вывести достаточно полноценное сообщение об ошибке, как только она возникла. Подробности, связанные с видами, см. в §4.1.1 и §4.6.

Другие взгляды. Перед тем, как продолжить изучение классов, стоит рассмотреть два других взгляда на классы типов Haskell . Первый из них пришел из мира объектно-ориентированного программирования (ООП). В следующем утверждении об основе ООП просто подставьте вместо *классов* - *классы типов*, а вместо *объектов* - *типы*:

> „*Классы* объединяют общие операции. Конкретный *объект* является экземпляром класса и имеет методы, отвечающие его операциям. *Классы* могут иметь иерархию, формируя связи между супер*классами* и под*классами*, разрешая наследование операций/методов. С операцией может быть связан метод по-умолчанию.“

Должно быть понятно, что в отличии от ООП, типы не есть объекты, и не существует понятия внутреннего изменяемого состояния (internal mutable state) типов или объектов. Преимущество Haskell по сравнению с ООП языками заключается в том, что методы в Haskell полностью типобезопасны: любая попытка применения метода к значению, тип которого не

имеет нужного класса, приведёт к ошибке времени компиляции, а не к ошибке времени выполнения. Другими словами, методы не „ищутся“ во время выполнения, а просто передаются как функции высшего порядка.

Другой взгляд заключается в том, что он рассматривает связь между параметрическим и специальным полиморфизмом. Мы показали, как полезен параметрический полиморфизм в определении семейств типов с помощью квантификации. Иногда, тем не менее, не нужен столь „широкий“ подход - необходимо определить лишь узкое множество типов, таких как в случае со сравнениями. Классы типов можно рассматривать как способ осуществить это. Поэтому можно считать параметрический полиморфизм (также)видом перегрузки! Но перегрузка происходит неявно на всём множестве типов, а не на ограниченном (т.е. не только на определённом классе типов).

Сравнение с другими языками. Классы, используемые в Haskell очень походят на те, что используются в ОО-языках (C++ и Java, например). Тем не менее, существуют и значительные отличия:

- Haskell разделяет определение типа и определение методов, связанных с этим типом. Класс в C++ или Java обычно сразу же определяет и структуру данных (данные-члены), и функции, связанные со структурой (методы)[33]. В Haskell эти определения разграничены.
- Методы классов, определяемые в Haskell походят на виртуальные функции C++. Каждый экземпляр класса имеет своё собственное определение каждого метода; функции по-умолчанию соответствуют виртуальной функции в базовом классе[34].
- Классы в Haskell немного походят на интерфейсы в Java[35]. Как и определение интерфейса, объявление класса в Haskell определяет протокол (контракт) по взаимодействию с объектом, а не сам объект.
- Haskell не поддерживает стиль перегрузки C++, при котором функции с различными типами имеют одно имя.
- Тип объекта в Haskell нельзя неявно приводить; не существует единого базового класса (вроде `Object`).
- Компиляторы C++ и Java формируют вспомогательные структуры (вроде виртуальной таблицы), которые существуют и во время выполнения. В Haskell таковая информация существует лишь на логическом уровне, а не на физическом.
- В Haskell не существует модификаторов доступа (как то `public` или `private`). Для этого существует система модулей.

7 Вновь Типы

Давайте изучим более сложные аспекты системы типов.

[33] Имеется в виду, что в C++, несмотря на возможность раздельного написания для метода объявления (декларации) и определения (дефиниции),класс описывает как правило одновременно и свои поля (данные-члены), и методы (функции-члены). В некоторых случаях в C++ возможны исключения, когда члены класса описаны отдельно, но в Haskell это всегда должно быть так- Прим.пер.

[34] Конечно же подразумевается неперегруженная или чистая виртуальная функция - Прим.пер.

[35] Или на классы с чистыми виртуальными функциями в C++. Авторы, видимо, посчитали отсутствие `implements` за недостаток C++ - Прим.пер.

7.1 Newtype-объявление

Обычной практикой является объявление нового типа, который идентичен уже существующему. В Haskell для этого служит `newtype`-объявление (newtype-declaration). Например, натуральные числа можно представить с помощью `Integer`:

```
newtype Natural = MakeNatural Integer
```

Это объявление создаёт полностью новый тип `Natural`, единственный конструктор которого содержит один `Integer`. Конструктор `MakeNatural` конвертирует натуральные числа в целые (и наоборот):

```
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 заносит `Natural` в класс `Num`:
Без этого объявления `Natural` не будет являться экземпляром `Num`. Объявление экземпляров

```
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)
```

для старых типов не переходит к новым. Т.е. мы ввели этот тип для того, чтобы образовать ещё один экземпляр класса `Num`. Это было бы невозможно, если бы `Natural` был бы объявлен как синоним типа `Integer`.

Всё сработало с помощью `data` объявления, а не `newtype` объявления. Тем не менее, `data` объявление добавляет немного накладных расходов (extra overhead)(возникающих из-за ленивости), а `newtype` избегает их. См. раздел §4.2.3 Haskell Report , чтобы лучше разобраться в `newtype, data, type` объявлениях.

[Если не считать само название, `newtype` объявления имеют такой же синтаксис, как и `data` объявления с одним конструктором, содержащим единственное поле (field). Это очень удобно, так как типы, созданные с помощью `newtype`, практически идентичны созданным с помощью `data` объявлений.]

7.2 Метки и поля

К полям в Haskell можно обращаться либо по их позиции, либо используя *метки полей* (field labels). Представьте следующую структуру, представляющую точку на двумерной плоскости:

```
code Point = Pt Float Float
```

Два `Point` являются первым и вторым аргументами для конструктора `Pt`. Функции наподобии таких:

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

могут быть использованы для доступа к первому компоненту, но для больших структур становится весьма проблематично писать такие функции.

Поэтому конструкторы в `data` объявлениях могут быть объявлены с использованием связанных *имен полей* (field names), заключенных в фигурные скобки. Такие имена „ссылаются“ на компоненты конструктора по имени, а не по позиции. Вот другой способ определения `Point`:

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

Этот тип данных абсолютно идентичен предыдущему определению `Point`. Конструктор `Pt` точно такой же. Тем не менее, последнее объявление определяет ещё и два имени: `pointx, pointy`. Такие имена могут быть использованы в качестве *функции-селектора* (selector function) для доступа к компонентам. В этом примере имеется два селектора:

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

Приведём функцию, использующую их:

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

Метки полей могут быть использованы и для создания новых значений. Выражение `Pt { pointx=1, pointy=2 }` идентично `Pt 1 2`. Использование меток при объявлении конструкторов данных не запрещает позиционный доступ: доступны и `Pt { pointx=1, pointy=2 }`, и `Pt 1 2`. При конструировании значений с помощью имён полей некоторые имена могут быть опущены - в этом случае они являются неопределёнными.

Сопоставление с образцом с помощью имён полей использует такой же синтаксис:

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

Функция обновления (update function) использует значения полей из исходной структуры для того, чтобы создать компоненты новой. Если `p` является `Point`, то `p { pointx=2 }` - это точка с таким же `pointy` как у `p`, но с `pointx` равным 2. Это не деструктивное обновление (destructive update): функция обновления создаёт копию объекта, заполняя поля значениями.

[Скобки, использованные с метками полей, приведены для удобства: Haskell позволяет опускать скобки на основании *правил разметки* (о которых шла речь в §4.6). Тем не менее, скобки используемые с *именами* полей должны явно присутствовать.]

Имена полей не ограничены типами с единственным конструктором (такие типы обычно называются 'записями' (records)). Операции селекции или обновления (с использованием имён полей) у типов с многочисленными конструкторами могут приводить к ошибке времени

выполнения. Это очень походит на поведение функции `head`, вызванную с пустым списком в качестве аргумента.

Метки полей находятся в пространстве имён высшего уровня, а значит делят его с переменными и методами классов. Метки полей не могут быть использованы более чем в одном типе данных. Тем не менее, одно имя поля может быть использовано множество раз в конструкторах одного типа данных, при условии, что во всех конструкторах оно имеет одинаковый тип:

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

имя поля `f` используется в двух конструкторах `T`. Если `x` имеет тип `T`, то `x { f=5 }` сработает для значений, сконструированных любым конструктором `T`.

Имена полей не изменяют природу алгебраических типов; они являются просто специальным синтаксисом, который облегчает доступ к компонентам структур данных. Они помогают легче использовать конструкторы с многочисленными компонентами, так как поля могут быть добавлены или удалены без изменения каждой ссылки на конструктор. Детали см. в разделе §4.2.1.

7.3 Строгие конструкторы данных

Структуры данных в Haskell обычно являются *ленивыми*: их компоненты не вычисляются до тех пор, пока в этом не будет надобности. Это разрешает структуры с элементами, которые в момент вычисления могут выдать ошибку или не смогут завершить процесс вычислений (войдут в бесконечный цикл). Ленивые структуры повышают выразительность языка и являются базовым аспектом стиля программирования Haskell .

Каждое поле ленивого объекта данных „обёрнуто“ в структуру, которую обычно называют переходником (thunk). Переходник „скрывает“ вычисления. Он не используется, пока не требуется значение. Переходники, содержащие ошибку ($\perp$), не влияют на другие элементы структуры данных. Например, кортеж `('a', ⊥)` является абсолютно корректным значением. Значение `'a'` может быть использовано без затрагивания других компонентов кортежа. Напротив, большинство языков программирования являются *строгими*, а не ленивыми: все компоненты структур „приводятся“ к значениям перед помещением их в память.

Существует различные накладные расходы, связанные с переходниками: они требуют времени на создание и вычисление, они занимают место в хипе (heap), они заставляют сборщик мусора (garbage collector) учитывать дополнительные структуры, нужные для переходников. Для того, чтобы избежать этих накладных расходов, существует специальный *флаг строгости* (strictness flag) в объявлениях `data`, позволяющий вычислять выбранные поля контсруктора немедленно. Этот флаг выборочно подавляет ленивость. Поле, отмеченное восклицательным знаком (!) в `data` объявлении, вычисляется когда структура создаётся, а не откладывается на потом.

Бывает полезно применить флаг строгости в нескольких случаях:

- Если компоненты структуры должны быть вычислены на определённом этапе выполнения программы.

- Если комоненты структуры легко вычисляются и не приводят к ошибкам.
- Типы, в которых некоторые значения не требуются (частично определённые типы).

Для примера, библиотека для работы с комплексными числами определяет тип `Complex` следующим образом:

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

[Обратите внимание на инфиксную запись конструктора `:+`.] Такое определение указывает, что два компонента - действительная и мнимая части комплексного числа - являются строгими. Это более компактное представление комплексного числа, но это приводит к невозможности создания частично неопределённых структур (например, `1 :+` $\perp$ приводит к $\perp$). Так как для комплексных чисел в этом нет особой надобности, то использование флагов строгости полностью оправданно.

Флаги строгости могут быть использованы для нахождения утечек памяти (memory leaks) - структур, оставленных сборщиком мусора, но более ненужных[36].

Флаг строгости, !, может использоваться только в `data` объявлениях. Он не может использоваться в других сигнатурах типов или в каких-либо определениях типов. Не существует равноценного способа обозначить аргументы функций на наличие строгости. Этого эффекта можно добиться с помощью функций `seq` или `!$`. См. §4.2.1.

Трудно четко определить, когда должен использоваться флаг строгости. Будьте очень осторожны: ленивые вычисления являются одним из основных свойств Haskell , а использование флага строгости может привести к трудноуловимым проблемам и бесконечным циклам.

8 Ввод/Вывод

Система I/O в Haskell является чисто функциональной, но имеет все возможности обычных языков программирования. В императивных языках программы описываются *действиями* (actions), которые проверяют и изменяют текущее состояние мира. Обычные действия включают чтение и запись глобальных переменных, запись в файл, чтение с клавиатуры, создание окон. Такие действия есть и в Haskell , но они чётко отделены от чисто функционального ядра языка.

Система I/O Haskell построена на основе какой-то страшно непонятной математики, называемой *монадами* (monads). Тем не менее, изучение теории вовсе не обязательно для программирования на Haskell . Монады являются концептуальными структурами, которые подходят для I/O. Для того, чтобы работать с I/O не нужно ничего, кроме знания теории групп (<возможно, исправить>) для простых арифметических вычислений. Полноценное описание монад находится в разделе 9.

Операции с монадами, на которых построена система I/O, используются и для других целей; вернёмся к этому позже. В данный момент постараемся избегать слова „монада“, а просто рассмотрим использование системы I/O. Лучше всего рассматривать монады как абстрактные структуры данных.

[36] Подробнее об этом <???> - Прим.пер.

В Haskell действия скорее описываются, нежели вызываются (исполняются). Вычисление определения действия на самом деле не вызывает этого действия. Действие происходит в другом месте, а не в выражении, о котором мы здесь говорим.

Действия либо являются атомарными (как и базовые системные действия), либо являются последовательной композицией других действий. Последнее очень похоже на последовательное объединение выражений с помощью `;` в других языках. Монады служат „клеем“, который связывает действия в программе.

8.1 Базовые операции I/O

Любое I/O действие возвращает значение. В системе типов такое значение завязано с типом `IO`, что отличает действия от других значений. Например, типом функции `getChar` является:

```
getChar :: IO Char
```

`IO Char` показывает, что `getChar`, когда вызывается, выполняет какие-то действия, которые возвращают символ. Действия, которые возвращают ненужные значения, используют тип `()`. Например, функция `putChar`:

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

принимает символ, но не возвращает ничего полезного. Такой тип по сути является эквивалентом `void` других языков.

Действия упорядочены (sequenced) с помощью оператора, который имеет загадочное имя: `>>=` (читается как „привязать“). Вопреки прямому использованию этого оператора, мы используем синтаксический сахар в виде `do`. Такое сокращение можно тривиально развернуть в `>>=`, как описано в §3.14.

Ключевое слово `do` начинает последовательность утверждений (statements), которые выполняются по порядку. Утверждениями называют либо действия, т.е. шаблон, связанный с результатом действия с помощью `<-`, либо множество локальных определений с использованием `let`. `do` использует разметку точно так же, как и `let` или `where`, так что мы можем опустить скобки и точки с запятой. Вот простая программа, которая считывает символ с клавиатуры, а затем выводит символ на консоль: Обратите внимание на использование имени

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

`main`: оно является входной точкой программ на Haskell (как и `main` в C) и должно иметь тип `IO`, обычно `IO ()`. (имя `main` является особым только в модуле `Main`; вскоре мы обсудим и модули.) Эта программа последовательно выполняет два действия: сначала она считывает символ, привязывая результат к переменной `c`, а затем выводит его на консоль. В отличии от выражений `let`, где переменные видимы во всех определениях, переменные объявленные `<-` видимы лишь в следующих утверждениях.

Остаётся непонятным одно: как можно вернуть результат последовательности действий? Например, представьте функцию `ready`, которая считывает символ и возвращает `True`, если это был `'y'`: Это не сработает, так как второе утверждение в `do` - это просто значение, а не

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

действие. Нужно каким-то образом преобразовать этот буль в действие, которое ничего не делает, но возвращает буль в качестве результата. Функция `return` делает именно это:

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

На этой функции заканчивается набор „упорядоченных“ примитивов. Последняя строка `ready` должна быть изменена на

```
return (c == 'y')
```

Теперь мы готовы приступить к изучению функций посложнее. Вот функция `getLine`:

```
getLine     :: IO String
getLine  = do c <- getChar
              if c == '\n'
                                then return
                                else do l <- getLine
                                        return (c:l)
```

Обратите внимание на второй `do`. Каждый `do` начинает новую цепочку утверждений. Каждая новая конструкция, вроде `if` должна использовать свой `do`, чтобы начать последовательность действий.

Функция `return` вводит обычное значение (например, буль) в мир ввода/вывода. А как быть, если нужно в точности до наоборот? Можно ли в обычном выражении использовать какие-то операции I/O? Например, можем ли мы написать просто `c + print y` и ожидать от этого вывода `y`, как только произойдёт вычисление этого выражения? Ответ: нет. *Не*возможно нырнуть в мир императива прямо из центра функционального языка. Любое значение „зараженное“ императивом, должно иметь соответствующий „ярлык“. Функции вроде:

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

вообще не могут делать никаких операций I/O, так как `IO` не указан в возвращаемом типе. Это обычно очень нервирует программиста, который привык расставлять `print` и `read` по всей программе. На самом деле, существуют особые небезопасные функции, которые призваны бороться с этим. Оставим их для программистов уровнем повыше. Пакеты для отладки, например `Trace`, обычно позволяют использовать такие „запрещенные функции“ в безопасном виде.

8.2 Программирование с действиями

I/O действия являются обыкновенными значениями в Haskell : их можно передавать функциям, помещать в структуры и использовать как любое другое значение. Представьте следующий список действий:

Такой список не вызывает никаких действий - он их просто содержит. Чтобы объеденить эти действия в одно, нужно использовать функцию, подобную `sequence2`:

Можно сделать проще с помощью раскрытия `do x;y` в `x » y` (см. раздел 9.1). Этот пример рекурсии отлично запечатлен в функции `foldr` (см. определение `foldr` в прелюдии). Поэтому лучше определить `sequence2` так:

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

sequence2          :: [IO ()] -> IO ()
sequence2 []           = return ()
sequence2 (a:as)         = do a
                            sequence as


          sequence2 :: [IO ()] -> IO ()
        sequence2 = foldr (») (return ())
```

Оператор `do` обычно очень полезен, но в данном случае лучше использовать монадный оператор (monadic operator) `»`. Понимание работы оператора, на котором основан `do`, очень важно для Haskell программиста.

Функция `sequnce2` может быть использована для конструирования `putStr` из `putChar`:

```
          putStr :: String -> IO ()
      putStr s = sequence2 _ (map putChar s)
```

Отличие Haskell от императивных языков видно и по функции `putStr`. В императивных языках программирования отображение (mapping) императивной версии `putChar` на строку выведет её на экран. В Haskell функция `map` не выполняет никаких действий. Она лишь создаёт список действий, по одному на символ. Операция свёртки (folding)[37] в функции `sequence_` использует функцию `»` для объеденения всех действий в одно. Здесь абсолютно необходим `return ()`, так как функции `foldr` требуется завершающее действие (null action) в конце цепочки действий, которую она создаёт (особенно если в строке нет символов).

Прелюдия и библиотеки Haskell содержат множество функций, полезных для упорядочения действий I/O. Они обычно обобщены до различных монад; любая функция с контекстом `Monad m =>` работает с типом `IO`.

8.3 Исключения

До сих пор мы избегали обсуждения исключений (exceptions) в операциях I/O. Что произойдёт, если `getChar` встретит конец файла[38]. Для обработки исключительных ситуаций, таких как „файл не найден“, из монад I/O используется схожий с ML механизм обработки. Никакого специального синтаксиса! Обработка исключений является частью объявления операций упорядочения I/O.

Для ошибок существует специальный тип данных: `IOError`. Этот тип обозначает все возможные исключения, которые могут произойти в монадах I/O. Он является абстрактным: у него нет доступных пользователю конструкторов. Различные предикаты позволяют проверять `IOError` значения. Например, функция

[37] Не путать со свёрткой функций (function convolution) - Прим.пер.

[38] Мы используем термин *ошибка* для $\perp$: незавершение или несопоставление образца. *Исключения*, с другой стороны, могут быть отловлены и обработаны с помощью монад

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

определяет, вызвана ли ошибка окончанием файла. Так как `IOError` является абстрактным, то новые виды ошибок могут быть добавлены без кардинального изменения типа данных. Функция `isEOFError` определена в отдельной библиотеке `IO`, которую нужно явно импортировать в программу.

Обработчик исключения (exception handler) имеет тип `IOError -> IO a`. Функция `catch` связывает обработчик с действием или множеством действий:

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

Аргументами для `catch` являются действие и обработчик. Если действие проходит успешно, его результат возвращается без вызова обработчика. Если же происходит ошибка, она передаётся обработчику в качестве значения типа `IOError`, а затем вызывается действие, ассоциированное с обработчиком. Для примера, следующая версия `getChar` возвращает новую строку, если происходит ошибка:

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

Такой подход является очень грубым, так как обрабатывает все ошибки одинаково. Если требуется определить конец файла, то нужно использовать предикат:

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

Функция `ioError`, используемая здесь, выкидывает исключение к следующему обработчику ошибок. `ioError` имеет тип:

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

Она похожа на `return`, только она передаёт управление обработчику ошибок вместо того, чтобы перейти к следующему действию I/O. Разрешаются и вложенные вызовы `catch`, а значит вложенные обработчики ошибок.

Можно переопределить `getLine` с помощью `getChar'`, чтобы продемонстрировать использование вложенных обработчиков ошибок:

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

Вложенные обработчики ошибок позволяют `getChar'` обрабатывать конец файла отдельно от всех других ошибок, которые вернут строку, начинающуюся с `"Error: "`.

Для удобства, Haskell имеет обработчики ошибок на самом верхнем уровне, которые выводят сообщение и завершают выполнение программы.

8.4 Файлы, Каналы, Обработчики

Помимо монад I/O и механизма обработки исключений, средства I/O схожи с таковыми в других языках. Многие из этих функций находятся в библиотеке `IO`, а не в прелюдии, что требует явного импортирования этой библиотеки (импорт модулей обсуждается в главе 11). Большинство из этих функций рассматривается в „Library Report“.

При открытии файла создаётся *дескриптор* (handle), который имеет тип `Handle`. Закрытие дескриптора заканчивает работу с файлом:

```
type FilePath        = String - path names in the file system
openFile                :: FilePath -> IOMode -> IO Handle
hClose                             :: Handle -> IO ()
data IOMode    = ReadyMode | WriteMode | AppendMode | ReadWriteMode
```

Дескрипторы могут быть связаны с *каналами* (channels): порты взаимодействия, которые не связаны напрямую с файлами. Определено несколько дескрипторов, включая `stdin` (стандартный ввод), `stdout` (стандартный вывод),`stderr` (стандартный канал ошибок). Операции I/O символьного уровня включают `hGetChar` и `hPutChar`, которые принимают дескриптор в качестве аргумента. Функцию `getChar` можно переопределить так:

```
getChar = hGetChar stdin
```

Существует возможность вернуть полное содержание файла или канала в виде единой строки:

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

Кажется, что `getContens` должна сразу же прочитать весь файл или канал, что приведёт к чрезмерному использованию памяти и ухудшению производительности. Тем не менее, это вовсе не так. Дело в том, что `getContens` возвращает „ленивый“ (т.е. нестрогий) список символов (вспомните, что строки - это просто списки символов), элементы которого считываются „по надобности“, точно так же, как и с любым другим списком. Можно представить, что такое считываение „по необходимости“ берёт по одному символу за раз.

В этом примере мы копируем 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."
```

С помощью ленивой функции `getContents` всё содержание файла не должно считываться сразу. Если `hPutStr` использует буферизацию, то в любой момент времени всего один блок ввода должен быть в памяти. Входной файл неявно закрывается, когда считан последний символ.

8.5 Haskell и императивное программирование

При программировании I/O возникант проблема: такой стиль весьма похож на обычное императивное программирование. Например, вот функция `getLine`:

```
getAndOpenFile    ::    String  -> IOMode   -> IO Handle
getAndOpenFile  prompt  mode    =
                  do    putStr  prompt
                        name    <- getLine
                        catch   (openFile   name mode)
                                \_ -> do    putStrLn ("Невозможно открыть" ++ name ++) "\n")
                                            getAndOpenFile prompt mode

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

Эта функция очень похожа на императивную версию (написанную на псевдо-коде):

```
function getLine() {
 c := getChar();
 if c == '\n'          then return
                       else { l := getLine();
                              return c:l }}
```

Так значит Haskell снова изобрёл императивный велосипед?

В некотором роде это так. Монады I/O содержат в себе небольшой императивный язык прямо внутри Haskell , так что эти компоненты I/O могут выглядеть как обычные императивные вставки. Но существует одно кардинальное различие: не существует никакой специальной семантикики, с которой пользователю пришлось бы разбираться. Императивное ощущение вовсе не заглушается функциональным подходом Haskell . Опытный функциональный программист должен иметь возможность минимизировать императивную составляющую программы, используя монады в редких случаях высокоуровнего упорядочения. Монады чётко отделяют функциональные компоненты от императивных. В противоположность этому императивные языки с функциональными возможностями не имеют такой (столь чёткой) границы между ними.

9 Стандартные классы Haskell

В этой главе мы рассмотрим стандартные классы типов Haskell . Мы изучим лишь самые интересные методы. Haskell Report содержит более детальное описание. Некоторые стандартные классы являются частями стандартных библиотек Haskell . Они рассмотрены в „Library Report“.

9.1 Классы Eq и Ord

Классы `Eq` и `Ord`[39] уже обсуждались. В прелюдии определение `Ord` немного сложнее того, которое мы давали до этого. В частности, посмотрите на метод `compare`:

[39] Соответственно их названия есть сокращения от „равенства“ (equality) и „упорядоченности“ (order) - Прим.пер.

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

Метод `compare` играет большую роль для определения всех остальных методов этого класса и является лучшим способом создать экземпляры `Ord`.

9.2 Класс Enumeration

Класс `Enum` имеет набор операций, которые лежат в основе синтаксического сахара для арифметических последовательностей; например, выражение (арифметической последовательности) `[1,3..]` означает `enumFromTen 1 3` (см. § 3.10). Теперь видно, что такие выражения могут быть использованы для генерации списков любых типов, которые являются экземплярами класса `Enum`. К этому относится не только большинство численных типов, но и `Char`, а значит `['a'..'z']` означает список из всех строчных букв в алфавитном порядке. Более того, для пользовательских типов, таких как `Color`, можно легко задать объявление экземпляра `Enum`:

```
[Red .. Violet] ⇒ [Red, Green, Blue, Indigo, Violet]
```

Обратите внимание на то, что такая последовательность является *арифметической*, то есть приращение значений между элементами есть константа (даже если значения не являются числами). Большинство типов, порождённых от `Enum`, могут быть легко отображены в целые; функции `fromEnum` и `toEnum` служат для преобразования `Int` в перечисляемый тип и наоборот.

9.3 Классы Read и Show

Экземпляры класса `Show` являются типами, которые могут быть преобразованы в строки из символов (обычно для I/O). Простейшей функцией класса является `show`:

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

Судя по этому определению, `show` принимает любое значение подходящего типа и возвращает его в виде строки (списка символов). `show (2+2)` означает „4“. Это достаточно хорошо, но обычно мы хотим создавать строки посложнее, которые содержат многочисленные значения:

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

Все эти конкатенации приводят к потере производительности. Представим функцию, которая отображает бинарные деревья (из раздела 2.2.1) в виде строк с помощью подходящих обозначений. Такое отображение деревьев должно правильно отображать вложенность, а также разделение левых и правых ветвей (предположим, тип элемента отображаем в качестве строки):

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

Так как `(++)` имеет линейную сложность, то `showTree` имеет квадратичную.
Чтобы повысить производительность (другимим словами, уменьшить алгоритмическую сложность до линейной), существует функция `shows`:

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

Она берёт отображаемое значение и строку, а возвращает строку с присоединённым к её голове строковым представлением числа. Второй аргумент служит в качестве накопителя строк, а значит можно определить `show` как `shows` с нулевым аккумулятором. Так и сделано по-умолчанию:

```
show x = shows x
```

Можно использовать `shows` для определения более эффективной версии `showTree`, которая тоже имеет строковый накопитель:

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

Это решает нашу проблему с эффективностью (`showsTree` имеет линейную сложность), но можно улучшить и само объявление. Для начала введём синоним для типа:

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

Это тип функции, которая возвращает строковое представление чего-либо,после чего идёт строка-накопитель. Далее, можно избежать множества аккумуляторов и скобок в правых частях длинных конструкций с помощью композиции функций:

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

Произошло кое-что интерснее простого рефакторинга (улучшения кода): мы перешли с *обвектного уровня* (object level) на *функциональный уровень* (functional level). Можно сказать, что `showsTree` отображает дерево в *функцию вывода (отображения)* (showing function). Функции вроде `('<' :)` или `("a string" ++)` являются примитивами для функций вывода. На их основе мы с помощью композиции создали функцию гораздо сложнее.

Мы умеем превращать деревья в строки, а теперь давайте попробуем наоборот. Основной идеей является разборщик (parser) для типа `a`, который принимает строку и возвращает список из пар - `(a, String)` [9]. Прелюдия уже имеет соответствующий синоним типа для таких функций:

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

Разборщик возвращает список, содержащий значение типа `a`, которое было взято из входной строки и остальную строку, которая осталась после разбора. Если не существует возможного разбора, то результатом будет являться пустой список, а если существует больше одного варианта разбора (неоднозначность), то результирующий список будет содержать несколько пар. Стандартная функция `reads` служит разборщиком для любого экземпляра `Read`:

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

```
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]
```

Можно использовать эту функцию для определения функции разбора для нашего строкового представления бинарных деревьев, которое производит `showsTree`. Для построения такого разборщика очень удобно использовать составления списков[40]:

Давайте немного остановимся и подумаем над этим определением. Если первым символом является `'<'`, то мы встретили ветвь, иначе - мы встретили лист. В первом случае оставшаяся часть строки (т.е. без `'<'`) называется `s`; она должна быть сопоставлена с `Branch l r`; оставшейся частью является `u`. Вот некоторые условности такого разбора:

1. Дерево `l` должно быть разобрано с начала строки `s`.
2. Оставшаяся строка () начинается с `'|'`. Назовём этот хвост строкой `t`.
3. Дерево `r` должно быть разобрано с начала строки `t`.
4. Строка, оставшаяся после разбора, начинается с `'>'`, а её хвостом является `u`.

Обратите внимание на удобство комбинированного использования сопоставления с образцом и составления списков: вид результирующего разбора задаётся главным составлением, два первых условия заданы первым генератором („`(l, '|':t)` берётся из списка возможных разборов `s`“), а остальные условия заданы вторым генератором.

Второе уравнение говорит о том, что для разбора представления листа мы разбираем представление дерева, а затем применяем к полученному значению конструктор `Leaf`.

Будем считать, что существуют экземпляры `Read` (и `Show`) для типа `Integer` (как и для многих других типов). Поэтому `reads` должна работать верно, т.е.:

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

Читатель должен убедиться, что функция ведёт себя верно со следующими примерами:

```
readsTree «1|<2|3»"   ⇒   [(Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3)), "")]
readsTree «1|2"       ⇒   []
```

У нашего разборщика есть несколько недостатков. Первый заключается в том, что он весьма груб - не обрабатывает пробелы перед элементами (или между элементами) представления дерева. Второй вызван тем, что мы обрабатываем специальные символы иначе, нежели листья и поддеревья. Такая нехватка единства ведёт к тому, что определение функции становится сложнее. Можно справиться с этими проблемами с помощью лексического анализатора, который имеется в прелюдии:

```
lex :: ReadS String
```

[40] Более элегантный подход к разбору заключается в использовании монад и комбинаторов разборщика (parser combinators). Они являются частью стандартной библиотеки для разбора, поставляемой с большинством систем Haskell

`lex` возвращает список (с одним элементом), содержащий пары из строк: первую лексему из входной строки и остаток. Лексические правила такие же, как и в программах, написанных на Haskell , включая комментарии, которые `lex` пропускает как и пробелы. Если входная строка пуста или содержит лишь пробелы и комментарии, `lex` возвращает `[("","")]`. Если входная строка не пуста, но не начинается с валидной лексемы, `lex` возвращает `[]`.

Теперь наш разборщик выглядит так:

```
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 мы должны использовать `readsTree` и `showsTree`, чтобы объявить `(Read a) => Tree a` экземпляром `Read`, а `(Show a) => Tree a` экземпляром `Show`. Это позволит нам использовать обобщённые функции из прелюдии для разбора и отображения деревьев. Более того, мы сможем автоматически разбирать и выводить многие другие типы, которые содержат деревья в качестве компонентов (например, `[Tree Integer]`). Как оказывается, `readsTree` и `readsTree` являются практически идеальными для того, чтобы стать методами `Show` и `Read`. Функции `showsPrec` и `readsPrec` являются методами `shows reads` с дополнительным параметром. Этим параметром является уровень предшествования, используемый для верной расстановки скобок в выражениях, содержащих инфиксные конструкторы. Для типов, таких как `Tree`, уровень предшестования может быть проигнорирован. Приведём объявления экземпляров для `Tree`:

```
instance Show a => Show (Tree a) where
       showPrec _ x = showsTree x

instance Read a => Read (Tree a) where
      readsPrec _ s = readsTree s
```

Существует ещё один способ объявления экземпляра `Show`:

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

Тем не менее, такая версия будет менее эффективна. Обратите внимание на то, что класс `Show` определяет методы по-умолчанию для `showsPrec` и `show`, что позволяет пользователю определить один из них в объявлении экземпляра. Так как эти методы взаимно рекурсивны, объявление экземпляра, которое не определяет ни одного из них, зациклится при вызове. Другие классы, например такие как `Num`, имеют такие же „взаимосвязанные методы по-умолчанию“.

Отошлём заинтересованного читателя к главе §D, в которой приведены детали классов `Read` и `Show`.

Можно протестировать экземпляры классов `Read` и `Show` с помощью аппликации `(read. show)` к деревьям, где `read` является специализацией `reads`:

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

Данная функция не сработает, если ввод содержит что-либо, кроме представления одного значения типа `a` (ну и, возможно, комментарии или пробелы).

9.4 Производные экземпляры (наследование экземпляров)

Вернёмся к экземпляру класса `Eq` для деревьев, который мы привели в разделе 5; его объявление весьма простое, но его достаточно трудно использовать, так как мы требуем наличия в листьях сравнимых элементов (т.е. типа `Eq`). Два листа равны, только если они содержат одинаковые элементы, а две ветви равны, только если их левые и правые поддеревья равны. Любые другие деревья не являются равными:

```
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 поступать так каждый раз, когда нам нужна операция сравнения на эквивалентность для нового типа; можно просто *автоматически наследовать* (derive automaticly) экземпляр класса `Eq`, если мы укажем так в объявлении `data`:

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

Такое объявление неявно создаёт экземпляр класса `Eq`, полностью идентичный тому, который приводился в разделе 5. Таким образом, с помощью ключевого слова `deriving` можно создать экземпляры классов `Ord, Enum, Ix, Read, Show`. [Можно использовать несколько имён классов, для чего они должны быть записаны в скобках и разделены запятыми.]

Производный экземпляр класса `Ord` для `Tree` является чуть более сложным, чем экземпляр для класса `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'
```

Конструкторы упорядочены таким образом, что приводит к *лексикографическому упорядочению* (lexicographic order). Напомним, что аргументы конструкторов сравниваются слева на право. Тип встроенного списка семантически эквивалентен обычному двуконструкторному типу. По сути, вот полное определение (псевдокод):

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

(Списки имеют экземпляры `Show` и `Read`, которые не являются производными.) Производные экземпляры `Eq` и `Ord` для списков являются стандартными. Символьные строки, как и списки символов, упорядочены так, как это определяет тип `Char`: строка меньшей длины идёт первой. Например, `"cat`$\ll$ `"catalog"`.

На практике, экземпляры `Eq` и `Ord` обычно являются производными, а не пользовательскими. Вообще, мы должны дважды подумать, прежде чем использовать свои определения,

так как нельзя нарушать алгебраические свойства сравнения и упорядоченности. Нетранзитивный предикат `(==)`, для примера, может быть просто разрушительным, вводя в заблуждение людей, читающих ваш код. Помимо этого он может нарушить правильность ручных или автоматических преобразований, которые основаны на том, что предикат `(==)` является олицетворением эквивалентности. Тем не менее иногда необходимо иметь собственные экземпляры `Eq` и `Ord`. Возможно, самым полезным примером является абстрактный тип данных, в котором одно абстрактное значение отображается во многие конкретные.

Перечисляемый тип может иметь производный экземпляр `Enum`. Порядок конструкторов имеет большое значение. Например:

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

Вот несколько примеров использования производных экземпляров этого типа:

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

Производные экземпляры `Read (Show)` доступны для любых типов, типы компонентов которых тоже имеют экземпляры `Read (Show)`. (Экземпляры для `Read (Show)` для большинства стандартных типов имеются в прелюдии. Некоторые типы, такие как `(->)` имеют экземпляр `Show`, но не `Read`.) Текстуальное представление, определённое производным экземпляром класса `Show` согласуется с константными выражениями типа Haskell . Например, если мы добавим `Show` и `Read` в `deriving` для типа `Day`, то получим:

```
show [Monday .. Wednesday] ⇒ „[Monday,Tuesday,Wednesday]„
```

10 Монады

Многие новички испытывают трудности с *монадами*. Монады достаточно часто встречаются в Haskell : система I/O построена на них. Существует специальный синтаксис (выражения `do`), а стандартная библиотека содержит модуль, полностью посвященный им. В этом разделе мы рассмотрим программирование с использованием монад более подробно.

Данный раздел имеет более глубокий подход к изучению. Мы рассматриваем не только язык, но и стараемся охватить большую картину: почему монады настолько важны, и как они используются. Не существует единственно верного пути объяснения, который бы подошел для каждого; дополнительную информацию можно найти на сайте `haskell.org`. Хорошим руководством по программированию с использованием монад можно считать работу Вэдлера (Wadler) *Monads for Functional Programming* [10].

10.1 Классы монад

Прелюдия содержит несколько классов для монад. Эти классы основаны на теории категорий, в то время как теория категорий задаёт имена для классов монад и операций, не обязательно знать математические основы для того, чтобы использовать эти классы и операции.

Монады имеют в своей основе полиморфный тип `IO`. Монада определена с помощью объявления экземпляра, который связывает тип с некоторыми или всеми монадными классами:

`Functor`,`Monad`,`MonadPlus`. Ни один из классов монад не может быть производным. В дополнение в `IO` прелюдия содержит два других типа, которые являются членами классов монад: списки ([]) и `Maybe`.

Для монадных операций математика имеет несколько *законов*[41]. Использование законов не ограничивается лишь монадами: Haskell имеет многие операции, которые используют, хотя бы и неявно, различные законы. Например, `x /= y` и `not (x == y)` должны являться идентичными для любых сравниваемых значений. Тем не менее, нет четкой гарантии, так как `==` и `/=` являются разными методами класса `Eq`, и нет способа убедиться, что они верно связаны между собой. Точно также некоторые законы для монад не навязаны Haskell , а просто должны работать для любых экземпляров классов монад. С помощью изучения этих законов мы можем лучше понять принципы работы с монадами и, самое главное, научимся их использовать.

Класс `Functor`, рассмотренный в разделе 5, имеет лишь одну операцию `fmap`. Функция отображения применяет операцию к объектам, содержащимся внутри контейнера (можно рассматривать полиморфные типы как контейнеры для значений других типов) и возвращает контейнер такого же вида. Эти же „законы“ работают и для `fmap` из `Functor`:

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

В частности, один из законов утверждает, что тип контейнера не изменяется функцией `fmap`, а также что содержимое контейнера не изменяется с помощью операции отображения.

Класс `Monad` определяет две базовые операции: `>>=` 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 привязки `>>` и `>>=` объединяют два монадных значения, а операция `return` заносит значение в монаду (контейнер). Сигнатура `>>=` помогает понять нам её работу: `ma >>= \ v -> mb` объединяет монадное значение `ma`, которое содержит значение типа `a`, и функцию, которая работает со значением `v` типа `a` и возвращает монадное значение `mb`. В результате `ma` и `mb` объединяются в монадное значение, содержащее `b`. Функция `>>` используется, если функция не нуждается в значении, которое производит первый монадный оператор.

Чёткое понимание операции привязки очень помогает для работы с монадами. Например, в монаде I/O `x >>= y` последовательно выполняет два действия, передавая результат первого на вход второго. Для других встроенных монад, списков и типа `Maybe` эти монадные операции можно рассматривать как передачу нуля или нескольких значений от одного вычисления к другому. Вскоре мы приведём соответствующие примеры.

[41] Здесь и далее приводится дословный перевод - Прим.пер.

Команда do является сокращением для последовательностей монадных операций. Следующие два правила хорошо объясняют принципы работы do:

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

Так как шаблон (образец) второго вида do является опровержимым, неудачное сопоставление вызовет операцию fail. Она может вызвать ошибку (что и происходит в монаде IO) или вернуть „нуль“ (как в монаде списка). Приведём „раскрытие“ многословнее:

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

где "s" является строкой, которая указывает на утверждение do. Она используется для сообщения об ошибке. Например, в монаде I/O действие вроде 'a' <- getChar вызовёт fail, если введённым символом не является 'a'. В свою очередь, это приведёт к завершению программы, так как монада I/O через fail вызывает error.

Приведём законы, применяющиеся для >>= 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, которые имеют *нулевой* элемент и операцию *плюс*:

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

Нулевой элемент отвечает следующим законам:
Для списков нулевое значение есть [], т.е. пустой список. Монада I/O не имеет нулевого элемента и не является членом этого класса.

Оператор mplus отвечает следующим законам:
Оператор mplus является обычной конкатенацией.

10.2 Встроенные монады

Итак, у нас есть монадные операции и законы, которые они должны соблюдать. Что же можно построить, имея этот набор? Мы видели I/O монады в действии, так что приступим к изучению двух других встроенных монад.

Для списков монадическое связывание означает объединение набора вычислений для каждого значения в списке. При использовании со списками >>= имеет следующую сигнатуру:

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

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

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

Т.е., имея список из элементов типа `a` и функцию, которая отображает `a` в список из элементов типа `b`, связывание применяет эту функцию для всех элементов `a`, а затем возвращает все сгенерированные элементы `b`, соединённые в список. Читатель должен быть знаком с таким типом операций: составления списков могут быть легко заменены монадными операциями, определёнными для списков. Приведём три вида выражений, которые означают одно и то 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) >=
```

Это определение зависит от определения `fail` этой монады в качестве пустого списка. Каждый `<-` генерирует набор значений, который передаётся далее к остатку монадных вычислений. Поэтому `x <- [1,2,3]` использует остаток монадных вычислений три раза, по одному на каждый элемент списка. Возвращаемое выражение `(x,y)` будет вычислено для всех возможных комбинаций привязки, которое охватывает их. Монада списков может рассматриваться как описатель функций с многочисленными аргументами. Например, следующая функция:

превращает обычную функцию двух аргументов (f) в функцию многих аргументов (списка аргументов) и возвращает значение для любой возможной комбинации двух аргументов. Например:

Эта функция является специализированной версией `LiftM2` из библиотеки монад. Можно рассматривать её как перевод функции из внешней „стороны“ монады списка `f` во внутреннюю, где в вычислениях участвуют многочисленные аргументы.

Монада, объявленная для `Maybe` идентична монаде для списка: значение `Nothing` служит как [], а `Just x` как `[x]`.

10.3 Использование монад

Рассмотрение монадных операторов и связанных с ними законов вовсе не показало, для чего же монады действительно нужны. Во-первых, они служат основой для модульности (modularity). С помощью объявления монадной операции можно скрыть базовую функциональность таким путём, который позволяет новым возможностям быть прозрачно добавленным в монады. Работа Вэдлера (Wadler) *Monads for Functional Programming* [10] является прекрасным примером построения модульных программ на основе монад. Продолжим наше

```
( \ r -> case r of  True  -> return (x,y)
                    _     -> fail ""))

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

изучение с рассмотрения монады состояния (state monad), взятой оттуда, а затем построим монаду посложнее с похожим определением.

Вот определение монады состояния, построеной над типом `S`:

В этом примере определён новый тип `SM`, который неявно производит вычисления с типом `S`. Так, вычисление типа `SM t` определяет новое значение типа `t` и одновременно взаимодействует (чтение и запись) с состоянием типа `S`. Определение `SM` очень простое: оно состоит из функции, которая принимает состояние и возвращает два результата - возвращаемое значение (любого типа) и обновленное значение. Здесь нельзя использовать синоним типа, так как необходимо имя типа, такое как `SM`, которое может быть использовано в объявлении экземпляров. В таком случае объявление `newtype` часто используется вместо `data`.

Объявление экземпляра „объединяет“ монады: упорядочивает два вычисления и определение пустого вычисления. Упорядочение (оператор `>>=`) определяет вычисление (на что указывает конструктор `SM`), которое передаёт начальное состояние `s0` в `c1`, а затем передаёт возвращаемое значение `r` функции, которая возвращает второе вычисление `c2`. Окончательно состояние, возващаемое из `c1` передаётся в `c2`, и последним результатом является результат `c2`.

Определение `return` проще: он вообще не меняет состояния, а лишь служит для передачи значения в монаду.

В то время как `>>=` и `return` являются базовыми операциями упорядочения, нам нужны и другие *монадные примитивы* (monadic primitives). Монадными примитивами называют операции, которые работают внутри монад и являются как бы их „шестерёнками“, которые приводят монады в действие. Для примера, монада `IO` имеет оператор `putChar`, который является примитивом, так как работает с внутренним состоянием этой монады. Точно так же, наша монада состояния имеет два примитива: `readSM` и `updateSM`. Обратите внимание на то, что их определение зависит от внутренней структуры монады - изменение типа `SM` потребует изменения этих примитивов.

Определения `readSM` и `updateSM` очень просты: `readSM` достаёт состояние из монады, а `updateSM` позволяет пользователю изменить его. (Можно было бы использовать `writeSM` в качестве примитива, но обновление является более естественной операцией при работе с состояниями).

Осталось разобраться с функцией, которая запускает вычисления в монаде - `runSM`. Она принимает начальное состояние и вычисление и возвращает значение вычисления и конечное состояние.

Если окинуть взглядом нашу работу, то станет видно, что мы пытаемся определить конеч-

```
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] []             ⇒   []

data SM a = SM (S -> (a,S))  -- Тип монады
```

ное вычисление в качестве набора шагов (функций типа `SM a`) с помощью упорядочения (`>>=`) и `return`. Шаги могут взаимодействовать с состоянием (через `readSM` и `updateSM`) или вовсе не затрагивать его. Тем не менее, использование (или неиспользование) состояния сокрыто - мы вызываем или упорядочиваем вычисления одинаково, независимо от того, используют ли они `S` или нет.

Вместо того, чтобы рассмотреть примеры использования простой монады состояния, приступим к изучению примеров монады состояния посложнее. Объявим небольшой *встроенный язык* (embedded language) для требующих ресурсов вычислений, который является языком специального назначения, базирующийся на основе типов и функций Haskell . Такой язык использует базовые инструменты Haskell , чтобы образовать библиотеку операций и типов, специально заточенных для определённой области применения.

Представьте, что необходимо произвести вычисление, требующее некий вид ресурса. Если ресурс доступен, то происходит вычисление; если же ресур недоступен, то вычисление спит[42]. Для обозначения ресурса, которым управляет наша монада, будем использовать тип `R`. Вот его определение:

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

Каждое вычисление является функцией доступных ресурсов, возвращающих оставшиеся ресурсы, объеденнённые либо (either) с результатом, либо с отложенным вычислением типа `R a`, которое засыпает до тех пор, пока ресурс не освободят.

Экземпляр класса `Monad` для типа `R` имеет следующий вид:

Тип `Resource` используется таким же образом, как и состояние в монаде состояния. Его определение можно прочитать так: для объеденения двух требующих ресурсов вычислений `c1` и `fc2` (функцию, возвращающую `c2`) передайте начальные ресурсы в `c1`. Результатом будет одно из двух:

- Значение `v` и оставшиеся ресурсы, которые служат для определения следующего вычисления (вызова `fc2 v`)
- Отложенное вычисление `pc1` и ресурс, оставшийся на момент входа в спящее состояние.

При входе в спящее состояние не стоит забывать о следующем вычислении, так как `pc1` откладывает лишь первое вычисление `c1`, а значит необходимо связать `c2` с ним, чтобы отложить все вычисления. Определение `return` оставляет ресурс нетронутым, но помещает `v` в монаду.

Объявление экземпляра определяет базовую структуру монады, но не определяет того, как используются ресурсы. Эта монада может быть использована для контролирования различных типов ресурсов, либо реализовать различные политики использования ресурсов. Продемонстрируем очень примитивное определение ресурса: пусть `Resource` будет `Integer`, отражающий количество доступных шагов вычисления:

[42] Используется устоявшийся перевод для обозначения простаивания операций/потоков - Прим.пер.

```
instance        Monad SM where
-- Определение  распространения  состояния
SM c1 >>= fc2   =                 codeSM (\ s0 -> let   (r,s1) = c1 s0
                                                        SM c2 = fc2 r in
                                                        c2 s1)
return k        =                 SM (\ s -> (k,s))

-- Получение    состояния         монады
readSM          ::                SM S
readSM          =                 SM (\ s -> (s,s))

-- Обновление   состояния         монады
updateSM        ::                (S -> S) -> SM ()     -- изменение состояния
updateSM f      =                 SM (\ s -> (() f,s))

-- Вычисления   в монаде          SM
runSM           ::                S -> SM a -> (a,S)
runSM           =                 c s0

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)))
```

Следующая функция использует шаг (ресурс) до тех пор, пока их больше не останется:

Конструкторы `Left` и `Right` есть части типа `Either`. Настоящая функция продолжает вычисление в `R`, возвращая `v` до тех пор, пока остался хотя бы один шаг. Если не осталось ни одного шага, функция `step` откладывает текущее вычисление (сохраняется в `c`) и передаёт его обратно в монаду.

Мы имеем средство для определения последовательности требующих ресурсов вычислений (монад), а также средство для выражения использования ресурсов (с помощью `step`). Наконец, разберёмся с тем, как определять вычисления, используя нашу монаду.

Представьте функцию инкремента для нашей монады:

Это определяет инкремент как один шаг вычислений. `<-` необходим для извлечения значения из монады. Типом `iValue` является `Integer`, а не `R Integer`.

Такое определение функции инкремента не может удовлетворить нас, так как не согласуется со стандартным определением. Можем ли мы „обернуть“ имеющиеся операции (вроде `+`), чтобы они смогли существовать в нашем мире монад? Начнём с набора функций „поднятия“ (`lifting`). Они и будут заниматься этим. Вот определение функции `lift1` (которое слегка отличается от определения `liftM1` в библиотеке `Monad`):

Она принимает функцию одного аргумента `f` и создаёт функцию `R`, которая выполняет полученную функцию за один шаг. При помощи `lift1` функция `inc` превращается в:

Уже лучше, но все ещё далеко от совершенства. Вот `lift2`:

Обратите внимание на то, что эта функция явно определяет порядок вычислений для

```
type Resource  =  Integer

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

принимаемой функции - вычисление a1 выполняется до вычисления a2.

С помощью lift2 определим новую версию == для монады R:

Мы были обязаны использовать какое-то другое имя, так как == уже используется, но в некоторых случаях можно использовать одинаковое имя для „функций поднятия“ и обычных функций. Следующее объявление экземпляра позволяет всем операторам из Num быть использованым в R:

Функция fromInteger применяется неявно ко всем целым константам в программах на Haskell (см. раздел 10.3); такое определение позволяет целым константам иметь тип R Integer. Теперь можно записать инкремент самым обычным образом:

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

Обратите внимание на то, что мы не можем „поднять“ класс Eq так же, как и Num, так как сигнатура ==* несовместима с допустимыми перегрузками ==, ведь тип результата первой - R Bool,а не Bool.

Перед тем как показать интересные примеры вычислений с R, давайте добавим „условный оператор“. Так как мы не можем использовать if (необходим тип Bool, а не R Bool), назовём такую функцию ifR:

Теперь мы готовы для написания программы чуть больше:

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

Хотя это и нестандартное определение функции для вычисления факториала, зато оно легче читается. Идея написания новых определений для уже существующих операций (таких как + или if) является основной при определении встроенного языка на Haskell . Монады чрезвычайно полезны для инкапсуляции семантики таких встроенных языков, что помогает писать модульные и очень „грамотные“ приложения.

Давайте же запустим наши программы! Следующая функция запускает программу с максимальным количеством шагов, равным M:

Тип Maybe используется для того, чтобы работать с вычислениями, незавершёнными за допустимое количество шагов. Теперь можно вычислить

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

Можно добавить ещё несколько интересных функций к нашей монаде. Представьте следующую функцию:

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

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

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

Она запускает параллельно два вычисления, возвращая значение первого завершившегося. Одно из возможных определений такой функции приведено далее:

```
c1 ||| c2 = oneStep c1 ( \ c1' -> c2 ||| c1')
```

Здесь берётся один шаг для `c1`, возвращается значение, если `c1` завершилось. Если `c1` возвращает отложенное вычисление (`c1'`), то это приводит к вычислению `c2 ||| c1'`. Функция `oneStep` принимает один шаг в аргумент, возвращая либо вычисленное значение, либо передавая остаток вычисления в `f`. Определение `oneStep` очень простое: она передаёт `1` в `c1` как ресурсный аргумент. Если конечное значение „достижимо“, оно возвращается, изменяя возвращенное число шагов (возможно, что вычисление произойдёт, не взяв шаг, это означает, что возвращаемое число свободных ресурсов не обязано равняться нулю). Если вычисление откладывается, то изменённое количество ресурсов передаётся в последнее возобновление.

Можно вычислить выражения вроде `run 100 (fact (-1) ||| (fact 3))` без циклов, так как вычисления параллельны. Возможно множество вариантов, основанных на этом. Например, можно добавить трассирующие функции. Можно включить эту монаду в стандартную `IO`, чтобы позволить вычислениям в `M` взаимодействовать с окружающим миром.

Хотя рассмотренный пример имеет достаточно высокую сложность, он служит всего-лишь показателем мощи монад при описании семантики различных систем. Мы создали так называемый *Язык Специфической Области* (Domain Specific Languages), для определения которых Haskell идеально подходит. С помощью Haskell разработано множество других DSL, см. `haskell.org`. Одним примером интересного языка является Fran - язык для анимаций, другим является Haskore - язык для описания компьютерной музыки.

11 Числа

Haskell имеет богатый набор численных типов, основанных на типах из Scheme [7], которые в свою очередь основаны на Common Lisp [8]. (Но эти языки динамически типизированны.) В стандартный набор входят целые фиксированной и бесконечной точности, рациональные числа, вещественные числа (в.т.ч. комплексные) с плавающей точкой одинарной и двойной точности. В данной главе мы рассмотрим основные характеристики численных классов. Отправляем читателя за подробностями к § 6.4.

11.1 Структура численных классов

Численные классы типов(класс `Num` и его подклассы) работают со многими классами Haskell . `Num` является подклассом `Eq`, но не `Ord` из-за того, что комплексные числа не работают с

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

lift2    ::  (a -> b ->       c) -> (R a -> R b -> R c)
lift2 f  =   \ ra1 ra2 -> do  a1 <- ra1
                              a2 <- ra2
                              step (f a1 a2)
```

предикатами упорядочения. Подкласс `Real`, тем не менее, является подклассом `Ord`.

Класс `Num` включает несколько базовых операций: помимо других это сложение, вычитание, отрицание, умножение, а также взятие модуля.

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

[`negate` - это функция, которая применяется единственным префиксным оператором в Haskell - унарным минусом; мы не можем назвать её (-), так как это имя уже занято функцией вычитания. Например, `-x*y` эквивалентно `negate (x*y)`. (Префиксный минус имеет такой же уровень предшествования (который ниже, чем у умножения), как и инфиксный минус.)]

Учтите, что `Num` *не* имеет оператора деления; два различных оператора имеются в двух непересекающихся подклассах `Num`:

Класс `Integral` включает операцию деления целых чисел, а также другие. Стандартными экземплярами `Integral` являются `Integer` (неограниченный, или математические целые числа, известные также как „большие числа“ (bignums)) и `Int`(ограниченные, машинные целые числа, эквивалентные минимум 29-битным бинарным знаковым числам). Конкретная система Haskell может иметь и другие типы целых чисел. Обратите внимание на то, что `Integral` является подклассом `Real`, а не наследником `Num`. Это значит, в первую очередь, что не имеется реализации чисел Гаусса.

Все другие численные типы исходят от класса `Fractional`, который имеет оператор деления `(/)`. Подкласс `Floating` содержит тригонометрические, логарифмические, экспоненциальные функции.

Подкласс классов `Fractional` и `Real` - `RealFrac` имеет функцию `properFraction`, которая делит число на целую и дробную части, а также набор функций округления по различным правилам целых чисел:

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

Подкласс классов `Floating` и `RealFrac` - `RealFloat` имеет различные специализированные функции для эффективного доступа к компонентам (экспоненте и мантиссе) чисел с плавающей точкой. Стандартные типы `Float` и `Double` исходят из `RealFloat`.

11.2 Конструируемые числа

Стандартные численные типы `Int, Integer, Float, Double` являются примитивными. Другие типы конструируются из них с помощью конструкторов типа.

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

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

`Complex` (из библиотеки `Complex`) является конструктором типа, который конструирует комплексный тип класса `Floating` из `RealFloat`:

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

Символ `!` указывает на строгость операции, см. раздел 6.3. Обратите внимание на контекст `RealFloat a`, который накладывает ограничение на тип аргумента. Поэтому стандартными комплексными типами являются `Complex Float` и `Complex Double`. Из объявления `data` видно, что комплексное число записано с помощью `x :+ y`, а аргументы являются Декартовыми действительной и мнимой частью соответсвенно. Так как `:+` есть конструктор данных, то можно использовать сопоставление с образцом:

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

Точно так же, конструктор типа `Ration` (из библиотеки `Rational`) конструирует рациональный тип класса `RealFrac` из экземпляра `Integral` (`Rational` является синонимом типа для `Ratio Integer`). Тем не менее, `Ratio` - это абстрактный конструктор типа. Вместо использования конструкторов данных (таких как `:+`), рациональные числа используют функцию `'%'` для построения числа из двух других. Вместо сопоставления с образцом, имеются функции-селекторы[43]:

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

Для чего нам такое различие? Комплексные числа в декартовом виде являются необычными - не существует нетривиальных сущностей, использующих `:+`. С другой стороны, рациональные числа не являются чем-то необычным, просто они имеют канонический (редуцированный) вид, которому должен соответствовать абстрактный тип данных. Это не всегда верно: `numerator (x % y)` эквивалентен `x`, а действительная часть `x:+y` всегда равна `x`.

11.3 Численное приведение и перегруженные литералы

Стандартная прелюдия и библиотеки имеют различные перегруженные функции, которые служат для явного приведения:

Две из них используются неявно для того, чтобы были доступны перегруженные численные литералы: целый численный литерал (без десятичной точки) эквивалентен аппликации `fromInteger` к значению этого числа, имеющего тип `Integer`. Точно так же численный литерал с плавающей точкой (имеющий десятичную точку) рассматривается как аппликация

[43] Соответственно, для доступа к числителю (numerator) и знаменателю (denominator) - Прим.пер.

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

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

`fromRational` к значению числа, имеющего тип `Rational`. Поэтому `7` имеет тип `(Num a) => a`, а `7.3` имеет тип `(Fractional a) => a`. Это означает, что мы можем использовать числовые литералы в обобщенных функциях, например так:

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

Этот, скорее нетрадиционный, вид перегрузки имеет дополнительный плюс, заключающийся в том, что метод интерпретации численных литералов как чисел может быть указан в объявлении экземпляра `Integral` или `Fractional` (так как `fromInteger` `fromRational` являются соответствующими операциями этих классов). Например, экземпляр класса `Num` для `(RealFloat a) => Complex a` имеет следующий метод:

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

Это означает, что экземпляр `Complex` для `fromInteger` объявлен для создания комплексных чисел, чья действительная часть „поставляется“ подходящим экземпляром `RealFloat` для `fromInteger`. Поэтому любые пользовательские типы (допустим, кватернионы) могут быть использованы с перегрузкой чисел.

Вспомним наше первое определение `inc` из раздела 2:

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

Наиболее общим типом этой функции является `(Num a) => a -> a`. Такая явная сигнатура типа работоспособна, но она *более специфична*, чем сигнатура, использующая основной тип (задание ещё более общего типа, чем основной приведёт к ошибке). Сигнатура типа ограничивает тип функций `inc`, что означает, что `inc (1::Float)` неверно типизированно.

11.4 Численные типы по-умолчанию

Посмотрите на следующее определение:

Функция возведения в степень (`^`) (один из трёх операторов для различных типов, см. § 6.8.5) имеет тип `(Num a, Integral b) => a -> b -> a`, а так как `2` имеет тип `(Num a) => a`, то `x ^2` имеет тип `(Num a, Integral b) => a`. Возникла проблема, так как нет способа разрешить перегрузку, ассоциированную с переменной типа `b`. Она входит в контекст, но не входит в выражение типа. Программист указал, что необходимо возвести `x` в квадрат, но не указал, какой тип имеет двойка - `Int` или `Integer`. Конечно, можно решить и это:

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

```
where
          oneStep       ::  R a -> (R a -> R a) -> R a
       oneStep (R c1) f  =
           R ( \ case    c1  1 of
                             (r', Left v) -> (r+r'-1, Left v)
                             (r', Right c1') -> -- r‘ должен быть равен 0
                              let R next = f c1' in
                              next (r+r'-1))

       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
```

Ясно, что это не выход из положения.

Вообще, такой вид неоднозначности перегрузки затрагивает не только числа:

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

Какой тип должна иметь считанная строка? Этот пример гораздо сложнее предыдущего, так как там подошел бы любой экземпляр класса `Integral`, а здесь существует слишком много различий в каждом из экземпляров класса `Text`.

Из-за различий между численной и любой другой перегрузкой, Haskell имеет решение проблемы для чисел: каждый модуль может содержать *default-объявление* (default declaration), состоящее из ключевого слова `default` и списка численных монотипов (типов без переменных), разделённых запятыми и записанных в скобках. Если встречается неоднозначность (как с `b` выше), и один из классов является численным (а все классы стандартны), то просматривается default-список. Самый первый подходящий к контексту тип будет использован для разрешения неоднозначности. Для примера, если имеется default-объявление `default (Int, Float)`, то неоднозначность возведения в степень будет разрешена с помощью типа `Int`. (См. § 4.3.4 для подробностей)

Самым стандартным списком по-умолчанию („default default“) является `(Integer,Double)`, но и `Integer,Rational,Double` подойдёт не хуже. Некоторые осторожные программисты предпочитают использовать `default ()`, чтобы вовсе не иметь типов по-умолчанию.

12 Модули

Программа на Haskell состоит из набора *модулей*. Модуль в Haskell служит для создания пространства имён и абстрактных типов данных.

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

Модуль может содержать любые виды объявлений, которые мы уже рассматривали: объявления типов и данных, объявления устойчивости, объявления классов и экземпляров, сигнатуры типов, определения функций, образцы для сопоставления. Кроме объявлений импорта (рассматриваются далее), которые должны идти в первую очередь, объявления могут располагаться в любом порядке (область видимости высшего уровня взаимно рекурсивна).

Строение модулей Haskell достаточно консервативно: пространство имён модулей полностью плоское (flat), а модули не являются никоим образом „первоклассными сущностями“. Имена модулей должны начинаться с заглавной буквы и состоять из чисел и букв. Не существует формальной связи между именами модулей Haskell и именами файлов, которые (обычно) согласуются с ними. Вообще такой связи может и не быть. Это означает, что один файл может содержать в себе множество модулей, а один модуль может „распадаться на множество файлов“. Конечно, типичные реализации Haskell обычно имеют более жесткие связи между файлами и модулями.

Модуль, по сути, является одним большим объявлением, которое начинается с ключевого слова `module`. Вот пример модуля, названного `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 leaf right)       =   fringe left ++ fringe right
```

Тип `Tree` и функция `fringe` должны быть вам известны. Они приводились в пример в разделе 2.2.1. [Из-за ключевого слова `where`, разметка работает уже на высшем уровне модуля, а значит все объявления должны быть расположены на одной колонке (обычно на первой). Обратите внимание на то, что модуль имеет такое же имя, как и тип. Это разрешено.]

Этот модуль явно *экспортирует* `Tree,Leaf,Branch` и `fringe`. Если список экспорта после слова `where` опущен, то *все* имена, находящиеся на высшем уровне модуля, являются экспортируемыми. (В нашем примере все имена явно экспортируются.) Учтите, что имя типа и его конструкторы были сгруппированы вместе, например `Tree(Leaf,Branch)`. Можно было бы написать просто `Tree(..)` для краткости. Допускается экспортирование подмножества конструкторов. Имена, перечисленные в списке экспорта, не обязаны быть локальными для экспортирующего модуля; могут быть перечислены любые имена в области видимости.

Теперь можно *импортировать* модуль `Tree`:

```
module Main (main) where
import Tree (Tree(Leaf,Branch),fringe)
main = print (fringe(Branch(Leaf 1) (Leaf 2)))
```

Различные составляющие импортированные *в* и экспортированные *из* модуля, называются *сущностями* (entities). Обратите внимание на явный список импорта в объявлении импорта; если он опущен, то все сущности, экспортируемые из модуля `Tree`, будут импортированы.

12.1 Квалификаторы имён

При импортировании имён в пространство имён может возникнуть проблема - что если текущий модуль уже содержит такие имена? Haskell позволяет решить возникшую проблему с помощью *квалифицированных имён* (qualified names). Объявление импорта может использовать ключевое слово `qualified`, которое заставляет импортируемые имена иметь префикс в виде названия импортируемого модуля. За такими префиксами следует точка ('.') без пробелов. [Квалификаторы являются частью синтаксиса. Поэтому `A.x` и `A . x` отличаются: первый является квалифицированным именем, а втрой использует инфиксную функцию '.'] Например, используя модуль `Tree`, описанный выше, можно записать:

```
   module Fringe(fringe) where
import Tree(Tree(..))

fringe :: Tree a -> [a] -- Другое определение 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)))
 main  =     print (Fringe.fringe(Branch (Leaf 1) (Leaf 2)))
```

Некоторые программисты предпочитают всегда использовать квалифицированные имена для импортируемых сущностей. Другие любят использовать короткие имена и применяют квалифицированные имена лишь по необходимости.

Квалифицированные имена используются для разрешения конфликтов имён между разными сущностями, которым даны одинаковые имена. Но что, если сущность импортируется из более чем одного модуля? К счастью, это разрешено - сущность может быть импортирована без вопросов. Компилятор знает, что сущности из различных модулей являют собой одно и то же.

12.2 Абстрактные типы данных

Помимо контролирования пространств имён, модули являются единственным средством для создания абстрактных типов данных (abstract data types) (ADT). Например, главной характеристикой ADT является то, что *тип представления* (representation type) *сокрыт*; все операции с ADT выполняются на абстрактном уровне, который не зависит от представления. Для примера, тип `Tree` достаточно прост для того, чтобы мы сделали его абстрактным. Соответствующий ADT может выглядеть так:

Вот сам модуль:

Обратите внимание на то, что в списке экспорта имя `Tree` приведено без своих конструкторов. Поэтому `Leaf` и `Branch` не экспортируются, и единственным способом построить или использовать деревья является применение различных (абстрактных) операций. Преимущество такого сокрытия заключается в том, что мы можем позже *изменить* тип представления без воздействия с имеющимися пользователями типа.

```
data Tree a  --  обычное имя типа
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

module TreeADT  (Tree,leaf,branch,cell
                 left,right,isLeaf) where
```

12.3 Особенности

Приведём перечень различных особенностей модульной системы. Обратитесь к Haskell Report за деталями.

- Объявление `import` может выборочно скрывать сущности с помощью `hiding`. Это удобно, когда необходимо явно исключить некоторые имена, которые уже используются для других целей.
- Объявление `import` может иметь выражение `as` для указания другого квалификатора вместо имени импортированного модуля. Это удобно, если необходимо сократить длинные имена или изменить имя модуля без изменения всех квалификаторов.
- Программы неявно импортируют модуль `Prelude`. Явный импорт `Prelude` перекрывает неявный импорт всех его имён. Поэтому при объявлении

  ```
  import Prelude hiding length
  ```

  функция `length` не будет импортирована, что позволяет определить её по-другому.
- Объявления экземпляров не должны явно включаться в списки импорта или экспорта. Каждый модуль экспортирует все его объявления экземпляров, и каждый импорт включает все экземпляры в область видимости.
- Методы классов могут быть названы либо в стиле конструкторов данных (в скобках и следующими за именем класса), либо как обычные переменные.

Модульная система Haskell достаточно консервативна. Существует множество правил импорта и экспорта значений. Большинство из них естественны - для примера, нельзя импортировать две различные сущности с одинаковым именем. Другие правила не такие простые - например, для типа и класса не может быть больше одного экземпляра, связывающего их. О подробностях см. Haskell Report (§5).

13 Подводные камни типизации

В этой короткой главе мы рассмотрим несколько обычных проблем, возникающих у новичков при работе с системой типов.

```
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
```

13.1 Let-связанный полиморфизм

Любой язык с системой типов Хиндли-Милнера имеет так называемый *let-связанный полиморфизм* (let-bound polymorphism), так как идентификаторы, не связанные с помощью выражений `let` или `where` (на высшем уровне модуля), ограничены в полиморфизме. В частности, *лямбда-связанные функции* (lambda-bound function) (т.е. которые передают аргумент другой функции) не может иметь разные варианты. Вот пример:

```
let f g = (g [], g 'a' -- неверная типизация
in f ( \ x->x
```

Здесь `g`, связанное с лямбда-абстракцией, которая имеет основной тип `a->a`, используется в `f` двумя различными способами: один из них имеет тип `[a]->[a]`, а другой - `Char->Char`.

13.2 Численная перегрузка

Очень легко забыть, что числа *перегружаются*, а не *явно приводятся* к разным типам, как в других языках программирования. Более общие численные выражения на самом деле не могут быть такими общими. Вот распространённая ошибка типизации:

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

(/) требует дробных аргументов, но `length` имеет `Int`. Несоответствие типов (type mismatch) можно разрешить с помощью явного приведения:

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

13.3 Ограничение мономорфизма

Система типов Haskell имеет ограничение, касающееся классов, которые не относятся к системе типов Хиндли-Милнера: *мономорфное ограничение* (monomorhipsm restriction). Такое ограничение введено из-за неоднозначности типизации и подробнее рассматривается в Haskell Report (§4.5.5). Вот краткое объяснение этой особенности Haskell :

Любой идентификатор, связанный с образцом (шаблоном) (включая связь с единственным идентификатором) и не имеющий явной сигнатуры типа, должен являться *мономорфным* (monomorphic). Идентификатор называют мономорфным, если он не перегружен или если он имеет всего одну перегрузку (и такой идентификатор не экспортируется).

Пренебрежение этим правилом приводит к неверной типизации (что вызывает статическую ошибку типизации). Можно довольно легко преодолеть эту проблему с помощью написания явной сигнатуры типа. Учтите, что подойдёт *любая* сигнатура (конечно же, если она верна).

Обычно нарушение этого ограничения наступает при объявлениях функций в высокоуровневом стиле, например как с функцией `sum` из прелюдии:

```
sum = foldl (+) 0
```

Без сигнатуры это приведёт к ошибке. Можно использовать следующую сигнатуру типа:

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

Обратите внимание, что не возникло бы никакой проблемы, если бы мы описали функцию так:

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

Запомните, что ограничение, рассмотренное в данной главе, касается лишь шаблонных связей.

14 Массивы

В идеале массивы в функциональных языках должны быть представлены просто функциями, отображающими индексы в значения, но для эффективного доступа на практике нужно быть уверенными, что используются все преимущества таких функций, которые являются изоморфными для конечного непрерывного подмножества (finite contiguous subset) целых чисел. Haskell поэтому работает с массивами как с абстрактным типами данных, а не как с функциями.

Существует два подхода к представлению массивов: *инкрементное* и *монолитное* определение. В первом случае имеется функция, которая производит пустой массив данного размера, а также функция, которая принимает массив, индекс и значение, чтобы произвести новый массив (у которого отличается лишь один элемент). Конечно же, такая реализация будет очень неэффективна, потому что она будет каждый раз создавать копию исходного массива, а также требовать линейного времени для доступа к элементу. Поэтому такой подход часто требует использования специальных техник статического анализа, а также специальной аппаратуры, которая позволяет избежать лишнего копирования. Монолитная реализация, с другой стороны, конструирует массив весь сразу, без использования временной памяти. Haskell имеет и инкрементальный оператор обновления массива (incremental array update operator), но обычно работа идёт с монолитной реализацией.

Массивы не входят в прелюдию, но в стандартной библиотеке имеется набор нужных операторов. Для использования массивов необходимо импортировать модуль `Array`.

14.1 Типы индексов

Библиотека `Ix` определяет класс типов для индексов массивов:

Имеются объявления экземпляров для `Int,Integer,Char,Bool`, а также кортежи из элементов класса `Ix` длиной до 5. В дополнение для перечисляемых и кортежных типов возможно автоматическое наследование экземпляров. Мы рассматриваем примитивные типы как

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

индексы векторов, а кортежи как индексы многомерных массивов. Обратите внимание на то, что первым аргументом каждой из операций класса `Ix` является пара индексов: передаются *размерности* (bounds)[44] массива (первый и последний индексы каждого измерения). Для примера, размерностью 10-элементного вектора, первый элемент которого имеет индекс 0, будет `(0,9)`, а размерностью матрицы 100 на 100 (при условии, что индексы начинаются с 1) будет `((1,1),(100,100)`. (В других языках программирования размерности обычно указываются например как `[1:100, 1:100]`, но представленная выше система подходит к системе типов Haskell лучше, так как каждая размерность имеет тот же тип, что и индекс.)

Операция `range` принимает пару размерностей и создаёт список индексов, лежащих между этими границами. Например,

```
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 данный индекс между парой данных размерностей(работает и для кортежа). Операция `index` позволяет привести (сместить) адрес элемента: она принимает пару размерностей и индекс (лежащий внутри границ) и возвращает начинающийся с нуля приведённый (смещённый) индекс. Вот пример:

```
index (1,9) 2 ⇒ 1
index ((0,0),(1,2)) (1,1) ⇒ 4
```

14.2 Создание массивов

Монолитная функция создания массивов создаёт массив по паре размерностей, а также список пар индекс-значение (так называемый *ассоциативный список* (association list)):

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

Вот пример определения массива квадратов чисел от 1 до 100:

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

Как и в этом выражении, при объявлении обычно используются составления списков. Вообще, такое использование очень похоже на *составления массивов* (array comprehensions) из языка Id[6].

Обращение к элементам происходит с помощью инфиксного оператора `!`, а размерность массива можно получить с помощью функции `bounds`:

```
squares!7 ⇒ 49
bounds squares ⇒ (1,100)
```

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

Можно обощить этот пример с помощью параметризации размерностей и функции, применяемой к каждому индексу:
Поэтому можно определить `squares` как `mkArray (\i -> i * i) (1,100)`.

Многие массивы определяются рекурсивно, т.е. значения некоторых их элементов зависят от значения других элементов. В следующем примере мы имеем функцию, возвращающую 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]])
```

Другим примером такой рекуррентности является *волновая матрица* (wavefront matrix) размерности `n` на `n`, в которой элементы первой строки и первого столбца имеют значение 1, а другие элементы являются суммами их западных, северозападных и северных соседей:

```
wavefront     ::   Int ->   Array (Int,Int) Int
wavefront 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]])
```

Волновая матрица названа так, потому что в параллельной реализации рекуррентность означает, что вычисления могут начаться с первой строки и первого столбца и продолжаться в виде волны, проходящей с северо-запада на юго-восток. Важно заметить, что „направление“ вычислений не может указываться ассоциативным списком.

В каждом из наших примеров мы задавали уникальную связь для каждого индекса массива, но только для допустимых индексов. Было необходимо сделать это для того, чтобы массив был полностью определён. Связь с индексом, не входящим в размерность, приводит к ошибке. Если индекс опущен или имеется несколько индексов[45], то это не приведёт к мгновенной ошибке, а лишь к неопределённому значению в массиве для этого индекса. Обращение к элементу по такому индексу приведёт к ошибке.

14.3 Аккумуляция

Можно ослабить ограничение, налагаемое на то, что каждый индекс должен иметься лишь в одном экземпляре, с помощью совмещения многих значений с одним индексом. Результатом будет являться т.н. *аккумулированный массив*[46] (accumulated array):

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

[44] Хотя в данном контексте подразумеваются именно *границы* массива, здесь и далее используется устоявшийся перевод, несмотря на неточность - Прим.пер.

[45] ???<дописать> - Прим.пер.

[46] Иногда называемый также *мультимассивом* - Прим.пер.

Первым аргументом `accumArray` является *функция аккумуляции* (accumulating function), вторым - начальное значение (одинаковое для всех элементов массива), а остальными аргументами являются размерности и ассоциативный список, как и в функции `array`. Обычно функция аккумуляции есть `(+)`, а начальное значение - нуль. Для примера, следующая функция принимает пару размерностей и список значений (индексного типа), а возвращает гистограмму, т.е. таблицу, показывающую распределение частот элементов списка:

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

Представьте, что имеется набор измерений на интервале `[a,b)`, и мы желаем разделить его на подинтервалы длиной в десять измерений[47], а затем получить гистограмму для каждого полученного подинтервала:

```
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)
```

14.4 Инкрементальное обновление

В дополнение к монолитному созданию массивов, Haskell имеет и функцию для инкрементального обновления массивов, которой является инфиксный оператор `//`. Обновление (замена) элемента `i` на `v` в массиве `a` записывается так: `a // [(i,v)]`. Квадратные скобки необходимы, так как левым аргументом для `(//)` служит ассоциативный список, обычно имеющий верное подмножество индексов массива:

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

Как и в функции `array`, индексы в ассоциативном списке должны иметь уникальные значения. Для примера, вот 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 разных составлений для одного списка индексов `j` слегка неэффективна. Это очень похоже на написание двух идентичных циклов вместо одного в императивных языках. Не бойтесь, у нас имеется возможность решить эту проблему с помощью „объединяющей циклы оптимизации“ (loop fusion optimization):

14.5 Пример: умножение матриц

Закончим наше рассмотрение массивов Haskell примером сдобренного перегрузкой достаточно обощенного умножения матриц. Так как для этого используются лишь операции сложения и умножения элементов, мы получим, если не будем сильно сопротивляться, функцию умножения матриц из элементов любого численного типа. Если же мы будем применять лишь

[47] На „декады“ - Прим.пер.

```
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
```

(!) и операции класса Ix к индексам, то мы сможем получить ещё более обобщенную функцию. Вообще, можно будет использовать различные типы для всех четырёх индексов. Для простоты, будем считать, что левый индекс для строк и правый индекс для столбцов имеют одинаковый тип, а так же что матрицы имеют равные размерности[48]:

```
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 "incompatible bounds"
```

Можно определить matMult с помощью accumArray. При этом такая реализация сильнее похожа на императивную:

Можно ещё больше обобщить функцию с помощью создания функции высшего порядка, просто заменив sum и * функциональными параметрами:

Любители APL легко признают очень полезные функции вида

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

В первом примере аргументами являются численные матрицы, а (i, j)-ый элемент результата есть максимальная разница между соответствующими элементами i-й строки и j-ого столбца. Во втором случае аргументами являются матрицы, тип элементов которых имеет класс Eq, а результатом является матрица, (i, j)-ый элемент которой будет иметь значение True только, если i-ая строка первого аргумента и j-ый столбец второго аргумента равны (как векторы).

Обратите внимание на то, что тип элементов genMatMult не обязан быть одинаковым, а просто подходить в качестве параметров для функции star.

Можно продолжить обобщение, если отменить требование на то, чтобы тип индекса первой строки и тип индекса второго столбца были одинаковы. Две матрицы подойдут, если длина столбца первой равна длине строки второй. Оставляем реализацию такой версии этой

[48] В случае различных размерностей выводится ошибка „incompatible bounds“(„несовместимые размерности“) - Прим.пер.

```
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 "incompatible bounds"

genMatMult  ::  (Ix a, Ix b, Ix c) =>
                ([f] -> g) -> (d -> e -> f) ->
                Array (a,b) d -> Array (b,c) e -> Array (a,c) g
```

функции читателю. (**Подсказка:** Используйте операцию `index` для определения длин.)

15 Что далее?

Большой объем ресурсов о Haskell доступен на сайте `haskell.org`. Вы найдёте компиляторы, демо-программы, документы и другую полезную информацию о функциональных языках и Haskell в частности. Компиляторы и интерпретаторы Haskell доступны для практически любых платформ и операционных систем. Система Hugs довольно небольшая и портабельная - а значит она является идеальной базой для изучения Haskell .

16 Благодарности

Благодарим Патрицию Фазель (Patricia Fasel), Марка Мандта (Mark Mundt) из Los Alamos, Ника Карриеро (Nick Carriero), Чарльза Консела (Charles Consel), Амира Кишона (Amir Kishon), Сандру Лузмор (Sandra Loosemore), Мартина Одерски (Martin Odersky), Дэвида Рошберга (David Rochberg) из Yale University за их быстрый отклик и помощь в исправлении ранних версий данного труда. Отдельно благодарим Эрика Мэйра (Erik Meijer) за его многочисленные поправки и комментарии нового материала, добавленного в версии 1.4.

```
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 "incompatible bounds"
```

Ссылки

[1] R.Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, New Yourk, 1998.

[2] A.Davie. *Introduction to Functional Programming System Using Haskell.* Cambridge University Press, 1992.

[3] P.Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359-411, 1989.

[4] Simon Peyton Jones (редактор). Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language. *Yale University, Department of Computer Science Tech Report* YALEU/DSC/RR-1106,Feb 1999.

[5] Simon Peyton Jones (редактор). The Haskell 98 Library Report. *Yale University, Department of Computer Science Tech Report* YALEU/DSC/RR-1105,Feb 1999.

[6] R.S.Nikhil. Id (version 90.0) reference manual. Technical report, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1990.

[7] J.Rees and W.Clinger (редакторы). The revised report on the algorithmic language Scheme. *SIG-PLAN Notices*, 21(12):37-79, December 1986.

[8] G.L.Steele Jr. *Common Lisp: The Language.* Digital Press, Burlington, Mass.,1984.

[9] P.Wadler. How to replace failure by a list successes. In *Proceedings of Conference on Functional Programming Languages and Computer Archicture*, LNCS Vol.201, pages 113-128. Springer Verlag, 1985.

[10] P.Wadler. Monads for Functional Programming In *Advanced Functional Programming*, Springer Verlag, LNCS 925, 1995.

17 Словарь терминов

Поддерживать в алфавитном порядке.

obsolete - устаревший
typeful - строго-типизированный
type expressions - типовыражения
first-class values - первоклассный (сущности)
typing - типизация
eqauation - уравнение
declaration - объявление
type signature declaration - объявление сигнатуры типа
to evaluate - вычислять
type safe - типобезопасный
ill-typed - несогласованное с типом
to infer - выводить
polymorphic - полиморфные
type variables - переменные типов
pattern matching - сопоставление с образцом
binding to - привязываются к ...
principal type - основной тип
least general - наименее общий
Hindley-Milner type system - система типов Хиндли-Милнера
type constructor - конструктор типов
data constructor - конструктор данных
tuple type - тип-кортеж
namespace - пространство имён
type synonyms - синонимы типа
guards - гарды.
list comprehensions - составления списков.
generator - генератор
arithmetic sequences - арифметические последовательности
curried function - функции Карри
application - аппликация („применение“ или в некоторых случаях - „вызов“)
partial application - частичная аппликация
higher-order function - функция высшего порядка
function composition - композиция функций
section - секция (частичная аппликация инфиксных операторов)
fixity declaration - объявление устойчивости
non-terminating expression - незавершающееся выражение
non-strict functions - нестрогие функции
bottom - основание
circular list - цикличный список
Case Expressions - Case-выражения
backtracking - бэктрекинг
linearity - линейность
never fail to match a value - всегда сопоставляются со значением
irrefutable - неопровержимый
refutable - опровержимый
to diverge - дивергировать
top-level - высшего уровня

conditional expressions - условные выражения
lazy patterns - ленивые шаблоны
simulation programs - симуляторы
stream - поток
request - запрос
response - отклик
destructured form - разрушенный вид
pattern binding - привязка шаблона
Lexical Scoping - Лексическая область видимости
Let-expressions - Let-выражения
mutually recursive - взаимно рекурсивные
layout - разметка
keyword - ключевое слово
parametric - параметрический
ad hoc - специальный
equality operator - оператор равенства
type classes - классы типов
overloading - перегрузка
referential transparency - ссылочная прозрачность
instances - экземпляры
operations - операции
contex - контекст
instance declarations - объявления экземпляров
method - метод
default methods - методы по-умолчанию
class extension - расширение классов
inherits - наследует
superclass - суперкласс
subclass - подкласс
multiple inheritance - множественное наследование
higher-order types - типы высшего порядка
generic - общий
kind - вид
internal mutable state - внутреннее изменяемое состояние
newtype-declaration - newtype-объявление
extra overhead - накладные расходы
field labels - метки полей
field names - имена полей
selector function - функция-селектор
update function - Функция обновления
destructive update - деструктивное обновление
thunk - переходник
heap - хип
garbage collector - сборщик мусора
strictness flag - флаг строгости
memory leak - утечка памяти
actions - действия
monads - монады
sequenced - упорядочены
statements - утверждения

monadic operator - монадный оператор
folding - свёртка
null action - завершающее действие
exception - исключение
exception handler - обработчик исключения
handle - дескриптор
channel - канал
object level - объектный уровень
functional level - функциональный уровень
showing function - функция вывода (отображения)
parser - разборщик
parser combinators - комбинаторы разборщика
derive automaticly - экземпляр класса
lexicographic order - лексикографическое упорядочение
modularity - модульность
state monad - монада состояния
monadic primitives - монадные примитивы
embedded language - встроенный язык
Domain Specific Languages - Языки Специфической Области
bignums - большие числа
default declaration - default-объявление
entities - сущности
qualified names - квалифицированные имена
abstract data types - абстрактные типы данных
representation type - тип представления
let-bound polymorphism - let-связанный полиморфизм
lambda-bound function - лямбда-связанная функция
type mismatch - несоответствие типов
monomorhipsm restriction - мономорфное ограничение
monomorphic - мономорфный
finite contiguous subset - конечное соприкасающееся подмножество
incremental array update operator - инкрементальный оператор обновления массива
bounds - размерности
association list - ассоциативный список
wavefront matrix - волновая матрица
accumulated array - аккумулированный массив
accumulating function - функция аккумуляции
loop fusion optimization - объединяющая циклы оптимизация