МИНИСТЕРСТВО ОБРАЗОВАНИЯ РФ
ИНСТИТУТ ИНФО

Н.А. Роганова

Функциональное программирование

УЧЕБНОЕ ПОСОБИЕ
ДЛЯ ВЫСШИХ УЧЕБНЫХ ЗАВЕДЕНИЙ

для студентов заочной
(дистанционной) формы обучения

ГИНФО
Москва 2002

ББК 32.97
УДК 681.3

Роганова Н.А. *Функциональное программирование: Учебное пособие для студентов высших учебных заведений* — М.: ГИНФО, 2002. — 260 с.

Учебное пособие предназначено для студентов высших учебных заведений специальностей 351500 и 510200 и соответствует программе курса «Функциональное программирование», изучаемого в течении восьмого семестра. Данное пособие может быть полезно всем желающим углубить свое представление о возможностях современных языков программирования.

Пособие знакомит с директивным стилем программирования, используя в качестве языка программирования Haskell, который сочетает черты чисто функционального языка с возможностями объектно-ориентированного стиля программирования. Изложение начинается с рассмотрения основ функционального программирования, продолжается изложением методов написания простейших программ на языке Haskell, и завершается кратким знакомством со средствами ввода и вывода информации.

Особенностью курса является то, что рассматриваемый материал не привязывается жестко к определенной операционной системе. Приведенные в учебном пособии программы работоспособны в двух наиболее распространенных на сегодняшний день операционных системах: Windows и Linux.

Подготовлено и издается по заказу Института ИНФО

Печатается в авторской редакции

Подписано в печать 20.12.02	Сдано в производство 21.12.02
Формат бумаги 60x90/16	Бум. множ.
Усл.печ.л. 16,25 Уч.-изд.л. 17,3	Тем. план 2002 г., №1-11/02
Тираж 300	Заказ № 474

ООП ГИНФО, 109280, Москва, Автозаводская, 16

Содержание

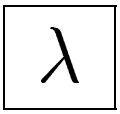
Предисловие

Общий принцип распространения знаний: новое побеждает не потому, что старики его выучивают, а потому что приходят новые поколения, которые его знают.

Книга является учебным пособием по основам программирования в так называемом функциональном стиле. В качестве основного языка программирования используется Haskell — современный язык, позволяющий проиллюстрировать все основные концепции функционального стиля программирования.

Программирование в функциональном стиле значительно отличается от более широко распространенного императивного (директивного) стиля. По историческим причинам — первые программы состояли из последовательности машинных инструкций — программы чаще всего представляют собой набор инструкций для работы с ячейками компьютерной памяти. Присваивания, изменение содержимого ячеек, передача управления на другие блоки инструкций — вот основные части программ, написанных в процедурном стиле.

Программы, написанные в чисто функциональном стиле, состоят лишь из объявлений и определений функций, они не содержат операторов присваивания и поэтому не могут иметь побочных эффектов, связанных с изменением значений переменных. Результатом работы такой программы являются вычисленные значения одной или нескольких функций.



Haskell является языком, поддерживающим полиморфизм типов и ленивые вычисления, он относится к чисто функциональным языкам программирования, отличаясь при этом от всех других функциональных языков.

Haskell базируется на лямбда исчислении (lambda calculus), поэтому греческая буква λ является его эмблемой.

Свое имя этот язык получил в честь математика и логика Хаскелла Брукс Карри (Haskell Brooks Curry), чьи работы по математической логике явились фундаментом для семейства функциональных языков (например, Curry).

При изложении материала предполагается, что читатель уже обладает некоторым опытом работы на компьютере и имеет представление о современных информационных технологиях (в частности, знаком с понятием информации и принципами ее кодирования, назначением операционных систем и их современном состоянием, умеет работать с редакторами неформатированного текста и т. д.).

В книге используются некоторые обозначения, помогающие структурировать излагаемый материал. Так, начало упражнений помечено знаком ▶, а их окончание знаком ◀. Назначение пометок λ и H на полях книги связано с определениями и вызывами функций, оно будет уточнено в следующей главе.

Глава I

Функциональная парадигма в программировании

1. Стили программирования

В сороковых годах двадцатого века были построены первые компьютеры. Они управлялись с помощью огромных плат переключателей, но как только стала возможна запись программы в память компьютера, появились и первые языки программирования.

Из-за того, что в то время использование компьютера было весьма дорого, очевидным выходом было создание языка программирования с архитектурой, настолько близкой к архитектуре компьютера, насколько возможно. Основными компонентами компьютера являются процессор и память. Поэтому программа состояла из инструкций, изменяющих память и исполняемых процессором. Наступило время *императивного стиля программирования.*

А в математике, уже на протяжении нескольких столетий, активно использовалось понятие функции. С помощью функций можно выразить связь между параметрами (аргументами, входными данными) и результатом (выводом) определенного процесса. В каждом вычислении результат определенным образом зависит от параметров. Поэтому функции — хороший инструмент для проведения вычислений. Их использование и легло в основу *функционального стиля* программирования. Программа, написанная в таком стиле, состоит из определений одной или нескольких функций. При выполнении программы функциям передаются параметры и вычисляется их результат.

При продолжающейся тенденции удешевления компьютерного времени и увеличении расходов на оплату труда разработчика

(программиста) становится все более важной возможность описания вычислений в языке программирования терминами более близкими к «человеческому миру», чем к миру компьютеров. Функциональные языки сочетают в себе традиции математики и независимости от конкретной архитектуры компьютера.

Теоретическая основа императивных языков была разработана в тридцатых годах двадцатого века Аланом Тьюрингом (в Англии) и Джоном фон Ньюманом (в США). Теория функций, как модель для вычислений, также пришла из двадцатых и тридцатых годов. Среди основателей были М. Шонфинкель (в Германии и России), Хаскелл Карри (в Англии) и Алонсо Черч (в США).

В начале пятидесятых годов появились идеи реального использования этой теории в качестве основы для языка программирования. Язык Lisp Джона МакКарти был первым функциональным языком программирования и он оставался единственным долгие годы. Несмотря на то, что Lisp до сих пор используется, он не удовлетворяет многим современных потребностям. В 80-х появилось большое число функциональных языков программирования: ML, Scheme (модифицированный Lisp), Miranda — вот несколько подобных языков.

Практически каждый ученый того времени пытался разработать свой язык. Объединившись вместе большая группа выдающихся ученых создала новый язык, сочетающий лучшие стороны различных языков. Первые версии этого языка, названного Haskell, появились в начале 90-х, и он быстро стал популярным.

2. Фундаментальные концепции

Функциональное программирование обязано своим названием тому факту, что программы полностью состоят из функций. Характерной чертой функционального программирования является тот факт, что, несмотря на отсутствие заданного порядка вычислений, результат определен однозначно. Другими словами: значение выражения (функции) — это величина и задача компьютера упростить выражение и вычислить его значение.

Процедурная программа состоит из последовательности операторов и предложений, управляющих последовательностью их выполнения. Типичными операторами являются операторы присваивания и передачи управления, операторы ввода/вывода и специальные предложения для организации циклов. Из них можно составлять фрагменты программ и подпрограммы. В основе такого

программирования лежат взятие значение какой-то переменной, совершение над ним действия и сохранение нового значения с помощью оператора присваивания. Этот процесс продолжается до тех пор пока не будет получено (и, возможно, напечатано) желаемое окончательное значение.

Функциональная программа состоит из совокупности определений функций. Функции, в свою очередь, представляют собой вызовы других функций и предложений, управляющих последовательностью вызовов. Вычисления начинаются с вызова некоторой функции, которая в свою очередь вызывает функции, входящие в ее определение и т. д. в соответствии с иерархией определений и структурой условных предложений. Функции часто либо прямо, либо опосредованно вызывают сами себя (рекурсия).

Каждый вызов возвращает некоторое значение в вызвавшую его функцию, вычисление которой после этого продолжается; этот процесс повторяется до тех пор, пока запустившая вычисления функция не вернет конечный результат пользователю. «Чистое» функциональное программирование не признает присваиваний и передач управления.

Часто определяют функциональное программирование через отрицание, говоря, что функциональное программирование — это:

- *Программирование, в котором нет побочных эффектов (side effects), в том числе традиционного присваивания. Единственным «эффектом» выполнения любой функции является вычисленный результат.*

Бывают редкие исключения, например, операции ввода-вывода, которые, естественно, изменяют физическое состояние внешних устройств.

- *Поскольку нет присваивания, переменные, получив значение в начале блока вычислений, больше никогда его не меняют.*

Это похоже на роль переменной в математических формулах и в математике вообще. Таким образом, переменные — это просто сокращенная запись их значений, и на место переменных в программе всегда можно вставить сами значения).

- *Нет явного управления последовательностью выполнения операций (flow control).*

Отсутствие побочных эффектов ведет к тому, что порядок проведения вычислений становится несущественным: поскольку на вычисляемый результат не влияют никакие побочные эффекты, не важно, когда вычислять этот результат.

Практически всегда программа, написанная в функциональном стиле, бывает на порядок короче программы в стиле императивном.

2.1. Величины

В функциональном программировании, как в математике, выражение используется только для описания (или обозначения) величины. Среди типов величин в выражениях могут встречаться следующие: числа различных типов, логические величины (truth values), символы, упорядоченные множества (tuples), функции и списки. Все они будут описаны в надлежащем месте курса. Как мы далее увидим, возможно создание новых типов величин и определение операций для генерации и манипуляции этими типами.

Может быть несколько представлений одной и той же величины. Например абстрактное число сорок девять может быть представлено десятичным числом 49 или выражением $7*7$. Компьютер обычно оперирует с двоичным представлением чисел (110001).

2.2. Функции

Наиболее важным типом данных в функциональном программировании являются функции. Мы можем применить функцию к каким-либо аргументам и вывести результаты (в предположении, что результат может быть показан). Говоря языком математики, функция f — это закон соответствия, который сопоставляет каждому элементу из данного множества A ***один единственный*** элемент из множества B. Тип элементов из множества A называется исходным (source) типом, тип элементов из множества B — целевым (target) типом функции. Сокращенно мы будем записывать эту информацию как $f :: A \to B$. Например, функция $square$, ассоциирующая с каждым целым числом его квадрат, имеет следующий тип:

```
square :: Integer -> Integer
```

О функции $f :: A \to B$ говорят, что она берет аргумент из A и возвращает результат из B. Если x есть элемент из множества A,

тогда мы пишем $f(x)$ или просто $f\ x$ для указания результата применения функции f к x. Вторая форма записи не используется, когда аргумент не является простой константой или величиной. Так, следует записать $square\ (3+4)$, потому что $square\ 3+4$ означает $(square\ 3)+4$. Это связано с более высоким приоритетом применения функции к выражению по сравнению с операцией сложения.

Две функции равны, если они дают равные результаты на равных аргументах. Так $f = g$, тогда и только тогда, когда $f\ x = g\ x$ для любого x. Это определение равенства (экстенсиональности, extensionality) функций свидетельствует о такой важной вещи, как соответствие между аргументами и результатами, но ничего не говорит о том, как это соответствие описывается.

В качестве примера приведем два различных способа определения функции, увеличивающую свой аргумент в три раза:

```
three_times' x = x + x + x
three_times'' x = 3 * x
```

2.3. Композиция функций

Для получения результата программы приходится применять функции к результатам вычисления других функций. В математике этот процесс называют композицией функций. Композиция двух функций f и g обозначается $f \cdot g$ и определяется так

$$(f \cdot g)\ x = f\ (g\ x)$$

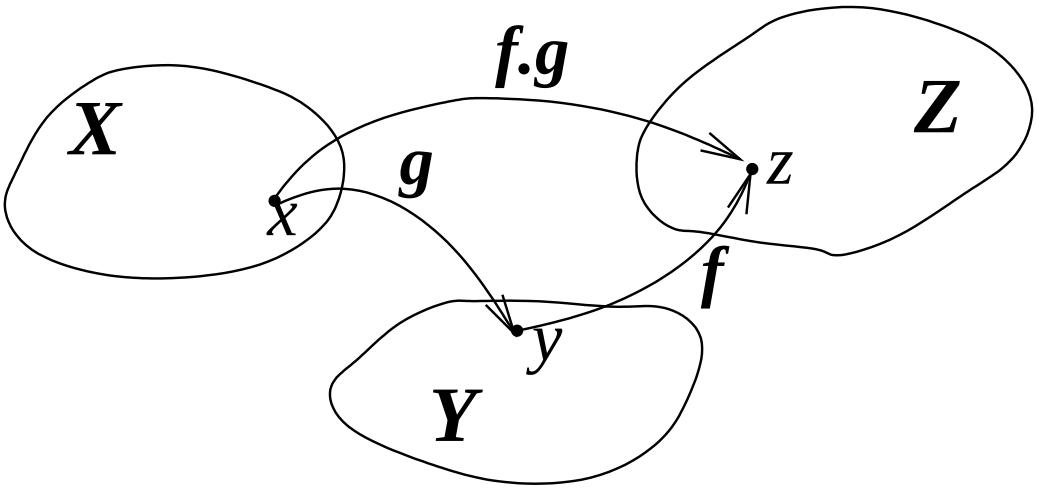


Рис. 1. Композиция функций

Другими словами, $f \cdot g$ примененная к x определяется как применение сначала функции g к x, и затем применение f к результату. Не каждая пара функций может составить композицию функций, нужно соответствие типов: мы требуем, чтобы g имело тип $g :: X \to Y$, для некоторых типов X и Y, и чтобы f имело тип $f :: Y \to Z$, для некоторого типа Z. Тогда мы получим $f \cdot g :: X \to Z$.

Композиция функций — ассоциативная операция. Это означает, что порядок расстановки скобок для всех функций f, g и h, имеющих соответствующий тип, не важен и их можно опустить:

$$(f \cdot g) \cdot h = f \cdot (g \cdot h) = f \cdot g \cdot h$$

для всех функций f, g и h, имеющих соответствующий тип. Поэтому не нужно добавлять скобки при записи последовательности композиций.

В языке Haskell, где композиция функций встречается практически в любой программе, для указания операции композиции функций употребляется символ . (точка).

3. Виды вычислений

В императивных (декларативных) языках при применении функции к аргументу последний сначала вычисляется, а затем уже передается функции. В этом случае говорят, что аргумент передается **по значению**, подразумевая при этом, что только его значение передается функции. Такое правило вычислений или механизм вызова называется **вызовом по значению**.

Преимущество вызова по значению заключается в том, что эффективная реализация проста: сначала вычисляется аргумент, а затем вызывается функция. Недостатком является избыточное вычисление, когда значение аргумента не требуется вызываемой функции. Альтернативой вызову по значению является **вызов по необходимости**, в котором все аргументы передаются функции в невычисленном виде и вычисляются только тогда, когда в них возникает необходимость внутри тела функции. Преимущество этого вызова состоит в том, что никакие затраты не пропадут попусту в случае, если значение аргумента, в конце концов, не потребуется. А недостаток — в том, что по сравнению с вызовом по значению вызов по необходимости является более дорогим, поскольку функциям передаются не значения тех или иных параметров, а невычисленные выражения.

В контексте функциональных языков можно говорить о двух видах вычисления, энергичном и ленивом, хотя возможны и другие варианты. В терминах традиционного программирования энергичное вычисление можно приблизительно соотнести с вызовом по значению, а ленивое — с механизмом вызова по необходимости. Однако между этими понятиями нет тождественного равенства.

3.1. Ленивые и энергичные вычисления

Стратегия вычислений (т. е. способ вычисления выражений), применяемая в Haskell, называется стратегией *ленивых вычислений*, также называемых *отложенными* вычислениями. При ленивом вычислении выражения (или его части) оно производится только тогда, когда его результат действительно необходим. Принцип ленивых вычислений — «*не делай ничего, пока это не потребуется*». Противоположностью этой стратегии являются *энергичные* (жадные) *вычисления*. Принцип энергичного вычисления — «*делай все, что можешь*». Другими словами, не надо заботится о том, пригодится ли в конечном случае полученный результат.

Языки, реализующие стратегию энергичных вычислений (а таковыми являются большинство императивных языков и некоторые из языков функционального программирования), не могут оперировать бесконечными структурами данных, которые широко применяются в языке Haskell.

В языке Haskell в качестве величины $\perp$ *используется функция* `undefined`.

Чтобы можно было говорить всегда, без исключений о вычислении правильно определенных выражений, вводится символ $\perp$ (bottom, основание), для обозначения неопределенной величины любого типа. В частности число ∞ (бесконечность) есть неопределенная величина типа `Integer`, $1/0$ есть неопределенная величина типа `Float`, т. е. $1/0 = \perp$. Ленивость языка Haskell приводит к тому, что все типы данных этого языка включают значение $\perp$.

3.2. Строгие функции

Говорят, что функция, которая всегда требует значение одного из своих аргументов, является **строгой** по отношению к этому аргументу. Другими словами, если $f\perp = \perp$, то говорят, что f есть

строгая, точно определенная функция, в противном случае говорят о нестрого определенной функции.

Невозможно применить операцию `+`, пока оба ее элемента не будут вычислены, поэтому она является строгой по отношению к обоим аргументам. Однако некоторые функции могут возвращать результат, и не зная значений одного или нескольких своих аргументов. Например, определенная пользователем функция вида

```
f (x, y) = if x < 10 then x else y
```

не всегда требует, чтобы была известна величина `y`. В то же время величину `x` знать необходимо, поскольку требуется определить истинность неравенства `x < 10`. Поэтому функция `f` является строгой по отношению к `x` и не строгой по отношению к `y`; это означает, что величина `x` требуется обязательно, а величина `y` — нет.

Итак, если в точке вызова вычисляются все аргументы (т. е. они передаются по значению, что соответствует энергичному вычислению), то некоторые из проделанных вычислений могут оказаться лишними. Еще большая неприятность может случиться в случае, если программа не сможет вычислить значение того аргумента, по отношению к которому функция не является строгой. Например, при вычислении подобного аргумента может произойти зацикливание и окажется невозможным вычислить аргумент за конечное время, тогда как если бы значения передавались по необходимости, то аргумент, который невозможно вычислить, мог бы и не потребоваться, и тогда программа благополучно завершилась бы за конечное время:

```
f (4, <зацикленное выражение>) = 4
```

Конечно, это не означает, что при ленивом вычислении можно всегда избежать зацикливания: примером этого может служить выражение

```
f (<зацикленное выражение>, 4)
```

которое не может завершиться за конечное время не зависимо от способа вычисления.

Рассмотрим определения нескольких функций:

```
infinity  = infinity + 1
three  x  = 3
square x  = x*x
```

Определение функции `infinity` (бесконечность) не является достаточно корректным. Но компьютер может, тем не менее, использовать его. Рассмотрим выражение `three infinity`. В случае жадных вычислений предварительно требуется вычислить аргумент функции `three`, применив правило его вычисления:

```
infinity = infinity + 1 =
```

в выражении встретилась функция `infinity`*,*
следовательно снова применяем правило ее вычисления

```
(infinity + 1) +1 =
((infinity + 1) + 1) + 1 =
     ...
```

Процесс вычисления аргумента функции `three` не сумеет завершиться, поэтому и сам вызов функции не завершиться.

При стратегии отложенных вычислений при вызове `three infinity` не потребуется вычислять аргумент, так как в любом случае ответ будет равен 3. Но при попытке вычисления `square infinity` даже ленивая стратегия не сумеет помочь в вычислении выражения, так как для того, чтобы возвести аргумент функции в квадрат, его необходимо предварительно вычислить.

В нашем примере `square` — строго определенная функция, а `three` — нестрогая. Ленивые вычисления допускают нестрогое определение функций, другие стратегии — нет.

4. Haskell как язык ФП

Дальнейшее изложение материала основано на использовании языка программирования Haskell. Haskell есть чисто функциональный язык программирования. Результат вычисления любого выражения есть инвариант, не зависящий от порядка вычисления его подвыражений, что весьма упрощает обсуждение свойств программ. Данный язык использует стратегию ленивых вычислений — подвыражения не вычисляются до тех пор, пока они явно не потребуются.

Haskell явился результатом попыток выполнения обязательств по разработке свободно-распространяемого не строгого, чисто функционального языка программирования. Haskell очень популярен в академических кругах. Многие статьи и книги, посвященные

проблемам методологии разработки программ, предполагают знание этого языка. Тем не менее, он мало распространен в производстве (хотя есть промышленные экспертные системы, написанные на Haskell). Одной из причин является плохая осведомленность об этом языке и функциональной методологии среди программистов-практиков и начинающих программистов. Этот пробел и призвано исправить данное учебное пособие.

Современное состояние языка Haskell описывается стандартом Haskell 98. Большое количество материалов, посвященных этому языку, можно найти на сайте `http://www.haskell.org`.

Дальнейшее изложение материала в основном ориентированно на использование интерпретатора Hugs 98 (сокращение от Haskell User's Gofer System) в среде ОС Linux. Hugs является самой простой реализацией языка Haskell и идеально подходит для целей обучения.

Порты Hugs 98 обеспечивают работу Haskell-кода на всех основных платформах, в том числе и для ОС Windows и Linux. Использование этого интерпретатора в Windows практически ничем не отличается от работы с ним в ОС Linux.

Открытость стандарта и возможность скачать исходные коды языка привели к появлению большого числа компиляторов с этого языка. Среди них HBC (Haskell B) — компилятор и интерпретатор, позволяет выполнять программы значительно быстрее, и GHC — оптимизирующий компилятор в машинный код и C. Дополнительную информацию о языке Haskell и разнообразное программное обеспечение для работы с ним можно найти в сети интернет по следующим адресам:

`http://haskell.org/hugs/`,
`http://haskell.org/ghc/`,
`http://www.cs.york.ac.uk/fp/nhc98/`,
`http://www.cs.chalmers.se/~augustss/hbc/hbc.html`,
`http://www.cs.chalmers.se/~augustss/hbc/hbc.html`,
`fpt://ftp.cs.chalmers.se/pub/haskell`,
`fpt://ftp.dcs.glasgow.ac.uk/pub/haskell`,
`fpt://ftp.nebula.cs.yale.edu/pub/haskell`.

Глава II
Знакомство с языком Haskell

1. Начало работы с Hugs

1.1. Команды интерпретатора

Для запуска интерпретатора выполните команду `hugs`. Интерпретатор в начале работы автоматически загружает определения наиболее часто используемых функций, размещенные в файле `Prelude.hs`, и называемом поэтому *прелюдией*.

При старте выводится некоторая справочная информация о системе и появляется приглашение для ввода команд, значением по умолчанию которого является имя загруженного модуля и знак >.

```
__   __ __  __  ____   ___
||   || ||  || ||  || ||__
||___|| ||__|| ||__||  __||
||---||         ___||
||   ||
||   || Version: December 2001

        . . .
Prelude>
```

После появления приглашения можно приступать к вводу команд. Для вычисления того или иного выражения достаточно ввести его стандартную математическую запись, после чего нажать на клавишу `Enter`:

```
Prelude> 2+3*4
14
Prelude> (2+3)*4
20
Prelude> 2.5*4
```

```
10.0
Prelude>
```

Как видим, интерпретатор различает целые числа и числа, представимые в виде десятичной дроби. Для отделения дробной части от целой используется десятичная точка. Круглые скобки используются для изменения порядка вычислений. Если хотя бы одно число в выражении не является целым, то и результат будет представлен в виде десятичной дроби. В следующей главе мы рассмотрим различные виды данных, используемых в языке Haskell.

Для завершения работы с интерпретатором Hugs служит команда `:quit` (а также ее сокращенная форма `:q`), которая также вводится после приглашения системы:

```
Prelude>:quit
```

Вид символа приглашения можно менять, задавая при старте ключ `-p`, например,

```
hugs -p'---> '
```

Запуск интерпретатора с указанным ключом приведет к установке приглашения `--->`:

```
     .  .  .
Hugs session for:
/usr/share/hugs/lib/Prelude.hs
Type :? for help
--->
```

В дальнейшем изложении мы будем использовать приглашение `--->` , что позволит отличать вывод интерпретатора от содержимого программ.

1.2. Сессии и скрипты

Последовательность интерактивных взаимодействий пользователя и компьютера называется **сессией** (session). Для выделения в тексте книги элементов интерактивных сессий на полях добавляется знак H, сигнализирующий о вводе и выводе интерпретатора Hugs.

Набор объявлений и определений функций, размещенный в отдельном файле, называется **скриптом** (script). Hugs не позволяет определять новые функции в интерактивном режиме, а только разрешает вводить выражения, использующие уже определенные функции.

Каждая программа (скрипт) на языке Haskell, называемая также модулем, размещается в отдельном файле, имеющем расширение `hs`. Символ λ на полях книги означает начало текста программы на языке Haskell, которая должна создаваться с помощью редактора plain-текста.

▶ *Упражнение II.1.1*

Создайте с помощью любого текстового редактора файл, содержащий следующие две строки:

λ
```
square :: Integer -> Integer
square x = x * x
```

Сохраните файл под именем `myprog.hs`.

◀

Наш первый скрипт содержит *объявление* (первая строка) и *определение* (вторая строка) функции `square`. Объявление функции указывает, откуда и куда действует функция, а определение задает способ преобразования аргумента `x`. Символ :: читается как «имеет тип».

Для ознакомления интерпретатора Hugs с содержимым файла используется команда `:load` (и ее сокращенная форма `:l`), например,

H
```
---> :load myprog.hs
Reading file "myprog.hs":

Hugs session for:
/usr/share/hugs/lib/Prelude.hs
myprog.hs
---> square 15
225
---> :l myprog.hs
Reading file "myprog.hs":

Hugs session for:
/usr/share/hugs/lib/Prelude.hs
myprog.hs
---> square 11
121
```

Напомним, что команды пользователя вводятся после символа приглашения (далее `--->`). В файле, предложенном интерпретатору, ошибок не обнаружено и следующая команда была указанием

вычислить значение определенной нами функции. После вычисления результата интерпретатор снова выдал приглашение к вводу команд.

При вызове команды `:load` без аргумента интерпретатор «забудет» все ранее введенные определения, кроме загруженных из файла `Prelude.hs`.

```
---> :load
```

H

```
Hugs session for:
/usr/share/hugs/lib/Prelude.hs
---> square 15
ERROR - Undefined variable "square"
```

Так как определения всех функций из прелюдии оказываются всегда загруженными по умолчанию, то их можно использовать как в любом месте программы (скрипта), так и во время сессии. Однако иногда возникает необходимость определить функцию с тем же именем, что и приведенная в файле `Prelude.hs`. В этом случае следует в начало скрипта вставить строку, которая делает «невидимой» (hide — скрывать) ту или иную функцию, например, функции `map` и `even`:

```
import Prelude hiding (map, even)
```

λ

Если в начало файла с программой поместить вышеприведенную строку, то далее можно давать свои собственные определения этих функций. Обычно необходимость в этом возникает только в учебных целях, когда начинающий программист пробует свои силы в написании стандартных функций. Определения функций, размещенные в прелюдии, можно считать эталонами функционального стиля программирования и поэтому они рекомендуются для внимательного изучения.

Для того чтобы не добавлять подобную строку в скрипт, но оставить «узнаваемое» имя у проектируемой функции, часто к имени стандартной функции добавляют символ ’ (апостроф, который не следует путать с символом обратного апострофа ‘, имеющего в языке Haskell специальное назначение).

Команда `:reload` используется без аргумента и загружает тот же файл, что и последняя команда `:load`. Ее полезно использовать после внесения изменений в файл с программой.

Команда `:type` позволяет узнать тип выражения или функции:

```
---> :type square
```

H

```
square :: Integer -> Integer
```

По умолчанию интерпретатор выводит минимальное количество информации о процессе вычисления и его результатах. Команда `:set` используемая всегда с дополнительными аргументами позволяет изменить вывод по умолчанию. Символ `+` устанавливает ту или иную опцию, а символ `-` отменяет ее. Наиболее полезны из опции `s` и `t`, первая из которых задает вывод статистической информации после каждого вычисления, а вторая отвечает за вывод типа результата вычисления, например,

H
```
---> :set +t
---> square 15
225 :: Integer
---> :set +s
---> square 15
225 :: Integer
(15 reductions, 24 cells)
---> :set -t
---> square 15
225
(15 reductions, 24 cells)
```

В качестве имени аргумента в функциях можно использовать не только `x` (что является данью математическим традициям), но и любую комбинацию латинских букв, символов подчеркивания (`_`), апострофов (`'`) и цифр, при этом первый символ в имени аргумента обязан быть прописной латинской буквой. Цифры и апострофы используют для того, чтобы указать на использование различных переменных или функций, никакой другой синтаксической нагрузки они не несут. Символ подчеркивания в основном используется для отделения одного слова от другого в именах, состоящих из нескольких слов. Другой способ сделать это — каждое слово, начиная со второго, записывать с заглавной буквы, например:

λ
```
three oneTwoThree'' = 3
```

Мы определили константную функцию, которая для любого аргумента возвращает число 3. Haskell позволяет опустить объявление функции, однако идеология функционального программирования такова, что объявление типа функции считается частью интеллектуального процесса программирования и потому считается обязательным. Принципиальное преимущество строгой типизации заключается в том, что многие программные ошибки могут быть устранены прежде, чем программа будет запущена на выполнение.

Огромное количество программных ошибок происходит из-за того, что в функциях указаны неправильные типы аргументов. Если же используется объявление типов, то при проверке типов программисту выдается сообщение об ошибках, и любое несоответствие типов в программе будет сразу обнаружено. Такая организация представляется более удобной, поскольку в противном случае ошибка проявляется уже во время выполнения программы и ее анализ и локализация требует значительных затрат времени.

Еще одним большим достоинством строгой типизации является то, что это заставляет программиста более дисциплинированно мыслить, а именно придумывать подходящий тип величинам в их определениях ранее определения их самих. Другими словами, приверженность дисциплине строгой типизации может существенно помочь в написании ясных и хорошо структурированных программ.

Будьте внимательны при создании скриптов: язык Haskell имеет двумерный синтаксис! Это означает, что интерпретатор реагирует не только на синтаксические ошибки, содержащиеся в программе, но и на расположение текста в файле.

Объявление и определение функции размещаются на отдельных строках (при желании их можно размещать и на одной строке, отделяя их точкой с запятой, что, однако, значительно ухудшает «читабельность» подобного скрипта). Если определение функции размещено на нескольких строках, то все последующие, начиная со второй, должны следовать с отступом в начале строки.

Как и в любом другом языке, в Haskell имеется несколько зарезервированных для служебных целей слов, которые нельзя использовать в качестве имен функций или переменных:

case	***class***	***data***	***else***	***if***	***in***	***infix***	***infixl***
infixr	***instance***	***let***	***of***	***primitive***	***then***	***type***	***where***

1.3. «Literate style»

Сложно дать адекватный перевод этого термина, предложенного Дональдом Кнутом, на русский язык. Основная концепция этого понятия — самодокументированность программ. Они должны быть понятны любому читателю, что достигается внедрением в тело программы достаточного количества комментариев.

Для включения в тело программы на языке Haskell однострочных комментариев используется комбинация символов `--`. Текст,

расположенный между ними и концом строки, считается комментарием.

Комбинациями символов `{-  ...  -}` ограничивают многострочные комментарии. Текст, заключенный между ними, игнорируется. Такой комментарий может быть вставлен на место любого пробела в программе. Допускаются и вложенные комментарии:

λ
```
{-
f x = {- hello -}  2*x
-}
```

В приведенном выше фрагменте программы не определена функция `f`, так как весь фрагмент является комментарием.

▶ *Упражнение II.1.2*

Измените файл `myprog.hs`, добавив в него комментарии и новые функции:

λ
```
{-
Для получения функции, возводящей целое число в четвертую
степень, воспользуемся композицией функций square
-}

square, quad :: Integer -> Integer -- объявлены две функции
square x = x*x
quad   x = (square . square) x

-- Другое определение
quad'   :: Integer -> Integer
quad' x = square (square x)
```

◀

Многие современные языки позволяют внедрить в тело программы комментарии таким образом, что при соответствующей обработке ее текста получить исчерпывающую документацию к ней. Напомним, например, о комментариях вида `/** ... */` в языке Java и соответствующей программе javadoc, обрабатывающей их.

Учитывая тот факт, что программы, написанные функциональном стиле, очень компактны, но далеко не всегда ясны неискушенному читателю в первого взгляда, понятно, что очень желательно уметь правильно документировать программу.

Программы на языке Haskell, написанные в таком стиле, должны сохраняться в файлах с расширением `lhs`. Они представляют

собой любой текст, который может как содержать, так и не содержать элементы разметки. Код на Haskell, размещаемый в таком файле, должен отделяться от остального текста пустыми строками и в первой позиции строки содержать символ `>`, после которого *обязательно* следует пробел.

▶ *Упражнение II.1.3*

Поместите следующий текст в файл `sq.lhs`, внимательно следя за размещением двух строк кода:

λ

```
Функция  squareNew, определенная на множестве
целых чисел, возвращает квадрат своего аргумента.

> squareNew :: Integer -> Integer
> squareNew(x) = x*x

Здесь использована другая форма вызова функции:
аргумент заключен в круглые скобки. Для функций,
зависящих от одной переменной, допустимы обе формы
вызова, например, squareNew(123) и squareNew 123.
```

◀

Такой файл загружается стандартным способом и обрабатывается интерпретатором аналогично файлам с расширением `hs`:

H

```
---> :load sq.lhs
Reading file "sq.lhs":

Hugs session for:
/usr/share/hugs/lib/Prelude.hs
sq.lhs
---> squareNew(123)
15129
---> squareNew 123
15129
```

Другой способ подготовки самодокументированных программ на языке Haskell предназначен для пользователей, использующих систему компьютерной верстки текста TEX. Эта система широко используется научными и инженерными работниками, так как она

позволяет полиграфически безупречно подготовить любой сложный многоязычный текст и любую математическую формулу. Нашей целью не является знакомство с этим языком разметки текста. Отметим только, что если в тексте содержится фрагмент программы, написанной на том или ином языке программирования, то его рекомендуется заключать между строками `\begin{code}` и `\end{code}`.

Hugs при просмотре подобного текста игнорирует все, что расположено вне окружения `code`, а текст, размещенный внутри него, считает программой написанной на Haskell.

Пусть файл, содержит следующий текст:

λ
```
\centerline{Максимум из двух чисел}
Функцию, определяющую максимальное из двух чисел,
можно задать с помощью конструкции
{\tt if \ldots then \ldots else}:

\begin{code}
myMax     :: Int -> Int -> Int
myMax x y  = if x >= y then x else y
\end{code}

Загрузив функцию, можно использовать ее вместо
функции {\tt max}, определенной в прелюдии.
```

Как и в предыдущем случае, файл должен иметь расширение `lhs`, например `max.lhs`:

H
```
---> :l max.lhs
Reading file "max.lhs":

Hugs session for:
/usr/share/hugs/lib/Prelude.hs
max.lhs
---> myMax 100 200
200
```

Вопросы и задания

II.1.1

Приведите причины, по которым следует придерживаться принципов строгой типизации.

II.1.2

Каково назначение файла `Prelude.hs`?

II.1.3

Определите функцию `cube`, возводящую в куб числа типа `Float`.

2. Базовые типы языка Haskell

В функциональном программировании все множество величин разделяется на организованные группы, называемые *типами*. Мы уже сталкивались с целыми (`Integer`) числами. Для работы с дробными числами чаще всего используется тип `Float`. Кроме них существуют еще другие типы чисел (например Int и $Double$), логические величины (элементы множества $Bool$), символы (элементы множества $Char$), списки, деревья и т. д. Далее мы познакомимся с этими типами, а также узнаем, как определять новые типы.

Одной из особенностей языков функционального программирования является то, что функции часто выступают в роли данных. Так функция может являться аргументом другой функции или быть результатом выполнения функции.

Каждый тип ассоциируется с определенным набором операций, которые могут не иметь смысла для других типов. Так, нельзя, например, разделить одну логическую величину на другую или не перемножить две функции.

Важным принципом многих языков программирования является то, что каждое правильно составленное выражение может быть соотнесено с некоторым типом. Кроме того, этот тип может быть выведен исключительно из типов составных частей выражения. Другими словами, значение выражения полностью определяется величинами, составляющими данное выражение.

Еще раз отметим важность строгой типизации: каждое выражение, которое не может быть ассоциировано с приемлемым типом, считается неверно сформированным и отвергается компьютером до вычисления. Например, выражение `square square 3` будет отвергнуто интерпретатором, потому что оно не является правильно сформированным.

Аналогично, если к нашему скрипту добавить функцию

```
quad :: Integer -> Integer
quad x = square square x
```

λ

то он будет отвергнут компьютером, потому что выражение `square square x` не является правильно сформированным (для исправления ошибки достаточно заключить в скобки выражение `square x`).

Другим большим достоинством строгой типизации является то, что большой диапазон ошибок — от простой орфографической ошибки до написанного кое-как определения — выявляется до вычисления.

Можно выделить две стадии анализа выражения, который проводится перед вычислением. Сначала проверяется корректность выражения с точки зрения синтаксиса. Если выражение не корректно, то выдается сигнал о *синтаксической ошибке (sintax error)*. Если синтаксических ошибок не обнаружено, то производится контроль на разумность соответствующих типов. Если выражение не прошло эту стадию, то выдается сообщение об *ошибке типа (type error)*. Только если выражение прошло обе эти стадии анализа, может начинаться процесс вычисления. Аналогичное замечание относится и к определениям, созданным в скриптах.

2.1. Функции

Напомним математическое определение функции.

> *Функция — это правило, обеспечивающее отображение объектов из множества величин, называемого областью определения функции, в объекты некоторого целевого множества, именуемого областью значений функций или диапазоном значений функции.*

Другими словами, при определении функции следует задать следующую *тройку*: откуда (X), куда (Y) и как (f). Основное требование при задании правила: каждый элемент исходного множества отображается в единственный элемент целевого множества.

Полезно представление о функции как о «черном ящике»: на вход подается элемент из области определения, на выходе получается элемент из области значений, определяемый по правилу, задающему функцию.

Различают следующие способы задания функций:

- табличный — громоздкий, требующий явного указания результата для любого элемента из области определения, фактически требуется задание множества соотношений вида $y = f(x), \forall x \in X$.

- с помощью правил, задающих способ преобразований исходного множества.

Рассмотрим функцию $sign$, определенную на множестве целых чисел следующим способом: $sign(x) = 1$ для положительных x, $sign(x) = -1$ для отрицательных x, в нуле функция принимает значение равное нулю. Для нее областью определения является множество целых чисел, областью значений — множество, состоящее из трех чисел: $\{-1; 0; 1\}$. Для любого целого числа $sign(x)$ однозначно определенно.

При табличном способе задания нам пришлось бы задать бесконечное число уравнений вида:

$$
\begin{aligned}
&\ldots \\
&sign(-3) = -1 \\
&sign(-2) = -1 \\
&sign(-1) = -1 \\
&sign(0) = 0 \\
&sign(1) = 1 \\
&sign(2) = 1 \\
&sign(3) = 1 \\
&\ldots
\end{aligned}
$$

Иначе эту функцию можно задать с помощью следующих правил:

$$
sign(x) = \begin{cases} -1, & \text{если } x < 0, \\ 0, & \text{если } x = 0, \\ 1, & \text{если } x > 0. \end{cases}
$$

Как определить такую функцию в языке Haskell, мы узнаем чуть позже.

В соответствии с математической теорией в языке Haskell различают *объявление* и *определение* функции: первое указывает, откуда и куда действует функция, а второе — по какому правилу.

Перед изучением способов задания функций, познакомимся с множествами величин, которыми чаще всего оперируют программы (являющиеся набором функций) на языке Haskell.

2.2. Числа

В языке Haskell числа записываются в стандартной десятичной системе счисления. Они могут быть как положительными, так и отрицательными. Признаком отрицательного числа является знак

минус, расположенный перед числом. Для представления целых чисел используются элементы двух базовых типов: `Integer` и `Int`. Числа типа Integer подобны обыкновенным целым, знакомым нам из математики. Их можно складывать (+), вычитать (−), умножать (∗), делить нацело (div), находить остаток от деления на другое число (rem и mod) и возводить в целую степень (^). Числа типа Int отличаются от Integer только тем, что они могут представлять числа только из некоторого ограниченного диапазона (как правило, от -2^{31} до $2^{31}-1$).

H
```
---> 12345678-123
12345555
---> 234+123456
123690
---> 234*567
132678
---> 2^10
1024
---> 120 `div` 7
17
---> 120 `mod` 7
1
---> 7*17+1
120
```

Обратите внимание, что если функция используется в качестве оператора (т. е. размещается между своими двумя аргументами), то ее имя требуется заключить в обратные апострофы.

H
```
---> div 120 7
17
---> 120 `div` 7
17
```

Результат операции деления (/) двух целых чисел не является целым числом в языке Haskell.

H
```
---> 4/2
2.0
```

Так как знак минус используется и как указатель отрицательности числа и как операция вычитания, то иногда приходится заключать отрицательные числа в скобки. Например, при вычислении остатка от деления числа 7 на отрицательное число −5 следует записать

```
---> mod 7 (-5)                                                    Н
-3
```

Если опустить скобки (записав mod $7 - 5$), то интерпретатор истолкует эту запись как $(\mathrm{mod}\ 7) - 5$, что не имеет смысла.

Использование этих двух типов целых чисел позволяет сделать процесс вычислений более эффективным. Работа с числами типа Integer происходит в сотни раз медленней, чем с числами типа Int, поэтому их следует использовать только там, где действительно нужны достаточно большие числа.

▶ *Упражнение II.2.1*

Создайте скрипт, содержащий определения двух тождественных функций (т. е. таких функций, что $f(x) = x$ для любого x из области определения функции f): `idInt`, определенную на числах типа `Int`, и `idInteger`, определенную на множестве `Integer`:

```
idInt :: Int -> Int                                                λ
idInt x = x

idInteger :: Integer -> Integer
idInteger x = x
```

Загрузив скрипт, выполните следующие вычисления:

```
--->idInteger (2^31 -1)                                            Н
2147483647
--->idInteger (2^31)
2147483648
--->idInteger (-(2^31))
-2147483648
--->idInteger (-(2^31)-1)
-2147483649
--->idInteger 2^100
1267650600228229401496703205376
--->idInteger (-2^100)
-1267650600228229401496703205376
--->idInt (2^31 -1)
2147483647
--->idInt (2^31)
-2147483648
--->idInt (-(2^31))
```

```
-2147483648
--->idInt (-(2^31)-1)
2147483647
```

Как видим, функция, определенная на множестве `Integer`, способна проводить вычисления на любых целых числах, в то время как функция, определенная на на множестве `Int`, иногда выдает «неожиданные» результаты. Для того чтобы понять, почему получены такие результаты, следует представлять себе множество величин типа `Int` в виде замкнутого кольца, в котором самое большое значение ($2^{31} - 1 = 2147483647$) «приклеивается» к наименьшему ($-2^{31} = -2147483648$). Операция сложения приводит к перемещению вдоль кольца вправо, а вычитания — влево. Поэтому, прибавляя 1 к наибольшему числу, мы получаем наименьшее.

◀

В прелюдии определены функции для работы с целыми числами.

Функция	Назначение
`gcd`	Нахождение наибольшего общего делителя двух чисел
`even`	Проверка четности числа
`odd`	Проверка нечетности числа
`fromInt`	Преобразование в число с плавающей точкой
`fromInteger`	Преобразование в число с плавающей точкой

Все функции из прелюдии загружаются при старте интерпретатора, поэтому ими можно пользоваться без загрузки какого-либо файла. Вот примеры работы этих функций:

H

```
---> gcd 45 5
5
---> even 3
False
---> even 34
True
---> odd 3
True
```

Результаты, возвращаемые функциями `even` и `odd`, — логические величины `True` (истина) и `False` (ложь), с которыми мы познакомимся чуть позже.

Для представления дробных чисел (так называемых чисел с плавающей точкой) используются типы `Float` и `Double`. Число типа `Float` не может содержать более 7 цифр после запятой, в то время как тип `Double` позволяет оперировать уже 16 знаками после запятой. Допустимы две формы записи таких чисел: с использованием десятичной точки, например, 2.3, 0.5984, и запись в экспоненциальной форме, например, `1.2e3`, `0.5e-2`. Символ `e` в такой записи означает «умножить на десять в степени». Так `1.2e3` =1200, а `0.5e-2` = 0.005.

Среди функций, определенных в прелюдии, отметим еще несколько, предназначенных для работы с числами.

Функция	Назначение
`abs`	Нахождение абсолютной величины числа
`signum`	Нахождение знака числа
`round`	Округление до целого числа

Функции, называемые *примитивами*, изначально встроены в интерпретатор. В их числе есть и функции для работы с дробными числами.

Функция	Назначение
`sqrt`	Функция извлечения квадратного корня
`sin`	Синус числа
`cos`	Косинус числа
`tan`	Тангенс числа
`asin`	Арксинус числа
`acos`	Арккосинус числа
`atan`	Арктангенс числа
`log`	Натуральный логарифм
`exp`	Экспонента (функция возведения числа `e` в степень)

Любые два числа или числовых выражения можно сравнить между собой с помощью операций `>` (больше), `>=` (больше или равно), `<` (меньше), `<=` (меньше или равно), `==` (равны) и `/=` (не равны). Результатом подобной операции также являются `True` (истина) или `False` (ложь).

```
---> 1 >= 0.5
True
---> 2 + 3 > 9
```

```
False
---> 2 + 3 == 5
True
---> 1/3 == 0.3333
False
```

2.3. Логические величины

Существуют всего две логических величины `True` и `False`. Эти две величины исчерпывают собой весь тип данных `Bool` или булевских величин (названных так по имени жившего в 19 столетии логика Джорджа Буля). Заметим, что эти имена — `True`, `False`, `Bool` — пишутся с большой буквы.

На множестве величин типа `Bool` определены стандартные логические операции: отрицание, задаваемое функцией `not`, конъюнкция (логическое умножение), задаваемая оператором `&&`, и дизъюнкция (логическое сложение) — `||`. Эти операции перечислены в порядке убывания приоритета. Как обычно, для изменения порядка вычислений используются скобки. Давайте посмотрим, как интерпретатор оперирует с логическими величинами:

H
```
---> True && False
False
---> False || True
True
---> not False
True
---> not False && False
False
---> not (False && False)
True
```

▶ *Упражнение II.2.2*

Определим функцию `myNot`, которая будет эквивалентна стандартному отрицанию. Создайте файл `bool.hs`, в который поместите следующие строки:

λ
```
myNot :: Bool -> Bool
myNot True = False
myNot False = True
```

Так как множество значений типа `Bool` состоит только из двух элементов, то нам легко было определить функцию, задав всего два уравнения.

Загрузите скрипт и проверьте, как работает заданная нами функция.

```
---> :load bool.hs
Reading file "bool.hs":

Hugs session for:
/usr/share/hugs/lib/Prelude.hs
bool.hs
---> 3 == 56-53
True
---> myNot (3 == 56-53)
False
```

Отсутствие скобок в последнем примере приведет к ошибке: так как *применение функции имеет наибольший приоритет*, то будет предпринята попытка вычислить функцию `myNot` с аргументом неподходящего типа (число вместо логической величины).

◀

Все рассматриваемые нами ранее функции зависели лишь от одного аргумента. Определим функцию, зависящую от пары аргументов и называемую «исключительное ИЛИ», результат которой равен `True`, если один из аргументов равен `True`, а другой `False`, и `False` во всех остальных случаях:

```
exOr :: (Bool, Bool) -> Bool
exOr (x, y) = (x || y) && not (x && y)
```

Конечно, мы могли, также как и в предыдущем случае, задать ее с помощью отдельного уравнения для каждой из четырех возможных пар аргументов, но приведенное решение выглядит значительно изящнее.

▶ *Упражнение II.2.3*

Создайте скрипт `exOr.hs`, в который поместите определение функции `exOr`. Убедитесь в том, что она работает корректно.

```
---> :load exOr.hs
Reading file "exOr.hs":

Hugs session for:
```

```
/usr/share/hugs/lib/Prelude.hs
exOr.hs
---> exOr (True, False)
True
---> exOr (True, True)
False
---> exOr (False, True)
True
---> exOr (False, False)
False
```

◀

На множестве логических величин также определены отношения порядка и равенства:

H
```
---> True > False
True
---> True < False
False
---> True == False
False
---> False >= False
True
```

2.4. Символы

Величины, входящие в стандартную таблицу ASCII-кодов и называемые *символами*, относятся к типу `Char`. Среди них:

- алфавитные символы
- цифры
- знаки пунктуации
- специальные символы

При записи любой символ должен быть заключен в апострофы (не следует путать с обратными апострофами, используемыми для превращения функции двух аргументов в операцию, см. стр.71).

Некоторые символы, такие как двойные кавычки, апостроф и символ \ (backslash), нельзя записать обычным способом. Их следует *экранировать* при помощи символа \:

```
кавычки       '\"'
апостроф      '\''
backslash     '\\'
```

Выражение	Тип	Значение
'x'	Char	Символ 'x'
x	...	Имя аргумента (параметра)
'3'	Char	Символ цифры '3'
3	Int	Число 3
'.'	Char	Символ знака пунктуации (точка)
.	...	Операция композиции функций
'f'	Char	Символ 'f'
f	...	Имя функции f
`f`	...	Функция f, примененная как оператор

Отметим еще несколько специальных символов, используемых для оформления выводимого на экран текста: '\n' — символ перехода на новую строку (newline) и '\t' — символ табуляции (tab). Для задания этих символов также используется экранирование с помощью \.

Две встроенные стандартные функции устанавливают соответствие между символом и его ASCII-кодом:

```
ord   :: Char -> Int
chr   :: Int  -> Char
```

Посмотрим на примеры их использования:

```
---> ord 'A'                                                    H
65
---> ord 'a'
97
---> ord 'A' - ord 'a'
-32
---> chr 51
'3'
---> ord '3'
51
```

Символы можно сравнивать между собой — они упорядочены в соответствии с их ASCII-кодом.

```
---> 'a' == 'A'                                                 H
False
---> 'b' < 'a'
False
---> 'c' >'A'
True
```

▶ *Упражнение II.2.4*

Определим функцию `capitalize`, переводящую прописные буквы в заглавные. Сначала определим разницу в кодах между заглавной и прописной буквами. Она постоянна для всех букв. В предыдущем примере мы видели, что для используемой нами кодировки символов эта разница равна отрицательному числу -32. Однако, включение в тело программы таких «магических чисел» является плохим стилем программирования. Предпочтительнее использовать следующие строки, объясняющие назначение этой величины:

λ
```
offset :: Int
offset = ord 'A' - ord 'a'
```

Целое число `offset` (смещение) получено путем вычитания кода символа 'a' из кода символа 'A'. Теперь можно определить и саму функцию:

λ
```
capitalize :: Char -> Char
capitalize ch = chr (ord ch + offset)
```

Применим эту функцию:

H
```
---> capitalize 'd'
'D'
```

◀

А теперь попробуем применить эту функцию не к прописной букве, а к какому либо иному символу:

H
```
---> capitalize 'Q'
'1'
---> capitalize '*'
'\n'
```

Для того чтобы избавится от подобных неожиданностей, следует предварительно убедиться, что аргументом является именно прописная буква.

В файле `Prelude.hs` уже определены несколько функций, проверяющих принадлежит тот или иной символ к определенной группе символов. Назначение этих функций ясно из их названий, все они в качестве исходного типа имеют тип `Char`, а в качестве целевого — тип `Bool`.

Функция	Назначение
`isDigit`	Является цифрой
`isAlpha`	Является буквой
`isUpper`	Является заглавной буквой
`isLower`	Является прописной буквой
`isSpace`	Является пробельным символом (`' '`, `'\n'`, `'\t'`)
`isAlphaNum`	Является печатным символом (буквы, цифры, пробельные символы)

Если пользователю хочется дать свое определение той или иной функции (предположим, что это некая функция `xxx`), вместо определения, приведенного в файле `Prelude.hs`, то следует в начале скрипта разместить строку вида

```
import Prelude hiding (xxx)
```

Подобная инструкция сделает "невидимым" загруженное из файла `Prelude.hs` определение функции `xxx`.

Давайте определим свои функции для работы с символами, аналогичные заданным в файле `Prelude.hs`.

▶ *Упражнение II.2.5*

Поместите в текстовый файл `char.hs` следующие определения функций:

λ

```
import Prelude hiding
                (isDigit, isUpper, isLower, isAlpha)

isDigit, isUpper, isLower, isAlpha :: Char -> Bool
isDigit c = c >= '0' && c <= '9'
isUpper c = c >= 'A' && c <= 'Z'
isLower c = c >= 'a' && c <= 'z'
isAlpha c = isUpper c || isLower c
```

Загрузите скрипт и убедитесь, что все функции определены корректно.

◀

Вернемся теперь к функции `capitalize`. Подправим ее так, чтобы она изменяла аргумент тогда и только тогда, когда он является прописной буквой. Для этого используем специальную конструкцию языка Haskell, очень похожую на стандартный условный

оператор в процедурных языках. Общий вид этой конструкции таков:
функция аргумент = **if** условие **then** значение1 **else** значение2

В нашем случае функция будет выглядеть так:

λ
```
capitalize ch = if isLower ch
  then chr (ord ch  + offset) else ch
```

Обратите внимание на отступ во второй строке, являющийся следствием двумерного синтаксиса языка (мы разместили функцию `capitalize` на двух строках). Подробнее о двумерном синтаксисе читайте на странице 61.

2.5. Списки

Списки являются коллекциями элементов, относящихся к *одному и тому* же типу. Так, говорят о списках целых чисел, списках логических величин, списках функций и т. д. Тип списка задают как тип его элементов, заключенный в квадратные скобки. Списки в свою очередь также могут быть элементами других списков. Элементы списка при записи заключают в квадратные скобки и, перечисляя их, отделяют друг от друга запятой.

В общем случае, список может содержать и нулевое число элементов. Такой список называют **пустым** списком и обозначают так `[]`.

Первый элемент списка называется **головой** списка (head). Только пустой список не имеет головы. Остальная часть списка называется **хвостом**. В отличие от головы списка, его хвост тоже является списком.

Список	Тип	Значение
`[1, 3, 5]`	`[Int]`	Список из трех целых чисел
`[1, 34, 5, 2345, 1]`	`[Int]`	Список из пяти целых чисел
`[True, False, True]`	`[Bool]`	Список логических величин
`[sin, cos, tan]`	`[Float -> Float]`	Список функций
`[[1, 2, 3], [4, 5]]`	`[[Int]]`	Список, содержащий два списка целых чисел

Кроме перечисления элементов, существуют другие способы задания списков, с которыми мы познакомимся позднее.

В файле `Prelude.hs` определены несколько стандартных функций для работы со списками. Отметим некоторые из них:

Функция	**Результат**
head	Один элемент, голова списка
tail	Хвостовой список элементов
length	Целое число, количество элементов списка
reverse	Список, записанный в обратном порядке

Посмотрим на работу этих функций на примерах.


```
---> head [1, 2, 3]
1
---> tail  [1, 2, 3]
[2,3]
---> head [[1, 2, 3], [3, 4]]
[1,2,3]
---> length [True, False, True]
3
---> reverse  [[1, 2, 3], [3, 4]]
[[3,4],[1,2,3]]
```

Операция : добавляет элемент в начало списка:


```
---> 1:[]
[1]
---> 1:2:3:[]
[1,2,3]
---> 1:(2:(3:[]))
[1,2,3]
```

▶ *Упражнение II.2.6*

Определим функцию построения списка целых чисел, аналогичную предыдущей операции. Наша функция добавляет элемент в начало списка (т. е. слева). Поместите в файл следующие строки:


```
myBuildLeft :: Int -> [Int] -> [Int]
myBuildLeft x ls = x : ls
```

Загрузите функцию и проверьте ее работоспособность.

◀

Для объединения двух списков одного и того же типа используется оператор ++, например,

H
```
---> [1, 4, 3] ++ [1, 56, 234]
[1,4,3,1,56,234]
```

▶ *Упражнение II.2.7*

Определим другую функцию построения списка целых чисел, которая добавляет новый элемент(первый аргумент функции) в конец списка (второй аргумент), т. е. *справа*.

λ
```
myBuildRight :: Int -> [Int] -> [Int]
myBuildRight x ls = ls ++ [x]
```

Загрузите функцию и проверьте ее работоспособность.

◀

Для того чтобы слить в один список элементы списка списков, используется функция `concat`:

H
```
---> concat  [[1, 2, 3], [ ], [3,4]]
[1,2,3,3,4]
```

Списки символов называют также **строками**. Для их записи обычно применяют другую форму: записывают все символы подряд, один за другим, ничем не разделяя, и заключают получившуюся строку в двойные кавычки, например,

H
```
---> ['a', 'b']
"ab"
---> head "abcd"
'a'
---> reverse "abcd"
"dcba"
```

Рассмотрим несколько функций для работы со строками, определенных в файле `Prelude.hs`.

Функция	Тип	Результат
words	`[Char] -> [[Char]]`	Разделяет строку на список слов, удаляя все пробельные символы (`' '`, `'\n'`, `'\t'`)
lines	`[Char] -> [[Char]]`	Разделяет строку на подстроки
unwords	`[[Char]] -> [Char]`	Функция, обратная `words`, объединяет список строк в одну строку, разделяя их пробелами
unlines	`[[Char]] -> [Char]`	Функция, обратная `lines`, объединяет список строк в одну строку, добавляя в конце каждого элемента символ новой строки

Посмотрите на примеры использования этих функций:

```
---> words "this is \n a string"
["this","is","a","string"]
---> lines "this is \n a string"
["this is "," a string"]
---> unwords ["this", "are", "the", "words"]
"this are the words"
---> unlines ["this are", "the words"]
"this are\nthe words\n"
```

Более подробно списки и функции для работы с ними будут рассмотрены в следующих главах.

2.6. Упорядоченные множества (tuples)

Все элементы в списке должны быть одного и того же типа. Невозможно поместить целое число, например, в список строк. Однако в процессе программирования возникает необходимость сгруппировать элементы различных типов. В этих случаях используют способ организации комбинированных типов: данные группируются в **тьюплы** (tuples) — упорядоченные множества величин, называемые также **кортежами**. Элементы тьюпла заключают в *круглые* скобки и разделяют запятыми. Ниже приведены примеры таких упорядоченных множеств.

Тьюпл	Тип	Значение
`(2.4, "cat")`	`(Float, [Char])`	Тьюпл из двух элементов (пара), состоящий из числа 2.4 и строки “cat”
`('a',True,1)`	`(Char, Bool, Int)`	Тьюпл из трех элементов: символа 'a', логической величины True и целого числа 1
`([1, 2], sqrt)`	`([Int], Float -> Float)`	Пара, состоящая из списка целых чисел и функции `sqrt` имеющей тип Float → Float
`(1, (2,3))`	`(Int, (Int, Int))`	Тьюпл из двух элементов: целое число и пара целых чисел

С тьюплами мы уже встречались при создании функции «исключительного или» `exOr` (стр. 35).

В кортеже важен порядок элементов. Тип кортежа определяется типом каждого его элемента. Так тьюплы `(1,(2,3))` и `((1,2),3)` есть различные тьюплы, имеющие соответственно следующие типы: `(Int, (Int, Int))` и `((Int, Int), Int)`.

Для кортежа из двух элементов часто используют термин *пара*, для тьюпла из трех элементов — *тройка* или 3-тьюпл и т. д. Не бывает 1-тьюплов: выражение (7) есть просто целое число, ведь мы всегда можем заключить любое выражение в круглые скобки. Тем не менее, существует 0-тьюпл: величина `( )` имеет тип `( )`.

Файл `Prelude.hs` (часто называемый прелюдией) содержит функции для работы с парами: `fst` возвращает первый элемент пары, а `snd` — второй, например,

H
```
---> fst ("cat", "dog")
"cat"
---> snd ("cat", "dog")
"dog"
```

▶ *Упражнение II.2.8*

Рассмотрим функцию `smaller`, которая берет кортеж из двух целых чисел и возвращает меньшее из них. Сохраните следующий код в файле `smaller.hs`.

```
smaller        :: (Integer, Integer) -> Integer
smaller (x, y) = if x <= y then x else y
```

λ

Ниже приведены примеры вызова определенной нами функции.

```
---> :load smaller.hs
Reading file "smaller.hs":

Hugs session for:
/usr/share/hugs/lib/Prelude.hs
smaller.hs
---> smaller (8,3)
3
---> smaller (8,38)
8
```

H

◀

▶ *Упражнение II.2.9*

Определим функцию, которая берет пару целых чисел и упорядочивает их по возрастанию. Создайте файл `smallBig.hs`, в который поместите следующий код.

```
smallBig       :: (Int, Int) -> (Int, Int)
smallBig (x, y) = if x <= y then (x, y) else (y, x)
```

λ

Загрузите функцию и примените ее к нескольким парам чисел.

◀

Кроме перечисленных типов программа на языке Haskell может содержать множество других (как стандартных, так и определяемых пользователем) типов и классов данных. Мы познакомимся с ними в следующих главах.

Вопросы и задания

II.2.1

Пусть значение `x` равно 5. Каково значение выражений `x == 3` и `x /= 3`?

II.2.2

Вычислите: а) `4e3 + 2e-2`; б) `4e3 * 2e-2`; в) `4e3 / 2e-2`.

II.2.3

Добавьте в скрипт `char.hs` определение функций `isAlphaNum` и `isSpace`.

II.2.4

Укажите типы следующих списков:

а) `[[True, False], [True, True, True], [ ], [False]]`;
б) `["alpha", "beta", "gamma"]`;
в) `[["alpha", "beta", "gamma"], ["psi", "omega"]]`.

II.2.5

Укажите выражения, имеющие следующие типы:
а) `(Bool, [Char])`, б) `([Bool], Char)`, в) `[(Bool, Char)]`.

II.2.6

Если функция `f` имеет тип `String -> String`, то каков тип следующего тьюпла `(2.4, f, 's')`?

II.2.7

Определите функцию `greater`, берущую в качестве аргумента кортеж из двух целых чисел и возвращающую большее из них.

3. Задание функций

Как уже отмечалось, интерпретатор Hugs не позволяет интерактивно определять функции, поэтому их определения помещают в файл, который затем обрабатывается интерпретатором.

В языке Haskell разделено **объявление** функции, т. е. указание ее области определения и области значений, от ее **определения** — задания правила, по которому осуществляется преобразование входной информации в выходную.

Объявление функции есть описание ее *типа*, так, фраза `(Integer, Integer)` → `Integer` описывает тип функции, которая берет пару целых (тьюпл) в качестве аргумента и вырабатывает целое в качестве результата. Такое описание типа также называется *назначением типа (type assignment)* или *сигнатурой типа (type signature)*.

Синтаксис объявления таков:

```
имя_функции :: область_определения -> область_значений
```

По некоторым данным указание объявлений функций позволяет находить до 99% ошибок, имеющихся в программе, еще на этапе синтаксического анализа интерпретатором.

Определяя функцию, мы добавляем к объявлению правило ее вычисления, например,

```
cube :: Integer -> Integer
cube n = n*n*n
```

λ

Здесь определена функция, действующая из множества Integer в множество Integer и вычисляющая куб числа. В определении функции использовался встроенный оператор умножения.

3.1. Комбинации функций

Зачастую при определении той или иной функции используют или встроенные функции и операторы (например, оператор умножения `*` в предыдущем примере) или ранее определенные функции (либо в прелюдии, либо самим программистом).

Подобные определения функций имеют следующую структуру:

- имя функции;
- имя(ена) необязательного(ых) параметра(ов);
- знак `=` (равно);
- выражение, которое может содержать параметры, стандартные функции и определенные программистом функции.

Функции с нулевым количеством аргументов называются константами, например,

```
e :: Float
e = exp 1.0
```

λ

В языке Haskell возможно определение функций с несколькими параметрами:

```
abcFormula       :: Float -> Float -> Float -> [Float]
abcFormula a b c  = [(-b + sqrt(b*b - 4.0*a*c))/(2.0*a),
                     (-b - sqrt(b*b - 4.0*a*c))/(2.0*a)
                    ]
```

λ

Функция `abcFormula` выдает список корней квадратного уравнения $ax^2 + bx + c = 0$.

```
--->  abcFormula 1.0 2.0 1.0
[-1.0,-1.0]
```

H

Функции, возвращающие булевские значения, справа от знака равенства должны содержать логическое выражение, например,

λ
```
negative, pozitive, isZero :: Integer -> Bool
negative x = x < 0
pozitive x = x > 0
isZero x   = x == 0
```

Если два определения размещены на одной строке, то их разделяют символом ; *(точка с запятой):*

```
answer = 42 ; facSix = 720
```

Обратите внимание на различия в использовании одинарного (=) и двойного (==) знака равенства: первый означает, что далее следует определение функции, в то время как второй — оператор проверки на равенство.

3.2. Частные определения

В определении рассмотренной выше функции `abcFormula` выражения `sqrt(b*b - 4.0*a*c)` и `(2.0*a)` используются дважды. Неудобство такой записи заключается не только во вводе повторяющихся фрагментов функции, но и в том, что при вычислении таких выражений теряется время: идентичные подвыражения вычисляются дважды. С целью предотвращения подобных вещей при задании формул используются, так называемые, *частные определения*, позволяющие давать имена подвыражениям в формулах. Улучшенное определение будет следующим:

λ
```
abcFormula'       :: Float -> Float -> Float -> [Float]
abcFormula' a b c  = [ (-b+d)/n, (-b-d)/n]
                     where    d = sqrt (b*b-4.0*a*c)
                              n = 2.0*a
```

Обратите внимание на двумерный синтаксис частных определений: их имена должны начинаться с одной и той же позиции разных строк. Можно расположить определения нескольких частных переменных на одной строке, разделив их в этом случае символом ; (точка с запятой).

Со статистическими опциями интерпретатора (команда `:set +s`) разница в использовании отчетливо видна:

H
```
--->  abcFormula 1.0 2.0 1.0
[-1.0,-1.0]
(80 reductions, 137 cells)
--->  abcFormula' 1.0 2.0 1.0
```

```
[-1.0,-1.0]
(64 reductions, 108 cells)
```

Ключевое слово `where` (где) — это не имя функции, а одно из зарезервированных ключевых слов (см. стр. 23). После `where` приводятся несколько определений, в данном случае определяются константы `d` и `n`. Такие константы могут использоваться в выражении, после которого указывается ключевое слово `where`. Они не могут использоваться вне этой функции — отсюда и название: *частные определения*. Может показаться странным, что `d` и `n` называют константами, ведь их значения в разных вызовах `abcFormula'` может быть разным. Но в момент вызова `abcFormula'` с заданными `a`, `b` и `c` они вычисляются лишь один раз, после чего уже не изменяются; другими словами, они являются константными выражениями в процессе вычисления функции.

Другая форма частных определений использует ключевые слова `let ... in` (пусть ... в). В ней сначала вводятся частные определения, а затем результат выражается через них. Вот как выглядит заданная в таком стиле формула для нахождения корней квадратного уравнения:

```
abcFormula''       :: Float -> Float -> Float -> [Float]
abcFormula'' a b c  =
    let d = sqrt (b*b-4.0*a*c)
        n = 2.0*a
    in [(-b+d)/n, (-b-d)/n]
```

λ

При использовании конструкции `let ...in` также необходимо внимательно следить за выравниваниями строк; все определения, относящиеся к одному и тому же уровню, должны иметь одинаковый отступ. Символ, следующий за ключевыми словами `where` или `let` задает начальную позицию для частных определений, задаваемых в этих конструкциях. Если в следующих строках отступ такой же, то частные определения рассматриваются как продолжения этих конструкций. Если же отступ уменьшился, то это рассматривается как окончание конструкции. Таким образом учитывается «окружение» (*layout*) каждой конструкции. Альтернативой этому является использование фигурных скобок и символа `;` для выделения. Так, следующие две конструкции эквивалентны:

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

λ

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

Общий вид частных определений таков:

let `имя = выражение1` **in** `выражение2`

или

`выражение2` **where** `имя = выражение1`

Выражение1 иногда называют *квалифицирующим выражением*, или *квалификатором*, а выражение2 — *результантом*. Обе эти конструкции имеют квалификатор, на который можно ссылаться с помощью имени в результанте.

Несколько частных определений могут размещаться на одной строке, разделяемые символом ; *(точка с запятой):*

```
f x = let a = 1; b = 2
          g y = exp2
      in exp1
```

Концепция использования частных (private) переменных взамен общедоступных (public) является способом инкапсуляции (сокрытия) частей программы, что позволяет скрывать внутренние детали реализации той или иной функции от других частей программы. Инкапсуляция является одной из наиболее важных идей программной инженерии. Без использования подобных технологий разработка больших программных систем была бы невозможна.

▶ *Упражнение II.3.1*

Определим функцию, подсчитывающую сумму двух последних цифр целого числа, использую два вида частных определений. Сначала воспользуемся конструкцией `where`:

λ
```
sumOfLastTwoDigits   :: Int -> Int
sumOfLastTwoDigits x = d1 + d0
      where d0    = x `mod` 10
            d1    = shift `mod` 10
            shift = x `div` 10
```

Здесь при вычислении `d0` используется параметр функции `x`, а при вычислении `d1` частная переменная `shift`.

Дадим альтернативное определение этой функции, использующее конструкцию `let ...in`:

```
sumOfLastTwoDigits'   :: Int -> Int
sumOfLastTwoDigits' x =
      let d0    = x `mod` 10
          d1    = shift `mod` 10
          shift = x `div` 10
      in d1 + d0
```

λ

Результаты применения этих функций совпадают:

```
---> sumOfLastTwoDigits 5555555
10
---> sumOfLastTwoDigits' 5555555
10
```

H

◀

3.3. Определения с альтернативами

Иногда бывает необходимо при задании функции указать, в каком случае используется та или иная часть определения. В подобных случаях используются так называемые **определения с альтернативами**. Рассмотрим в качестве подобного определения функцию, вычисляющую абсолютную величину числа. В соответствии с математическим определением

$$abs(x) = \begin{cases} x, & \text{если } x \geqslant 0, \\ -x, & \text{если } x < 0 \end{cases}$$

приходиться выбрать одно правило из двух в зависимости от знака аргумента. В языке Haskell эту функцию можно определить с помощью конструкции `if ... then ... else` (*если ... то ... иначе*). Мы будем добавлять символ `'` к именам функций, которые уже определены в прелюдии:

```
abs' x = if x >= 0 then x else -x
```

λ

Другая форма определения с альтернативами использует конструкцию `case ... of`:

```
abs' x =  case x >= 0 of
            True   ->  x
            False  -> -x
```

λ

Заметим, что количество альтернатив в конструкции `case` не обязано равняться двум.

▶ *Упражнение II.3.2*

Определим пару функций: `digitChar` — позволяющую перевести цифру в символ, ее обозначающий, и `charValue` — осуществляющую обратную операцию. Для этого нам потребуются определенные в прелюдии функции `ord`, возвращающая ASCII-код символа, и `chr`, которая по коду определяет соответствующий ему символ. Напомним, что в таблице ASCII-кодов коды цифр расположены в порядке возрастания.

λ
```
digitChar  :: Int -> Char
digitChar n = case n >= 0 && n < 10 of
              True -> chr (n +ord '0')

charValue  :: Char -> Int
charValue c = case isDigit c of
              True -> ord c - ord '0'
```

H
```
---> digitChar 5
'5'
---> charValue '5'
5
```

◀

3.4. Охранные выражения

Рассмотренную выше функцию нахождения абсолютной величины числа можно определить также следующим образом:

λ
```
abs' x | x >= 0 =  x
       | x <  0 = -x
```

Такая форма записи предпочтительнее конструкции `if`, если число различных случаев более двух. Например, функция `sign`, которая уже упоминалась на стр. 29, может быть задана так:

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

Вызов подобным образом определенной функции осуществляется стандартным способом:

H
```
---> sign (-6)
-1
```

Определения различных случаев «охраняются» булевыми выражениями, которые поэтому называются ***охранными выражениями***. При вычислении такой функции охранные выражения проверяются последовательно одно за другим до тех пор, пока одно из них не примет значение `True`. В этом случае выражение, стоящее справа от знака `=` и будет являться значением функции. Последнее в списке охранное выражение можно заменить на величину `True` или на специальную константу `otherwise` (иначе, в противном случае), например,

```
sign'' x                                                    λ
     | x > 0      =  1
     | x == 0     =  0
     | otherwise  = -1
```

Теперь мы можем дать более полное описание формы определения функции. При определении функции следует указать

- имя функции;
- имена нуля или более параметров;
- знак `=` (равно) и выражения, *или* одно или несколько охранных выражений;
- при необходимости слово `where`, за которым следуют локальные определения.

Здесь под охранным выражением понимается знак `|`, логическое выражение, знак `=` и некоторое выражение.

Тем не менее и это описание еще не исчерпывает многообразия форм представления функции в языке Haskell ...

▶ *Упражнение II.3.3*

Определим функции, вычисляющие максимум из двух и из трех целых чисел. Функция `max`, предназначенная для нахождения максимума из двух чисел, уже определена в прелюдии, поэтому нашу функцию назовем `max'`:

```
max' :: Int -> Int -> Int                                   λ
max' x y
  | x >= y      = x
  | otherwise   = y

-- Максимум из трех чисел
maxThree :: Int -> Int -> Int -> Int
maxThree x y z
```

```
  | x >= y && x >= z    = x
  | y >= z              = y
  | otherwise           = z
```

Создайте скрипт, содержащий эти определения, загрузите его и примените к различным числам. Добавьте в него альтернативное определение функции, определяющей максимум из двух чисел:

λ
```
max'' :: Int -> Int -> Int
max'' x y
  = if x >= y then x else y
```

Определите функцию `maxThree'`, также вычисляющую максимум из трех чисел, но уже с помощью функции, находящей максимум двух чисел:

λ
```
maxThree' :: Int -> Int -> Int -> Int
maxThree' x y z = max (max x y) z
```

Это определение может быть записано с использованием операторной формы функции `max`:

λ
```
maxThree'' :: Int -> Int -> Int -> Int
maxThree'' x y z =  (x `max` y) `max` z
```

Включив отображение статистической информации в интерпретаторе, выясните, какое из определений максимума трех чисел более экономично.

◀

3.5. Сопоставление с образцом

Параметры (аргументы) функции в ее определении, подобные переменным `x` и `y` в определении `f x y = x * y`, называются *формальными* параметрами функции. При запросах функции передаются *фактические* параметры. Например, в функциональном запросе

H
```
f 17 (1 + g 6)
```

17 — фактический параметр, согласованный с `x`, и `(1 + g 6)` фактический параметр, согласованный с `y`. При запросе функции фактические параметры заменили собой формальные параметры из определения. Поэтому результат запроса — `17 * (1+ g 6)`.

Итак, фактические параметры — это *выражения*. Формальные параметры были *названиями* (именами) переменных. В большинстве языков программирования формальные параметры должны всегда быть именами переменных. В языке Haskell имеются несколько других возможностей: формальный параметр может также быть *образцом*.

Рассмотрим следующее функциональное определение, где образец используется как формальный параметр:

```
f [1, x, y] = x+y
```

Такая функция применяется лишь к спискам, содержащим три элемента, причем первый элемент должен быть равен 1. Второй и третий элемент в нем могут быть произвольными. Но эта функция не определена для более коротких или длинных списков, или списков, у которых первый элемент не равен 1. То, что функции не определены при некоторых фактических параметрах, является вполне нормальным явлением. Так функция `sqrt` не определена для отрицательных чисел, а оператор / (деление) не определен при нулевом втором параметре.

Можно определять функции с различными образцами в качестве формального параметра, например,

```
sum' [] = 0
sum' [x] = x
sum' [x, y] = x+y
sum' [x, y, z] = x+y+z
```

Эта функция может применяться к спискам с 0, 1, 2 или 3 элементами (в дальнейшем эта функция будет определена на списках произвольной длинны). При вызове функции интерпретатор *сопоставляет* фактический с формальным параметром; например запрос `sum' [3, 4]` соответствует третьей строке определения. Тройка будет соответствовать `x` в определении, а 4 — `y`.

Аналогично тому, как в определении функции невозможно появление одного и того же параметра дважды, в образцах не допускается использование повторяющихся имен. Следующее определение функции неработоспособно, так как имя переменной `x` встречается дважды:

```
member x [] = False
member x (x:ys) = True -- Ошибка: дважды используется x
member x (y:ys) = member x ys
```

Вот примеры конструкций, допустимых в качестве образцов соответствия:

- числа (например, 3);
- константы True и False;
- названия параметров (имена) (например, `x`);
- списки, в которых элементы являются также образцами (например, `[1, x, y]`);
- оператор `:` с образцами слева и справа (например `a:b`);
- оператор `+`, слева от которого расположен образец, а справа натуральное число (например, `n+1`).

Отметим, что в образцах, содержащих оператор `+`, переменные всегда считаются натуральными числами.

▶ *Упражнение II.3.4*

Создайте скрипт, в который поместите следующий код

λ
```
qq :: Int -> Int
qq (x + 4) = x - 3
```

Загрузите скрипт в интерпретатор и вызовите функцию `qq` с несколькими фактическими параметрами (скобки в последнем вызове обязательны!):

H
```
---> qq 34
27
---> qq (34+4)
31
```

Приведите пример вызова функции, в результате которого будет выведено число 100.

◀

С помощью образцов соответствия в языке Haskell определено множество функций. Например, оператор конъюнкции `&&` в файле `Prelude.hs` определен так:

λ
```
False && False = False
False && True  = False
True  && False = False
True  && True  = True
```

Мы уже познакомились с встроенным оператором `:`, который используется при построении списков. Выражение `x:y`, означающее «элемент x, размещенный перед списком y», используется в качестве образца при задании функций для работы со списками.

При использовании оператора : в образце, первый элемент списка должен размещаться первым. С использованием таких образцов можно определить две очень полезные стандартные функции:

```
head (x:y) = x
tail (x:y) = y
```


Функция `head` возвращает первый элемент списка (голову, "head"); а `tail` — все кроме первого элемента (хвост, "tail"). Эти функции могут применяться к любому списку за исключением пустого (список без элементов): у пустого списка нет ни первого элемента, ни хвоста.

В некоторых случаях удобно использовать другую форму образцов, так называемые «as-образцы». Она применяется, если имя образца предполагается использовать в правой части уравнения, задающего функцию. Например, функция, дублирующая первый элемент списка, может быть записана так:

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


Напомним, что операция : правоассоциативна. Обратите внимание, что выражение `x:xs` входит в это определение дважды: как образец (*as-pattern*) в левой стороне определения и в выражении справа. Для улучшения читабельности программы хотелось бы включать `x:xs` в подобное определение лишь один раз. Этим целям и служит применение *as-образцов* в следующей форме:

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


В других, довольно часто встречающихся ситуациях, в правой стороне определения функции используются лишь некоторые части образца соответствия. Так, в рассмотренных выше определениях функций `head` и `tail` только одна из двух частей образца участвует в вычислениях. В подобных случаях для обозначения тех частей образца, которые не используются, применяется символ `_` (подчеркивание), называемый *анонимной переменной*:

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


В левосторонней части определения подобные образцы могут встречаться несколько раз. Каждый из них обрабатывается независимо, поэтому, в противоположность формальным параметрам, они никак не связаны между собой.

▶ *Упражнение II.3.5*

Определим функцию `middle`, возвращающую средний элемент кортежа из трех величин. Так как нас не интересуют значения первого и последнего элементов, то мы и не вводим их имена:

λ
```
middle (_,x,_) = x
```

Примените эту функцию к различным данным.

◀

Охранные выражения могут относиться к любым переменным в образцах соответствия. Если нет охранного выражения, соответствующего истинному значению, то поиск образца соответствия возобновляется со следующего уравнения.

▶ *Упражнение II.3.6*

Определим функцию, сочетающую в себе образцы соответствия и охранные выражения. Функция `isOrder` проверяет, упорядочены ли первые два элемента списка чисел:

λ
```
isOrder :: [Int] -> Bool
isOrder (x1:x2:xs) | x1 <= x2 = True
isOrder _                     = False
```

Примените функцию `isOrder` к различным спискам:

H
```
---> isOrder [1,2,3]
True
---> isOrder [2,1]
False
```

◀

3.6. Определение рекурсией или индукция

Как уже отмечалось, в определении функции можно использовать различные функции, как стандартные, так и ранее определенные программистом. Кроме этого, функция может выражаться через саму себя! Такое определение называется *рекурсивным определением* (слово «рекурсия» буквально переводится как «приходящий снова»; имя функции фигурирует в ее собственном определении).

Простейший пример рекурсивной функции выглядит так:

λ
```
f x = f x
```

Здесь имя определяемой функции (`f`) встречается в выражении, расположенном справа от знака равенства. Однако отметим, что такое определение довольно бессмысленно: чтобы вычислить значение `f 3`, следует вычислить значение `f 3`, для вычисления которого также следует вычислить `f 3`, и так далее до бесконечности.

Рекурсивно определенные функции будут полезны лишь при выполнении следующих двух условий:

- *параметр рекурсивного вызова должен быть более прост (например, численно меньший, или более короткий список), чем параметр определяемой функции;*
- *имеется нерекурсивное определение для основного (базового) случая.*

Хорошим примером рекурсивной функции является следующее определение факториала:

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

λ

Условие `n == 0` является базовым; в этом случае результат может быть определен непосредственно (т. е. без рекурсивного вызова). Определение для случая `n > 0` содержит рекурсивное обращение, а именно вызов `fact (n-1)`. Параметр этого вызова (`n-1`), как и требуется, меньше чем `n`.

Другим способом отличить эти два случая (базовый и рекурсивный) является сопоставление при помощи образцов:

```
fact'  0      =  1
fact' (n+1)   = (n+1) * fact' n
```

λ

В этом примере параметр рекурсивного вызова (n) меньше, чем параметр определяемой функции (n+1).

Использование образцов соответствия перекликается с математической практикой «определения по индукции». Если рекурсивное определение разделяется на несколько различных случаев сопоставления с образцами (вместо использования охранных логических выражений), то его называют *индуктивным определением функции.*

▶ *Упражнение II.3.7*

Дадим рекурсивное определение функции `power2`, являющейся аналогом математической функции 2^n:

```
power2 :: Int -> Int
power2 n
  | n==0        = 1
  | n>0         = 2 * power2 (n-1)
```

λ

◀

Функции на списках также могут быть заданы рекурсивно. Список становится «меньше» чем другой, если он содержит меньшее количество элементов (если он короче). Изменим функцию `sum'` из предыдущего раздела так, чтобы она позволяла складывать элементы списка произвольной длины. Этого можно добиться различными способами. Обычную рекурсивную функцию (использующую охранные выражения) можно определить так:

λ
```
sum' list | list == []   = 0
         | otherwize    = head list + sum' (tail list)
```

Индуктивная форма (содержащая сопоставление с образцами) такой функции может выглядеть следующим образом:

λ
```
sum' []       = 0
sum' (hd:tl) = hd + sum' tl
```

В этом определении переменная `hd` (от слова head) соответствует голове списка, а `tl` (от tail) — хвостовому списку. При выборе имен переменных, входящих в образцы, мы руководствовались принципом «говорящих имен», позволяющим полнее отразить смысл определения.

В большинстве случаев определение, использующие образцы соответствия, понятнее и яснее, так как каждое правило, определяемое отдельным образцом, использует элементы образца сразу после его указания.

▶ *Упражнение II.3.8*

Давайте определим функцию `length'`, аналогичную стандартной функции определения длины списка. Длина пустого списка, естественно, равна 0, а длина любого списка, у которого есть хвост (т. е. непустого) на единицу больше длины хвостового списка:

λ
```
length' []       = 0
length' (hd:tl) = 1 + length' tl
```

Отметив, что в выражении, стоящем справа от знака `=`, не используется переменная `hd` (она лишь указывает на то, что у списка

есть головной элемент), заменим ее на анонимную переменную. С ее использованием функция пример вид:

```
length' []      = 0
length' (_:tl)  = 1 + length' tl
```

3.7. Снова о двумерном синтаксисе

Использование дополнительных пробелов и пустых строк позволяет обычно сделать текст программы, написанной на любом языке, более понятным для чтения и анализа. Почти в любое место программы, написанной на языке Haskell, может быть добавлено произвольное число пробелов. Так, в последнем примере мы добавили несколько пробелов, выделяя и выравнивая знак `=`. Естественно, нельзя вставлять пробелы в числа, имена функций и переменных, в ключевые слова: запись `1 7`, конечно же, означает нечто отличное от `17`.

Кроме этого, можно, исходя из этих же принципов, разрывать строки и добавлять в текст программы пустые строки. Однако двумерный синтаксис языка накладывает дополнительные ограничения на разрывы строк. Так следующие локальные определения имеют совсем различный смысл:

```
where                    where
      a = f x y                a = f x
      b = g z                  y b = g z
```

Определяя функции, руководствуйтесь следующими принципами:

- строка, расположенная *с точно таким же* отступом, как и предыдущая, рассматривается как новое определение;
- строка, имеющая *больший* отступ, чем предыдущая, рассматривается как продолжение предыдущей строки;
- строка, имеющая *меньший* отступ, означает, что продолжение предыдущих строк завершено.

Наибольший интерес представляют строки, в которых фраза с использованием ключевого слова `where` используется внутри другой `where`-фразы, например,

```
f x y = g (x + w)
        where g u = u + v
                    where v = u * u
```

```
w = 2 + y
```

Здесь `w` есть локальная переменная для функции `f`, но не для `g`. Это следует из того, что отступ строки, в которой определяется `w` меньше, чем в строке, определяющей `v`. Если же сделать отступ при определении `w` меньше, чем в строке, определяющей `g`, то интерпретатор выдаст сообщение об ошибке. Следуйте правилу:

эквивалентные определения должны иметь равные отступы.

Это означает также, что все глобальные определения функций должны иметь один и тот же отступ (например, все должны начинаться с нулевой позиции).

Вопросы и задания

II.3.1

Объясните, в чем различие в использовании конструкций `x = 3` и `x == 3`.

II.3.2

Задайте функцию `greater`, возвращающую больший из двух аргументов.

II.3.3

Определите функцию, возвращающую площадь круга с радиусом r (используйте вместо числа π число $22/7$).

II.3.4

Напишите две версии функции `numberSol`, которая берет три числа `a`, `b` и `c` и вычисляет число решений уравнения $ax^2 + bx + c = 0$ (1) используя охранные выражения; (2) комбинируя стандартные функции[1].

II.3.5

Пусть определение функции `middleNumber`, находящей среднее по величине из трех чисел, таково:

[1] Воспользуйтесь определенной в прелюдии функцией `signum`, которая является аналогом рассмотренной нами функции `sign`.

```
middleNumber :: Int -> Int -> Int -> Int
middleNumber x y z
  | between  y x z      = x
  | between  x y z      = y
  | otherwise           = z

between ::  Int -> Int -> Int -> Bool
between = between
```

Замените приведенное в последней строке скрипта фиктивное определение функции `between` на такое, чтобы функция `middleNumber` была корректно определена.

Глава III
Функции в языке Haskell

1. Полиморфизм и перегрузка функций

Некоторые функции и операторы могут работать с различными типами данных. Для большинства функций на списках (например, `length`) не важно, каков тип элементов списка. Ведь можно говорить о длине списка, состоящего из целых чисел, булевых переменных или даже из функций. Тип этой функции в языке Haskell таков:

λ
```
length :: [a] -> Int
```

Эта запись говорит о том, что аргументом функции является список, причем не важно, к какому типу принадлежат его элементы. Тип переменных указывается с помощью **переменной типа**, в данном случае переменной `a`. Имя переменной типа, в отличие от названий самих типов, таких как `Int` или `Bool` должно начинаться с *прописной* буквы.

Функция `head`, которая возвращает первый элемент непустого списка, имеет тип

λ
```
head :: [a] -> a
```

Эта функция также оперирует списками и тип элементов списка для нее не важен. Тем не менее, результат применения функции имеет тот же тип, что и все элементы списка. Полиморфные функции, аналогичные выше рассмотренным, используют только *структуру* списка.

Тип, содержащий переменные типа называется *полиморфным*, т. е. применяемый к различным типам. Функция, объявленная с полиморфным типом, называется ***полиморфной функцией***. Для самого этого явления используется термин ***полиморфизм***.

Полиморфные функции встречаются очень часто. Большинство функций в прелюдии является полиморфными функциями.

Не только функции на списках могут обладать свойством полиморфности. Простейшая полиморфная функция есть функция «тождественности» (говоря языком математики, это такая функция `f`, что `f(x) = x` для любого `x`). Назовем ее `id'` (от identify — устанавливать тождество, функция `id` определена в прелюдии):

```
id'   :: a -> a
id' x = x
```

λ

Функция `id'` может оперировать элементами любого типа. Ее можно применять к числам, логическим величинам, спискам, спискам списков и т. д.

```
---> id' 456789
456789
---> id' 3.4
3.4
---> id' [1,2]
[1,2]
---> id' [[True, True], [], [False]]
[[True,True],[],[False]]
```

Н

Вспомним данное на стр. 59 определение функции `fact`:

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

λ

При вызове с неотрицательными аргументами эта функция работает замечательно, однако если вызвать ее с аргументом, меньшим нуля, то Hugs выдаст не очень ясное сообщение об ошибке:

```
---> fact 30
265252859812191058636308480000000
---> fact 0
1
---> fact (-4)
Program error: {fact (-4)}
```

Н

Дело в том, что эта функция не определена для отрицательных аргументов. В подобных ситуациях пригодится полиморфная функция `error`, заданная в прелюдии, которая использует в качестве параметра строку текста с пояснениями:

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

λ

```
    | n <  0  = error "отрицательный аргумент!"
```

Теперь вывод становится более понятным:

H
```
---> fact (-4)
Program error: отрицательный аргумент!
```

Обсудим тип функции `error`. Она использует строку в качестве аргумента, поэтому ее тип `String -> A` для какого-то типа `A`. Из описания программы ясно, что `A = Integer`, так как только ассоциированная с этим типом данных функция `fact` является хорошо определенной: первый и второй пункты оперируют с целыми, поэтому третий пункт, должен следовать тому же. Но это невозможно, так как функция `error` возвращает строку. Если же мы ограничимся только выводом чисел, зашифровывающих наше сообщение об ошибке, т. е. определим ее тип как `String -> Integer`, то мы сильно ограничим наши возможности в выводе сообщений. Кроме этого, может потребоваться возвращение функцией `error` значений и каких-либо других типов, например, в следующем примере возвращается логическое значение:

λ
```
e1 :: Bool
e1 = error "e1"
```

Вычислим значение `e1` и убедимся в корректности определения:

H
```
---> e1
Program error: e1
```

Подобные определения возможны только потому, что функция `error` является полиморфной функцией. В ее объявлении используется переменная типа:

λ
```
error :: String -> a
```

На практике функция `error` часто используется в роли неопределенного значения ($\perp$) языка Haskell. Вызов `error` приводит к прекращению выполнения и печати соответствующего сообщения. Функция `error` также применяется для создания «заглушек» — программных компонент, которые пока еще не написаны, или величин, входящих в различные структуры данных, где величина должна присутствовать, но никогда не может быть использована.

▶ *Упражнение III.1.1*

Определим константную функцию, которая для любого своего аргумента возвращает 1:

```
const1 :: a -> Int
const1 x = 1
e2 :: Int
e2 = const1 (error "e2")
```

λ

Теперь попробуем вычислить значение `e2`:

```
---> e2
1
```

Н

Как видим, функция `const1` является нестрогой по отношению к своему аргументу, т. е. `const1` $\perp = 1$

◀

Примером полиморфной функции с нулевым числом аргументов может служить функция `undefined`, которая подобно функции `error` употребляется для обозначения неопределенной величины любого типа ($\perp$):

```
---> length [undefined, 3]
2
---> True || undefined
True
```

Н

Многие функции от нескольких переменных являются полиморфными. Напомним, что при указании типа функции, зависящей от нескольких аргументов, размещают символы `->` как между типами параметров, так и перед типом результата. Так, функция `abcFormula`, приведенная на стр. 47, берущая в качестве параметров три числа и возвращающая список чисел, имеет тип:

```
abcFormula       :: Float -> Float -> Float -> [Float]
```

λ

Одним из примеров полиморфной функции от двух аргументов является функция `map`, определенная в прелюдии. Ее первый аргумент является функцией, а второй — списком. Функция `map` применяет функцию (первый аргумент) к каждому элементу списка и размещает результаты также в списке, например,

```
---> map sqrt [1.0, 4.0, 9.0]
[1.0,2.0,3.0]
---> map even [2, 4, 3, 6, 78]
[True,True,False,True,True]
```

Н

В первом примере из каждого элемента списка был извлечен квадратный корень, а во втором элементы списка были проверены на четность. Объявление типа функции `map` выглядит так:

λ
```
map :: (a -> b) -> [a] -> [b]
```

Ее первый аргумент (в скобках) произвольная функция, про которую известно лишь то, что ее аргумент должен иметь тип, обозначаемый переменной типа `a`, а возвращает она величину типа `b`. Поэтому второй аргумент (список) должен содержать величины типа `a`, а результат — список величин типа `b`. Отметим, что скобки, в которые заключен тип функции, необходимы. Их отсутствие будет означать, что у функции не два, а *три* параметра.

Попробуем определить тип композиции функций. Рассмотрим две функции, объявленные следующим образом:

λ
```
square   :: Int -> Int
sqrt     :: Int -> Float
```

Выражения `square . square` и `sqrt . square` осмысленны и имеют следующие типы

λ
```
square . square ::  Int -> Int
sqrt . square   ::  Int -> Float
```

Однако композиция функций имеет в этих двух случаях различные типы, а именно

λ
```
(.) :: (Int ->Int)     ->  (Int -> Int) -> (Int -> Int)
```

и

λ
```
(.) :: (Int -> Float)  ->  (Int -> Int) -> (Int -> Float)
```

Как видим, операция композиции функций также является полиморфной. При объявлении ее типа используются *типовые переменные*:

λ
```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

Здесь `a`, `b` и `c` задают тип переменных. При записи типовые переменные обычно обозначаются прописными латинскими буквами.

Распространенной практикой является обозначение одним и тем же именем функций, одинаковые по сути, но применяемые к величинам различного типа. Так, оператор `+` может применяться как к целым числам (типа `Int` или `Integer`), так и к числам с плавающей точкой (`Float`). Результат этой операции имеет тот же тип, что и параметры операции, например, `Int -> Int -> Int` или `Float -> Float -> Float`. Но нельзя считать `+` действительно полиморфным оператором. Если бы тип его был `a -> a -> a`,

то это означало бы возможность складывать, например, логические величины или функции, что невозможно. Функции, являющиеся «ограниченно полиморфными» называются *перегруженными* (overloaded) функциями.

Для того чтобы иметь возможность указывать тип перегруженных функций, все типы разделяют на *классы*. ***Класс*** *— это множество типов с некоторыми общими характеристиками.* В прелюдии определенно много различных классов, среди которых

- **Num** — класс, элементы которого могут складываться, вычитаться, умножаться и делиться друг на друга (числовые типы);
- **Ord** — класс, элементы которого могут быть упорядочены (упорядоченные типы);
- **Eq** — класс, элементы которого допускают проверку на равенство (сравниваемые типы).

Оператор `+` имеет тип `Num a => a -> a -> a`. Эта запись читается так: «Оператор `+` имеет тип `a -> a -> a`, где `a` из класса `Num`». Отметим, что в такой записи используется стрелка вида `=>` (иногда в литературе используется обозначение ⇒). Не следует путать ее с одинарной стрелкой: такая двойная стрелка может использоваться в объявлении типа лишь один раз. Вот еще несколько примеров перегруженных операторов:

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

λ

Перегруженными могут быть и функции, определяемые программистом, например, функция

```
square :: Num a => a -> a
square x = x*x
```

λ

является корректно определенной, так как она использует оператор `*`, который в свою очередь также перегружен.

▶ *Упражнение III.1.2*

Определим полиморфную функцию, аналогичную `smallBig`, рассмотренной на странице 45. На этот раз функция берет два числа и выдает кортеж, содержащий их в неубывающем порядке. Аргументы функции сравниваются между собой, поэтому они должны принадлежать классу `Ord`.

Определяя функцию, воспользуемся охранными выражениями:

λ
```
newSmallBig     :: Ord a => a -> a -> (a, a)
newSmallBig x y | x <= y    = (x, y)
                | otherwise = (y, x)
```

◀

Вопросы и задания

III.1.1

Укажите тип следующий величин: 3, `even` и `even 3`; `head`, `[1, 2, 3]` и `head [1, 2, 3]`. Что произойдет, если применить полиморфную функцию к фактическому параметру?

III.1.2

Выберите правильный ответ среди указанных:

1) Полиморфная функция

   а) изменяет типы своих аргументов;
   б) комбинирует данные различных типов;
   в) может применяться к различным типам аргументов;
   г) постепенно изменяет свой вид в процессе вычисления.

2) Функция, объявленная как `[a] -> [[a]]`,

   а) преобразует символ в строку;
   б) выделяет подстроку из данной строки;
   в) возвращает полученную в качестве аргумента строку, изменяя порядок символов в ней;
   г) трансформирует строку в последовательность подстрок.

3) Функция, имеющая тип `Eq a => a -> Bool`,

   а) требует аргумент с именем a;
   б) преобразует `True` в аргументы типа `a`;
   в) является полиморфной функцией;
   г) должна быть равна `a`.

4) Если в объявлении функции `f` содержится три стрелочки, то тип `f x` будет содержать

   а) одну стрелку; б) две стрелки;
   в) три стрелки; г) четыре стрелки.

2. Операторы

2.1. Префиксная и инфиксная нотации

Вызовы функций, которые мы рассматривали выше, использовали так называемую **префиксную** нотацию. В этом случае имя функции предшествует ее аргументу или аргументам. Однако из элементарной математики нам хорошо знакомы выражения вида $1 + 3$. В них функциональный символ расположен между операндами. Функцию, использующую такую запись, называют **инфиксной функцией** или **оператором**.

В языке Haskell можно любую функцию, зависящую от двух аргументов, записать в инфиксной (операторной) форме. Для этого ее имя заключается в *обратные апострофы* и помещается между аргументами, например, вызов функции `newSmallBig` может быть сделан как в префиксной, так и в инфиксной форме

```
---> newSmallBig 31 4
(4,31)
---> 31 `newSmallBig` 4
(4,31)
```

Синтаксис языка Haskell позволяет определять операторы, т. е. такие функции от двух аргументов, при вызове которых функциональный символ размещается между аргументами. Имя оператора может состоять как из одного (например, `+`), так и из двух или более символов (`&&`). При построении имени оператора могут использоваться только следующие знаки:

```
:  #  $  %  &  *  +  -  =  .  /  \  <  >  ?  !  @  ^  |
```

Допустимы, например, следующие имена операторов:

```
+   ++   &&    ||   <=    ==    /=     .    //    $
%   @@   -*-   \/   /\    ...   <+>    ?    :->
```

Операторы, расположенные в первой строке, уже определены в прелюдии. Операторы из второй строки можно определить. Оператор, начинающийся с двоеточия (`:`) означает функцию-конструктор (подробнее такие функции будут рассмотрены в одной из следующих глав).

Имеется одиннадцать комбинаций символов, которые не могут использоваться в качестве имен операторов, так как они уже наделены специфическим смыслом в языке Haskell:

```
::    =   ..   --    @    \    |    <-    ->    ~    =>
```

Оператор определяется так же, как и префиксные функции, с учетом того, что его имя в объявлении типа и при определении его в виде префиксной функции (т. е. помещая его имя перед аргументами) *имя оператора следует заключить в круглые скобки.* Если же в определении используется инфиксная нотация, то имя оператора не следует заключать в скобки.

▶ *Упражнение III.2.1*

Определим оператор, который для двух целых чисел выдает пару, состоящую из числа, полученного при делении нацело первого числа на второе, и остатка от деления:

λ
```
(%%) :: Int -> Int -> (Int, Int)
(%%) x y = (div x y, rem x y)
```

С использованием инфиксной нотации последнее определение запишется так:

λ
```
x %% y = ( div x y, rem x y)
```

При вызове оператора можно использовать как префиксную, так и инфиксную нотацию:

H
```
---> 23 %% 4
(5,3)
---> (%%) 23 4
(5,3)
```

◀

Кроме определения оператора, аналогичного определению префиксной функции, часто указывается **приоритет** оператора и тип его **ассоциативности**.

2.2. Приоритет операторов

Для однозначного определения порядка вычисления выражения, содержащего несколько различных операторов, учитывают приоритет операторов. В первую очередь выполняются те операторы, которые имеют больший приоритет. В частности, в языке Haskell, как и математике, возведение в степень имеет более высокий приоритет, чем умножение, которое в свою очередь выполняется раньше сложения.

H
```
---> 1 + 3 ^ 4 * 2
163
```

Приоритет	Операторы
9	`.` и `!!`
8	`^`, `^^` и `**`
7	`*`, `/`, `:%`, `%`, `` `div` ``, `` `quot` ``, `` `rem` `` и `` `mod` ``
6	`+` и `-`
5	`:` и `++`
4	`==`, `/=`, `<`, `<=`, `>`, `>=`, `` `elem` `` и `` `notElem` ``
3	`&&`
2	`||`
1	`>>`, `>>=` и `=<<`
0	`$`, `$!` и `` `seq` ``

Этот вызов эквивалентен вызову `1 + ((3 ^ 4) * 2)`.

Напомним, что **оператор применения функции имеет самый старший приоритет**. Так, запись `square 3 + 4` трактуется как `(square 3) + 4`.

В языке Haskell приоритет оператора задается целым числом. Чем больше число, тем выше приоритет оператора и тем раньше он выполняется. Ниже приведена таблица приоритетов операторов, определенных в прелюдии.

2.3. Ассоциативность операторов

Когда в выражении используются операторы с равным приоритетом, то порядок вычислений задается с помощью скобок или, при их отсутствии, типом ассоциативности операторов. Операторы могут быть ассоциативны, обладать только свойством ассоциативности слева или ассоциативности справа. Операторы могут быть и неассоциативны. Напомним определение ассоциативности.

Пусть $\oplus$ некий оператор. Если выражение $x \oplus y \oplus z$ вычисляется как $(x \oplus y) \oplus z$ для любых значений x, y и z соответствующих типов, то такой оператор называется *ассоциативным слева.* Операция вычитания является операцией ассоциативной слева:

```
---> 8 - 4 - 1
3
```

Сначала было вычислено выражение $8 - 4$, равное 4, а затем уже $4 - 1$.

Если при вычислении выражение $x \oplus y \oplus z = x \oplus (y \oplus z)$ для любых значений x, y и z соответствующих типов, то такой оператор называется *ассоциативным справа.* Примером ассоциативности справа является операция возведения в степень `^`: `2 ^ 2 ^ 3` равно $2^8 = 256$ как это и принято в математике, а не $4^3 = 64$.

Другими примерами правоассоциативных операторов являются операторы `:` (поместить элемент в начало списка) и оператор `->` (указание типа функции): `A -> B -> C` означает `A -> (B -> C)`.

Оператор $\oplus$ называется *ассоциативным*, если

$$(x \oplus y) \oplus z = x \oplus (y \oplus z) = x \oplus y \oplus z$$

Примерами ассоциативных операторов являются `+` и `*`. Для таких операторов выбор порядка вычислений не имеет значения.

И, наконец, оператор $\oplus$ называется *неассоциативным*, если выражение $x \oplus y \oplus z$ не имеет смысла и всегда требуется добавление скобок для задания порядка вычисления подобного выражения. Неассоциативны все операторы сравнения, например,

H
```
--->  True == False == False
ERROR - Ambiguous use of operator "(==)" with "(==)"
--->  True == (False == False)
True
--->  (True == False) == False
True
```

Для того чтобы проверить, принадлежит ли величина `x` интервалу $(2; 6)$, не следует писать `2 < x < 6`, правильно следующее сравнение: `2 < x && x <6`.

Отметим, что **применение функций всегда ассоциативно слева**, запись `f x y` означает `(f x) y` (причины этого обсуждаются далее).

2.4. Определение операторов

Приоритет и ассоциативность операторов задаются с помощью одного из следующих ключевых слов: `infixr` для оператора ассоциативного справа, `infixl` для оператора ассоциативного слева и `infix` для неассоциативного оператора. После ключевого слова указывается число, задающее приоритет, а затем имя оператора. *Строка, задающая ассоциативность и приоритет, должна располагаться до определения оператора.*

Вот как, например, задается в прелюдии правоассоциативный оператор ^ (возведение в степень) с приоритетом равным 8:

```
infixr 8 ^
```

▶ *Упражнение III.2.2*

Определим оператор !-!, аналогичный вычитанию, но ассоциативный справа. Вызов $7-3-1$ должен обрабатываться как $7-(3-1)$. Для этого поместим в файл `sub.hs` следующий текст:

```
infixr 6  !-!

(!-!) :: Num a => a -> a -> a
(!-!) x y = x - y
```

Загрузив скрипт, можно приступать к использованию нового оператора:

```
---> 7 !-! 3 !-! 1
5
```

◀

Если ту или иную функцию, определяемую программистом, планируется использовать в качестве оператора, то подобным образом можно задать ее ассоциативность и приоритет.

▶ *Упражнение III.2.3*

Из курса математики известно понятие числа сочетаний C_n^k, часто обозначаемое также, как $\binom{n}{k}$ и вычисляемое по формуле

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

Определим функцию `choose` и эквивалентный ей оператор `!^!`, находящий число сочетаний.

Так как иногда приходится вычислять выражения, аналогичные $\binom{a+b}{c}$, то, чтобы не использовать дополнительные скобки, хотелось бы иметь приоритет у определяемого оператора меньший, чем у сложения (равный 6). С другой стороны, довольно часто используются выражения вида $\binom{a}{b} < \binom{c}{d}$, и поэтому приоритет должен быть больше приоритета операторов сравнения, который равен 4. Следовательно, самым подходящим приоритетом будет значение 5.

Так как смысл выражения `a ‘choose‘ b ‘choose‘ c` не очень понятен, то сделаем наш оператор неассоциативным. Поместите следующий скрипт в файл `choose.hs`:

λ
```
infix 5  ‘choose‘, !^!

fact n  | n == 0  = 1
        | n >  0  = n * fact (n-1)
        | n <  0  = error "отрицательный аргумент!"

choose x y = fact x / (fact y * fact (x-y))
x !^! y    = fact x / (fact y * fact (x-y))
```

Загрузите скрипт, после чего примените операторы подсчета числа сочетаний к тем или иным данным:

Н
```
---> 4 ‘choose‘ 3
4.0
---> 4 !^! 3
4.0
---> 4 + 2 !^! 3
20.0
---> 4 !^! 3 < 5 !^! 2
True
---> 5 !^! 2
10.0
```

◀

Вопросы и задания

III.2.1

Укажите, какие из приведенных ниже комбинаций символов, если они встречаются в программе на языке Haskell, являются

- зарезервированным словом или символом;
- именем функции или параметра;
- оператором;
- ничем из выше перечисленного.

```
=>      3a      a3a      ::      :=      :e      X_1      <=>
a'a     _X      ***      'a'     A       in      :-<
```

3. Карринг (currying)

3.1. Частичная параметризация

Обычно в основе процесса упрощения выражения лежит идея замены сложного, структурированного аргумента последовательностью более простых.

Для иллюстрации рассмотрим снова функцию `smaller`, определенную следующим образом

```
smaller          :: (Integer, Integer) -> Integer
smaller (x, y) = if x <= y then x else y
```


Функция `smaller` берет простой аргумент, содержащий пару целых чисел, и возвращает целое. Дадим другое определение той же самой по существу функции

```
smallerc    :: Integer -> Integer -> Integer
smallerc x y = if x <= y then x else y
```


Функция `smallerc` берет два аргумента, один за другим. Более точно, `smallerc` — функция, которая берет целое `x` как аргумент и возвращает функцию `smallerc x`. В свою очередь функция `smallerc x` берет целое `y` в качестве аргумента и возвращает целое, являющееся меньшим из `x` и `y`.

Другой пример. Пусть функция `plus` складывает два аргумента:

```
plus        ::  (Integer, Integer) -> Integer
plus (x, y)  =  x + y

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


Для каждого целого `x` функция `plusc x` добавляет `x` к своему целому аргументу. В частности, `plusc 1` — функция, которая увеличивает аргумент на 1, `plusc 0` — тождественная функция на множестве целых чисел.

Этот простой прием замены структурированных выражений последовательностью нескольких простых называется *каррингом (currying)*, или частичным применением функции, по второй части имени американского логика Haskell B. Curry, чье имя носит и язык Haskell. Также используются термины *частичное вычисление функции* и *частичная параметризация*. Применив карринг,

мы избавились от составного объекта — пары, получив функцию, зависящую от двух простых объектов (чисел).

Как уже упоминалось, операция применения функции ассоциативна слева, поэтому запись

`smallerc 3 4`	означает	`(smallerc 3) 4`
`plusc x y`	означает	`(plusc x) y`
`square square 3`	означает	`(square square) 3`

Как видим, в последнем примере необходимо добавить скобки (записав `square (square x)`), так как выражение `square square` не имеет смысла потому.

Язык Haskell позволяет передать при вызове функции лишь часть ее параметров. Если функции `plusc` получит только один параметр, например, `plusc 1`, то она останется в состоянии ожидания оставшихся параметров. Такая функция может использоваться для определения новых функций:

λ
```
successor ::  Integer -> Integer
successor =   plusc 1
```

Вызов частично примененной функции ничем не отличается от вызовов других функций:

H
```
---> successor 4
5
```

Обратите внимание на определение функции: в нем не участвует параметр. Такая форма записи является более «функциональной», чем с использованием аргумента:

λ
```
successor x = plusc 1 x
```

Отметим два преимущества частично примененных функций. Во-первых, карринг помогает уменьшить число скобок при записи выражения. Во-вторых, употребление карринговых функций позволяет сделать программу еще «функциональнее». Поясним последнее утверждение примером. Рассмотрим функцию `twice`, которая позволяет применить одну и ту же функцию дважды

λ
```
twice    :: (Integer -> Integer) -> (Integer -> Integer)
twice f x = f (f x)
```

Такое определение полностью соответствует синтаксису языка Haskell. Первый аргумент функции `twice` является функцией с объявлением типа `Integer -> Integer`, второй аргумент — целое число. Применив `twice` лишь к первому аргументу `f`, мы получим

функцию `twice f`, дважды применяющую функцию `f`. С ее помощью можно определить функцию `quad`, возводящую свой аргумент в четвертую степень, например, так

```
quad :: Integer -> Integer
quad =  twice square
```

λ

Мы получили определение функции, в котором не задействованы никакие параметры.

С другой стороны, при определении `twice` в некарринговой форме

```
twice        :: (Integer -> Integer, Integer) -> Integer
twice (f, x) = f (f x)
```

λ

уже невозможно вызвать функцию без указания аргумента, к которому функция применяется. Вместо слов «*quad, где quad = twice square*», мы должны говорить «*quad, где quad x = twice (square, x) для любого x*». Второй стиль более громоздкий.

В случае необходимости всегда можно преобразовать функцию к виду частично примененной с помощью функции `curry`, которая берет не карринговую функцию и преобразует ее к частично примененной форме. Вот ее определение, содержащееся в прелюдии

```
curry             :: ((a,b) -> c) -> (a -> b -> c)
curry f x y        = f (x,y)
```

λ

Тип заголовка в функции `curry` будет рассмотрен далее. Заметим, что `curry` сама представляет собой частично примененную функцию: она получает три аргумента, один за другим. Теперь мы можем использовать `curry f` для получения частично примененного варианта любой функции `f`. Например, `plusc = curry plus`.

Функция `uncurry` осуществляет обратное преобразование — она превращает функцию с двумя параметрами в ее некарринговую форму:

```
uncurry           :: (a -> b -> c) -> ((a,b) -> c)
uncurry f (x, y)   = f x y
```

λ

3.2. Скобки в функциональной записи

Возможность частичной параметризации позволяет по новому взглянуть на тип функции `plusc`. Функция `plusc 1` имеет тот же

тип, что и `successor`: `Integer -> Integer`. Следовательно, функция `plusc`, которая в качестве параметра получает одно целое число, должна иметь тип `Integer -> (Integer -> Integer)` (ведь ее результат — функция, аналогичная `successor`).

Предположим, что `->` ассоциативна справа и опустим скобки:

```
plusc   ::  Integer -> Integer -> Integer
```

Это объявление в точности совпадает с объявлением функции, зависящий от двух аргументов, обсуждавшуюся на странице 77.

В действительности это не есть функция с двумя параметрами. Это функция, зависящая от одного параметра и возвращающая функцию, которая также берет только один параметр, и выполняет желаемое действие. А все вместе создает иллюзию того, что исходная функция зависит от двух аргументов.

Прием представления функции от нескольких параметров, в виде функции от одного аргумента, называется **карринг**.

«Невидимый» оператор применения функции *ассоциативен слева*. Это означает, что выражение `plusc 1 2` вычисляется как `(plusc 1) 2`, что в точности соответствует типу функции `plusc` — она берет один аргумент (например, число 1) и возвращает функцию, которая применяется к следующему аргументу (2 в нашем примере).

Предположив, что применение функции ассоциативно слева, то получили бы `plusc (1 2)`, что означает применение числа 1 к числу 2 (совершенно непонятно, как это можно было бы сделать!), а затем к результату применяется функция `plusc`.

Отсюда получаем следующие правила расстановки скобок. Если имеется последовательность некоторых величин в выражении, первая из которых применяется к другим, например, `f a b c d`, то она интерпретируется как `((((f a) b) c) d)`. Если величина `a` имеет тип `A`, величина `b` — тип `B` и так далее, то тип функции `f` в этом случае таков `f :: A -> B -> C -> D -> E`, или, добавляя все скобки, `f :: A -> (B -> (C -> (D -> E)))`. Конечно, запись без скобок намного удобнее и понятнее. Ассоциативность `->` и применения функции, выбранные таким образом, позволяет применять карринг совершенно незаметно: применение функции ассоциативно слева, а операция `->` ассоциативна справа. Легко запомнить следующее правило:

> *если где-то отсутствуют скобки, то они должны размещаться так, чтобы был возможен карринг.*

Скобки рекомендуется ставить только там, где они действительно необходимы. Приведем несколько примеров.

Скобки требуются при указании типа функции, если она *берет функцию в качестве параметра* (при карринге функция выдает функцию в качестве результата). Например, в прелюдии описана функция `map`, которая применяет ту или иную функцию ко всем элементам списка:

```
---> map sqrt [1, 4, 9, 16]
[1.0,2.0,3.0,4.0]
---> map even [1, 2, 3, 4]
[False,True,False,True]
---> map (plusc 1) [1, 2, 3, 4]
[2,3,4,5]
```

Тип функции `map` таков:

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

Скобки в выражении `(a -> b)` необходимы, так как иначе получалось бы, что функция `map` зависит от трех параметров.

Скобки необходимы также, если в выражении результат выполнения одной функции передается другой, например, если требуется вычислить корень квадратный из синуса числа:

```
---> sin (pi/2)
1.0
---> sqrt (sin (pi/2))
1.0
```

В первом случае скобки необходимы, так как приоритет операции применения функции выше приоритета деления, а во втором случае без скобок получилось бы вообще бессмысленное выражение `sqrt sin (pi/2)` — попытка извлечь квадратный корень из функции.

3.3. Операторные секции

Частично параметризованные операторы допускают использование двух специальных нотаций при их записи:

- $(\oplus x)$ — оператор $\oplus$ частично параметризован переменной x в качестве правого аргумента;
- $(x\oplus)$ — оператор $\oplus$ частично параметризован переменной x в качестве левого аргумента.

Такие нотации носят название *операторные секции*. Чаще всего операторные секции используются, если требуется передать частично параметризованную функцию в качестве параметра:

H
```
---> map (2*) [1, 2, 3]
[2,4,6]
```

▶ *Упражнение III.3.1*

Рассмотрим несколько примеров функций, записанных в виде операторных секций. Создайте скрипт `sections.hs`, в который поместите следующий код:

λ
```
successor    = (+1)     -- увеличить число на 1
double       = (2*)     -- умножить число на 2
half         = (/2.0)   -- половина числа
reverse'     = (1.0/)   -- обратная величина
square       = (^2)     -- квадрат числа
twoPower     = (2^)     -- возведение два в степень
oneDigit     = (<=9)    -- число из одной цифры
isZero       = (==0)    -- число равно 0
```

Загрузите скрипт и поработайте с функциями, определенными в виде операторных секций:

H
```
---> successor 100000
100001
---> double 256
512
---> half 512
256.0
---> reverse' 2
0.5
---> square 8
64
---> twoPower 8
256
---> oneDigit 23
False
---> oneDigit 2
True
---> isZero 0
True
```

◀

Вопросы и задания

III.3.1

Определите функцию `uncurry`, которая осуществляет преобразование, обратное к действию функции `curry`.

III.3.2

Какие скобки являются лишними в следующих выражениях?

- `(plusc 3) (plusc 4 5)`
- `sqrt (3.0) + (sqrt 4.0)`
- `(+) (3) (4)`
- `(2*3) +(4*5)`
- `(2+3)*(4+5)`
- `(a -> b) -> (c -> d)`

III.3.3

Дайте определения трех функций со следующими объявлениями типа:

- `(Float -> Float) -> Float`
- `Float -> (Float -> Float)`
- `(Float -> Float) -> (Float -> Float)`

III.3.4

В языке Pascal оператор `&&` имеет тот же приоритет, что и $*$, а `||` тот же, что и $+$. Является ли это правильным?

Глава IV
Функции высшего порядка

В этой главе будет рассмотрено еще одно важное свойство функциональных языков, а именно, — возможность применения функций высшего порядка. В ранее приведенных примерах каждая функция рассматривалась как статическая (неизменная) часть кода для преобразования входных величин в выходные. Это — так называемые функции первого порядка. Концепция функций высшего порядка возникает из идеи о том, что функции должны иметь тот же статус, что и любой иной объект: одни функции могут быть входными и выходными данными других.

В этой главе мы познакомимся со стилями применения функций высшего порядка и рассмотрим ряд примеров их использования на практике.

1. Функции на списках

В функциональном программировании функции во многих аспектах ведут себя так же, как и другие типы данных, например, числа или списки. Так,

- функции имеют *тип*;
- функции могут быть результатом работы других функций;
- функции могут использоваться в качестве параметра (аргумента) других функций.

Функции, использующие другие функции в качестве параметра часто называют **функциями высшего порядка**, подчеркивая таким названием их отличие от обыкновенных функций.

Встречавшаяся нам ранее функция `map` является примером функции высшего порядка. То, как будут преобразовываться элементы списка, определяется функцией, являющейся первым аргументом `map`.

Функция `map` может быть определена следующим образом:

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

Это определение использует сравнение с образцом: функция определяется отдельно для случая, когда второй аргумент является пустым списком, и для случая списка, состоящего из первого элемента `x` и остатка (хвоста) `xs`. Функция рекурсивна: в случае непустого списка функция `map` вызывается снова. При этом ее параметр становится короче (`xs` короче, чем `x:xs`), так что в конце концов будет использована нерекурсивная часть определения.

Другой часто используемой функцией высшего порядка, применяемой к спискам, является функция `filter`. Она возвращает те элементы списка, которые удовлетворяют некоторому условию. Что это будет за условие, определяется функцией, являющейся первым аргументом функции `filter`. Следующие строки демонстрируют ее использование:

```
---> filter even [1..10]
[2,4,6,8,10]
---> filter (>10) [2, 17, 8, 12, 5]
[17,12]
```

Если элементы списка имеют тип `a`, то функция-параметр должна иметь тип `a -> Bool`. Определение `filter` также рекурсивно:

```
filter  :: (a -> Bool) -> [a] -> [a]
filter p [ ]    = [ ]
filter p (x:xs) |  p x       = x : filter p xs
                |  otherwise = filter p xs
```

В случае, когда список не пуст (т. е. представим в виде `x:xs`), возможны два варианта: либо первый элемент `x` удовлетворяет условию `p`, либо нет. Если удовлетворяет, то `x` помещен в результирующий список, а оставшаяся часть списка при помощи рекурсии будет затем также подвергнута «фильтрованию».

Давайте определим еще одну очень полезную функцию высшего порядка, применяемую к спискам. Сначала посмотрим на определения трех, казалось бы, таких разных функций, как `sum` (вычисляющей сумму всех элементов списка чисел), `product` (произведение списка чисел) и `and` (которая возвращает `True`, если все элементы списка, содержащего логические величины, равны `True`):

λ
```
sum     [ ]    = 0
sum     (x:xs) = x + sum  xs

product [ ]    = 1
product (x:xs) = x * product xs

and     [ ]    = True
and     (x:xs) = x && and xs
```

Структура всех этих трех определений одинакова. Различия лишь в *значениях, возвращаемых на пустых списках* (`1`, `0`, `True`) и *операторах, используемых для присоединения первого элемента* к результату рекурсивного обращения (`+`, `*` или `&&`).

Передавая эти две переменные как параметры, можно определить одну из самых полезных функций высшего порядка:

λ
```
foldr            :: ( a -> b -> b) -> b -> [a] -> b
foldr f e [ ]    = e
foldr f e (x:xs) = x `f` foldr f e xs
```

Теперь предыдущие три функции могут быть определены при помощи частичной параметризации одной и той же функции:

λ
```
sum     = foldr (+)     0
product = foldr (*)     1
and     = foldr (&&)    True
```

Функция `foldr` (произносится как «fold right»), называемая в русскоязычной литературе **сверткой справа**, может быть использована для определения множества других функций, оперирующих списками, и поэтому ее определение включено в файл `Prelude.hs`.

▶ *Упражнение IV.1.1*

Поместите определение функции `foldr` в с отдельный скрипт и загрузите его. Не забудьте добавить в скрипт следующую строку, делающую «невидимым» каноническое определение функции `foldr`:

```
import Prelude hiding (foldr)
```

Посмотрите на результаты применения функции:

H
```
---> foldr (+) 0 [1, 2, 3, 4]
10
---> foldr (*) 1 [1, 2, 3, 4]
```

```
24
---> foldr (&&) True [True, True, False, True]
False
```

Добавьте в скрипт определения функций `sum`, `product` и `and`, приведенные выше. Не забудьте скрыть их стандартные определения, загруженные из прелюдии. Примените полученные функции к различным спискам.

◀

Название `foldr` можно объяснить следующим образом. Предположим, что у нас есть список `[a, b, c, d]`. Этот же список, используя операцию добавления элемента в начало списка можно записать в виде `a:(b:(c:(d:[ ])))`. Функция `foldr` заменяет пустой список `[ ]` на некий начальный элемент `e`, а операцию `:` на бинарный оператор, который мы обозначим символом $\oplus$, и возвращает результат. Т. е. она преобразует список

```
a : (b : (c : (d : [ ])))
```

к величине

```
a ⊕ (b ⊕ (c ⊕ (d ⊕ e)))
```

Так как операция `:` ассоциативна справа, то скобки в первом выражении можно было бы опустить. Мы, однако, оставляем их в результирующем выражении, потому что не уверены в том, что $\oplus$ ассоциативна справа.

`foldr` $(\oplus)\ e\ [x_1, x_2, \ldots, x_n] \quad = \quad x_1 \oplus (x_2 \oplus (\ldots (x_n \oplus e) \ldots))$
где $\oplus$ бинарный оператор, а `e` — стартовое значение.

Другими словами, функция `foldr` «сворачивает» список в одно значение, вставляя между всеми элементами списка данный оператор, продвигаясь при этом справа налево от заданного начального значения.

Может возникнуть потребность в аналогичной функции, осуществляющей свертку списка слева. Для иллюстрации, предположим, что мы хотим получить функцию `decimal`, которая берет список цифр $[x_0, x_1, x_2, \ldots, x_n]$ и возвращает соответствующее десятичное число, вычисляемое как

$$\texttt{decimal}\,[x_0, x_1, \ldots, x_n] = \sum_{k=0}^{n} x_k 10^{n-k}$$

Один из достаточно эффективных способов вычисления `decimal` заключается в процессе умножения каждой цифры на десять и добавления к результату следующей цифры. Например,

$$\texttt{decimal}\,[x_0, x_1, x_2] = 10 * (10 * (10 * 0 + x_0) + x_1) + x_2$$

Такой прием декомпозиции сумм степеней известен как *схема Горнера*. Определим оператор $\oplus$ как $n \oplus x = 10 * n + x$. Тогда можно переписать уравнение, приведенное выше следующим образом

$$\texttt{decimal}\,[x_0, x_1, x_2] = ((0 \oplus x_0) \oplus x_1) \oplus x_2$$

Это выражение выглядит почти как функция `foldr`, за исключением того, что скобки расставлены по другому и вычисления начинаются слева, а не справа. Фактически, это двойственный процесс: вместо обработки справа налево, здесь вычисление производится слева направо.

Рассмотренный пример объясняет причины появления второй функции свертки, называемой `foldl` (от fold left — свертка слева).

$\texttt{foldl}\ (\oplus)\ e\ [x_1, x_2, \ldots, x_n] \quad = \quad (\ldots((e \oplus x_1) \oplus x_2)\ldots) \oplus x_n$
где $\oplus$ бинарный оператор, а `e` — стартовое значение.

Заметим, что если e есть ассоциативная единица для операции $\oplus$, то `foldr` $(\oplus)$ `e` и `foldl` $(\oplus)$ `e` определяют одну и ту же функцию на конечных списках.

▶ *Упражнение IV.1.2*

Создайте файл и поместите в него определение функции `foldl`. Обратите внимание на объявление ее типа.

λ
```
import Prelude hiding (foldl)

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

Загрузите скрипт в интерпретатор и примените функцию `foldl` к различным операторам и спискам.

◀

Разберем поподробнее процесс применения этой функции к списку из трех элементов:

```
    foldl (⊕) e [ a, b, c]
=   foldl (⊕) (e ⊕ a) [b, c]
=   foldl (⊕) ((e ⊕ a) ⊕ b) [c]
=   foldl (⊕) (((e ⊕ a) ⊕ b) ⊕ c) [ ]
=   ((e ⊕ a) ⊕ b) ⊕ c
```

С помощью `foldl` функцию `decimal` можно определить так:

```
op     :: Integer -> Integer -> Integer
op a b = 10* a + b

decimal :: [Integer] -> Integer
decimal = foldl op 0
```

λ

Мы использовали в объявлении тип `Integer`, а не `Int` для того, чтобы можно было работать со списками достаточно большой длины.

▶ *Упражнение IV.1.3*

Создайте скрипт, позволяющий вычислить число, представленное списком своих цифр. Используйте для вычисления функцию `decimal`. Добавьте в скрипт определение функции `decimal'`, отличающееся только объявлением типа, для чего замените тип `Integer` на `Int`. Объясните ответ, полученный в последнем из приведенных примеров вычисления:

```
---> decimal [1, 2, 3, 4, 5, 6, 7, 8, 9,0]
1234567890
---> decimal [1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0]
123456789012345678 90
---> decimal' [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
1234567890
---> decimal' [1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0]
-350287150
```

Н

◀

Функции высшего порядка, такие как `map` и `foldr`, играют в функциональном программировании роль, которую выполняют управляющие структуры (аналогичные `for` и `while`) в процедурных языках. Однако, эти управляющие структуры являются встроенными в язык, в то время как функции можно определить самостоятельно. Это делает функциональное программирование очень гибким: имеется лишь небольшое число встроенных функций, но

программист самостоятельно может сделать все, что ему потребуется.

Вопросы и задания

IV.1.1

Объясните различия в объявлениях типа функций `foldr` и `foldl`.

IV.1.2

Почему в математике операция возведения в степень имеет правую ассоциативность

2. Итерация

Процесс *итерации* довольно широко применяется в математике. Он заключается в следующем: берется некое начальное значение и к нему применяется какая-то определенная функция до тех пор, пока не будет получен результат, удовлетворяющий заданному условию.

Итерация может быть легко определена как функция высшего порядка. В прелюдию включена подобная функция, называемая `until`, со следующим объявлением типа:

λ
```
until :: (a -> Bool)  ->  (a -> a)  ->  a  -> a
```

Как следует из объявления, функция имеет три параметра: логическое условие, которому должен удовлетворить конечный результат (функция, имеющая тип `a -> Bool`), функция, которая периодически применяется к промежуточному результату (`a -> a`), и стартовое значение, имеющее тип a. Конечный результат также имеет тип `a`. Вызов `until p f x` читается как: «*до тех пор, пока* `p` *не выполнено, применять* `f` *к элементу* `x`».

Определение функции `until` — рекурсивно. Рекурсивный и не рекурсивный случаи вызова на этот раз не используют сопоставление с образцом (patterns, паттерны), а содержат охранные выражения:

λ
```
until p f x  | p x         =   x
             | otherwise   =   until  p  f  (f  x)
```

Если начальное значение `x` сразу же удовлетворяет условию `p`, тогда оно и является результатом. В противном случае функция `f` один раз применяется к `x`. Полученный результат (`f x`) используется в роли стартового значения в рекурсивном вызове `until`.

Как и все функции высшего порядка, функция `until` может вызываться с частично параметризованными функциями в качестве параметра. Например, выражение, приведенное ниже, вычисляет наименьшую степень числа два, не превышающую 1000 (начиная с единицы и удваивая результат до тех пор, пока он не станет больше 1000):

```
--->  until (>1000) (2*) 1                                        Н
1024
```

В отличие от ранее рассмотренных рекурсивных функций, параметр рекурсивного вызова функции `until` *не становится меньше*, чем исходный параметр. Поэтому-то функция `until` не всегда способна закончить вычисление и выдать результат. При вызове функции `until (<0) (+1) 1` условие никогда не будет удовлетворено; функция `until` будет выполняться неограниченное число раз и результат не будет получен. При этом на экран компьютера не выводится никакой информации, так как выполняется бесконечное число рекурсий. Для остановки вычислений (interrupt — прерывать) в подобном случае следует нажать комбинацию клавиш `Ctrl` и `C` (нажмите клавишу `Ctrl` и, не отпуская ее, нажмите клавишу `C`):

```
--->  until (<0) (+1) 1                                           Н
```

нажатие Ctrl + C

```
{Interrupted!}

--->
```

3. Композиция функций

Мы уже обсуждали композицию функций с точки зрения математики, посмотрим теперь на эту операцию с точки зрения функций высшего порядка в функциональном программировании.

Если `f` и `g` — функции, то `f` · `g` есть математическая запись того факта, что функция `g` *выполняется после* `f`. Определим функцию `after` (после), зависящую от двух аргументов-функций `f` и `g`, которая применяет `f` первой, а затем, для получения окончательного

результата, применяет g. Наличие подобной функции позволило бы, например, давать следующие определения (мы используем ее операторную форму):

λ
```
odd         =  not  `after` even
closeToZero  = (<10) `after` abs
```

Оператор `after` может быть определен, как оператор высшего порядка следующим образом:

λ
```
infixr 8 `after`

g `after` f = h
      where h x = g (f x)
```

Напомним, что не для всех функций можно найти их композицию. Область значений функции f должна являться областью определения (доменом) для g. Другими словами, если функция f имеет тип a -> b, то функция g, соответственно, должна иметь тип b -> c. Композицией этих двух функций будет функция, которая действует из a -> c. Все это отражено в следующем объявлении типа функции after:

λ
```
after :: (b -> c) -> (a -> b) -> (a -> c)
```

Так как -> ассоциативна справа, то третьи скобки лишние, и тип after может быть описан и так:

λ
```
after :: (b -> c) -> (a -> b) -> a -> c
```

Функция after может быть рассмотрена как функция с тремя параметрами; в то же время, вспомнив о карринге, ее можно считать функцией с двумя параметрами, которая возвращает функцию (а также и как функцию с одним параметром, возвращающую частично параметризованную функцию, зависящую от одного аргумента-функции, которая в свою очередь возвращает функцию с одним параметром). Все это дает возможность определить after, как функцию с тремя параметрами:

λ
```
after g f x = g (f x)
```

При таком определении нет необходимости в локальном определении функции h с полощью ключевого слова where (хотя это, конечно же, допустимо). Возвращаясь к определению функции odd, видим, что after фактически является частично параметризованной параметрами not и even. Третий параметр при таком определении не указан: он будет передан при вызове функции odd.

Использование оператора `after` может показаться излишним, ведь функции, подобные `odd` можно было бы определить следующим образом:

```
odd x = not (even x)
```

λ

Тем не менее, композиция двух функций может служить параметром для какой-нибудь функции высокого порядка и в этом случае нет необходимости присваивать ей отдельное имя. Следующий пример демонстрирует получение списка нечетных чисел:

```
---> filter (not `after` even) [100,123,367,482,5,67]
[123,367,5,67]
```

Н

В прелюдии композиция функций определена изначально. Как уже отмечалось, она обозначается как символом . (точка), так как символа · нет на большинстве клавиатур. Итак, вы можете вычислить

```
---> filter (not . even) [100,123,367,482,5,67]
[123,367,5,67]
```

Н

Этот оператор особенно полезен в тех случаях, когда надо найти композицию трех и более функций. Программист может просто указать композицию функций, без необходимости упоминания в качестве параметров таких величин, как числа и списки. Гораздо приятнее писать

```
f = g . h . i . j . k
```

вместо

```
f x  = g (h (i (j (k x))))
```

Вопросы и задания

IV.3.1

Для каждого пункта приведите подходящую функцию:

- `(Float -> Float) -> Float`
- `Float -> (Float -> Float)`
- `(Float -> Float) -> (Float -> Float)`

IV.3.2

Является ли оператор `` `after` `` `(.)` ассоциативным?

IV.3.3

На стр. 92 утверждается, что `after` может рассматриваться как функция с одним параметром. Как это можно узнать из описания типа параметра? Дайте описание `after` в форме

```
after y = ···
```

4. Лямбда функции

Уже отмечалось (см. стр. 82), что функция, которая передается в качестве параметра другой функции, часто является частично параметризованной функцией, при этом в ее записи может использоваться или не использоваться нотация операторных секций:

H
```
---> map (plusc 5) [1 .. 10]
[6,7,8,9,10,11,12,13,14,15]
---> map (+2) [1 .. 10]
[3,4,5,6,7,8,9,10,11,12]
```

В других случаях функция, передаваемая в качестве параметра другой функции, может быть получена путем композиции других функций:

H
```
---> filter (not . even) [1 .. 10]
[1,3,5,7,9]
```

Запись `[n .. m]`, *где* `n` *и* `m` — *целые числа, является сокращенной формой записи списка целых чисел в диапазоне* `[n; m]`, *например,*

```
--> [1 .. 4]
[1,2,3,4]
```

Но в некоторых случаях очень непросто задать функцию подобным образом, например, если требуется вычислить $x^2 + 3x + 1$ для всех `x` из некоторого списка. Конечно, всегда остается возможность определить функцию, задав ее в виде локального параметра с помощью конструкции `where`:

λ
```
ys = map f [1 .. 100]
           where f x = x*x + 3*x + 1
```

Однако, если используется большое количество подобных функций, становится совсем непросто придумывать все новые и новые имена для них. В таких ситуаций используется специальная нотация, позволяющая определять функции *без явного указания их имен.*

Большинство функциональных языков программирования позволяет использовать, так называемые, лямбда-функции

(λ-функции) — функции без названия. Так как на клавиатуре нет греческой буквы λ в языке Haskell от греческой буквы λ осталась только одна палочка — `\`. Запись `\ x` следует понимать как функция от `x` (наличие пробела между символом `\` и аргументом функции не является необходимым).

Общий вид λ-функции в языке Haskell таков

```
\ образец -> выражение
```

Функция из предыдущего примера с использованием λ-нотации запишется как `\x -> x*x + 3*x + 1`. Такая запись читается следующим образом: «*функция от* `x`*, возвращающая значение* `x*x + 3*x + 1`». Обычно лямбда-функции используют, если требуется одну функцию передать в качестве аргумента другой функции, например,

```
---> map ( \ x -> x*x+3*x+1) [1 .. 10]
[5,11,19,29,41,55,71,89,109,131]
```

Тело лямбда-выражения может быть произвольным выражением, однако следует учесть, что оно не может быть рекурсивным, поскольку не существует такого связанного с функцией имени, на которое можно было бы ссылаться.

Вопросы и задания

IV.4.1

Оператор композиции функций имеет тип

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
```

Другим корректным объявлением типа этой функции является

а) `(.) :: (b -> c) -> (a -> b) -> (a -> c)`;
б) `(.) :: (a -> c) -> (b -> a) -> (b -> c)`;
в) `(.) :: (c -> a) -> (b -> c) -> (b -> a)`;
г) все выше перечисленные.

IV.4.2

Функция, имеющая тип `(a -> b) -> c` является функцией

а) низшего порядка;
б) среднего порядка;
в) высшего порядка;
г) невозможно определить.

Глава V
Числовые функции

1. Работа с целыми числами

При работе с целыми числами достаточно распространенной операцией является нахождение *остатка* (remainder) от деления нацело одного числа на другое. Так при делении нацело числа 10 на число 3 остаток равен 1. Одним из способов найти остаток является прием деления столбиком, хорошо известный из курса средней школы. К примеру при делении 345 на 12 частное равно 28 и остаток — 9:

```
 3 4 5 | 1 2
-       ---
 2 4   | 2 8
 -----
 1 0 5
-
   9 6
 -----
     9
```

В преамбуле определена функция `rem`, предназначенная для нахождения остатка:

H
```
---> 345 `rem` 12
9
```

Определение остатка от деления может потребоваться в таких задачах, как

- вычисление различных времéнных характеристик: например, если сейчас 9 часов, то через 33 часа текущее время будет равно `(9+35) `rem` 24 = 20` часов;
- определение имени дня недели: закодируем дни (0 — воскресенье, 1 — понедельник, ..., 6 — суббота), если сегодня среда (день с номером 3), то через 30 дней будет пятница (33 `rem` 7 = 5);

- выяснение возможности разделить нацело одно число на другое: число будет делится на число `n`, если остаток от деления этого числа на `n` равен нулю;
- определение количества цифр в десятичной записи числа: последняя цифра числа `x` находится по формуле `x ‘rem‘ 10`, следующая цифра равна `(x/10) ‘rem‘ 10`, третья `(x/100) ‘rem‘ 10` и так далее.

Отметим, что в языке Haskell имеется еще одна функция для нахождения остатка числа — это функция `mod`. Результаты функций `rem` и `mod` совпадают при нахождении остатков от положительных чисел, но различаются для отрицательных чисел: функция `mod` для любого числа (как положительного, так и отрицательного) находит остаток в соответствии со следующим математическим определением остатка:

> *Для любого целого числа* `m` *и положительного целого* `n` *существуют и при том единственные* `q` *и* `r`*, такие что* `m = q · n + r`*, где остаток* `r` *удовлетворяет условию* $0 \leqslant r < n$.

Другими словами, `(m ‘div‘ n) * n + m ‘mod‘ n` тождественно равно `m`. При таком определении остаток любого целого числа, найденный при помощи функции `mod` есть число неотрицательное, в отличии от функции `rem`, которая оперирует положительными числами, а лишь затем учитывает знак.

```
---> 34 ‘rem‘ 10                                                    H
4
---> 34 ‘mod‘ 10
4
---> (-34) ‘rem‘ 10
-4
---> (-34) ‘mod‘ 10
6
```

Рассмотрим еще две задачи, связанные с обработкой целых чисел. В обеих задачах используется функция `rem`.

1.1. Получение списка простых чисел

Говорят, что целое число `m` делится на целое `n`, если остаток от деления `m` на `n` равен нулю. Функция `divisible` проверяет делится ли одно число на другое:

```
divisible     :: Int -> Int -> Bool                                 λ
```

```
divisible m n  = m `rem` n == 0
```

Делителями числа называют такие целые числа, на которые исходное число делится без остатка. Функция `denominators` определяет список всех делителей заданного числа:

λ
```
denominators   :: Int -> [Int]
denominators x  = filter (divisible x) [1 .. x]
```

Заметим, что здесь функция `divisible` частично параметризована переменной `x`; функция `filter` «отфильтровывает» из списка целых чисел от 1 до `x` только те, которые являются делителями числа `x`.

Целое число называется *простым*, если оно имеет *ровно два* делителя: единицу и само себя. Функция `prime` проверяет, действительно ли в списке делителей находятся только эти два числа:

λ
```
prime   :: Int -> Bool
prime x = denominators x == [1, x]
```

И наконец, функция `primes` находит все простые числа, не превышающие данное:

λ
```
primes   :: Int -> [Int]
primes x = filter prime [1 ..x]
```

И хотя этот способ вычисления списка простых чисел, не превышающих данное, не является наиболее эффективным, он имеет неоспоримое преимущество — все функции в нем являются простым переложением математических определений на язык Haskell.

1.2. Определение дня недели

Каким днем недели будет последний день текущего года? А в какой день недели вы родились? Для ответа на подобные вопросы давайте определим функцию `day`, которая по заданному дню месяца, месяцу и году будет выдавать день недели, на который он приходится.

H
```
---> day 31 12 2002
"Tuesday"
```

Если уже известен номер дня недели, то, основываясь на предложенной выше схеме кодирования, функцию `day` легко написать:

λ
```
day d m y = weekday (daynumber d m y)
```

```
weekday 0 = "Sunday"    -- Воскресенье
weekday 1 = "Monday"    -- Понедельник
weekday 2 = "Tuesday"   -- Вторник
weekday 3 = "Wednesday" -- Среда
weekday 4 = "Thursday"  -- Четверг
weekday 5 = "Friday"    -- Пятница
weekday 6 = "Saturday"  -- Суббота
```

Функция `weekday` использует семь шаблонов для определения наименования дня недели (результатом является строка, заключенная в кавычки).

Функция `daynumber` определяет число дней, прошедших с последнего воскресенья, и добавляет к нему:

- число целых лет, умноженное на 365;
- поправку на число прошедших високосных годов;
- число дней во всех уже полностью прошедших в текущем году месяцев;
- число дней, прошедших с начала месяца.

Затем находится остаток от деления на 7 полученного (огромного) числа — это и будет номер дня недели.

В нашем (григорианском) календаре, введенном папой Григорием в 1752 году, действуют следующие правила определения високосных годов (длина которых 366 дней):

- если номер года делится на 4, то год является високосным (например, 1972);
- **но**: если номер года делится на 100, то год не високосный (например, 1900);
- **но**: если номер года делится на 400, то год *является* високосным (например, 2000).

Теперь, зная, что 1 января 0 года было воскресеньем, легко полностью определить функцию `daynumber`:

```
daynumber d m y = ( (y-1)*365                                    λ
  + (y-1) `div` 4
  - (y-1) `div` 100
  + (y-1) `div` 400
                  + sum (take (m-1) (months y))
                  + d
                  ) `rem` 7
```

Вызов `take n xs` возвращает первые `n` элементов списка `xs`. Эта функция может быть определена, например, так (данное определение немного отличается от приведенного в преамбуле):

λ
```
take 0 xs = []
take (n+1) (x:xs) = x : take n xs
```

Функция `months` возвращает число дней в каждом из месяцев заданного года:

λ
```
months y = [31,feb,31,30,31,30,31,31,30,31,30,31]
     where feb  | leap y    = 29
                | otherwise = 28
```

Функция `leap` используется в охранном выражении предыдущей функции:

λ
```
leap y = divisible y 4 &&
         (not (divisible y 100)) ||  divisible y 400
```

Ее же можно определить и по-другому:

λ
```
leap y | divisible y 100   = divisible y 400
       | otherwise         = divisible y 4
```

Для того чтобы сделать полностью корректным определение функции `day`, добавим охранное выражение, не позволяющее применить эту функцию к периоду до введения григорианского календаря:

λ
```
day d m y | y >1752  = weekday (daynumber d m y)
```

Теперь вызов этой функции с последним параметром, меньшим чем 1752, приведет к сообщению об ошибке.

H
```
---> day 16 9 2002
"Monday"
---> day 16 9 120
"
Program error: Номер года меньше 1752
```

1.3. Стратегии разработки программ

При проектировании программ для этих двух примеров использовались две различные стратегии разработки. Во втором примере мы первым делом определили функцию `day` в терминах функций `weekday` и `daynumber`. При разработке функции `daynumber` мы использовали функцию `months`, которая в свою очередь использовала функцию `leap`. Такая стратегия носит название «*сверху вниз*»:

начинается разработка с наиболее важных вещей и, затем, по мере надобности, определяются дополнительные функции.

В первом же примере применялась стратегия «*снизу вверх*»: сначала была написана функция `divisible`, с ее помощью определили функцию `denominators`, затем функцию `prime` и, наконец, `primes`.

Применение той или иной стратегии никак не сказывается на окончательном результате: ведь интерпретатор ничего не знает о том, в каком порядке разрабатывались функции. Тем не менее, при разработке программ следует обращать внимание на то, какой из стилей используется.

2. Численные вычисления

2.1. Численное дифференцирование

При вычислениях, в которых участвуют числа типа `Float`, точное нахождение результата в большинстве случаев невозможно. Результат вычисления округляется с точностью до нескольких десятичных цифр после запятой.

```
---> 10.0/6.0
1.66667
```


При вычислении некоторых математических функций, таких как `sqrt`, результат также округляется. Поэтому и при разработке своих собственных функций, оперирующих числами типа `Float`, полученный результат также будет являться апроксимацией «реального» значения.

Примером этого может служить вычисление производной той или иной математической функции. Математическое определение производной f' от функции f таково:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Точное значение предела не может быть вычислено на компьютере. Однако, приближенное значение может получено с использованием достаточно малых значений h.

Операция дифференцирования есть функция высшего порядка: функция берется в качестве аргумента и функция является результатом вычисления. Определение может быть таким:

```
diff   :: (Float -> Float) -> (Float -> Float)
```

```
diff f = f'
   where f' x = (f (x + h) - f x) / h
         h    = 0.001
```

С целью получения каррингговой формы функции можно убрать вторую пару скобок в объявлении типа, так как `->` ассоциативна справа.

λ
```
diff   :: (Float -> Float) -> Float -> Float
```

Теперь функцию `diff` можно считать функцией от двух параметров: (1) функции, производную которой следует вычислить, и (2) точки, в которой значение производной должно быть подсчитано. С этой точки зрения определение теперь выглядит так:

λ
```
diff f =  (f (x + h) - f x) / h
   where h    = 0.001
```

Эти два определения абсолютно эквивалентны. Вторая версия программы предпочтительнее, так как она проще и яснее (в ней отсутствует необходимость ввода дополнительной функции `f'`. С другой стороны, первое определение подчеркивает, что `diff` может быть рассмотрена, как преобразование функции.

Функцию `diff` очень удобно использовать после частичной параметризации, как в следующем определении:

λ
```
derivative_of_sine_square = diff (square . sin)
```

Величина `h` в обоих определениях `diff` задается в предложении `where`. Тем не менее, легко переделать программу таким образом, чтобы ее можно было бы в будущем легко изменять. Наиболее гибким путь — это задать `h` в качестве параметра `diff`:

λ
```
flexDiff h f x = (f (x + h) - f x) / h
```

Задав `h` в качестве первого аргумента функции `flexDiff`, можно использовать и ее в частично параметризованной форме, чтобы получить различные версии `diff`:

λ
```
roughDiff  = flexDiff 0.01
fineDiff   = flexDiff 0.0001
superDiff  = flexDiff 0.000001
```

2.2. Вычисление квадратного корня

В прелюдии определена функция `sqrt` для вычисления квадратного корня из числа. В этом разделе будет рассмотрен процесс разработки определения функции `sqrt`, в котором не будет

использоваться стандартная функция, предназначенная для этих целей. Этот пример позволит продемонстрировать технику работы с числами типа `Float`. В следующих разделах будет рассмотрен процесс вычисления обратной функции, что позволит дать другую реализацию функции вычисления квадратного корня.

Для квадратного корня из числа x имеет место следующее утверждение:

> если y есть хорошее приближение для $\sqrt{x}$,
> то $\frac{1}{2}(y + \frac{x}{y})$ является еще лучшим приближением.

Это свойство может быть использовано для вычисления корня из числа x: возьмем 1 в качестве первого приближения и будем подправлять приближение до тех пор, пока результат нас не устроит. Величина y является хорошим приближением $\sqrt{x}$, если y^2 не очень отличается от x. Для величины $\sqrt{3}$ приближения y_0, y_1 и т.д. таковы:

$$\begin{aligned} y_0 &= &&= 1 \\ y_1 &= 0.5 * (y_0 + 3/y_0) &&= 2 \\ y_2 &= 0.5 * (y_1 + 3/y_1) &&= 1.75 \\ y_3 &= 0.5 * (y_2 + 3/y_2) &&= 1.732142857 \\ y_4 &= 0.5 * (y_3 + 3/y_3) &&= 1.732050810 \\ y_5 &= 0.5 * (y_4 + 3/y_4) &&= 1.732050807 \end{aligned}$$

Квадрат последней аппроксимации только на 10^{-18} отличается от трех.

Для процесса «подправки» начального значения удобно использовать функцию `until`, рассмотренную на стр. 90:

```
root x = until goodEnough improve 1.0
    where improve y = 0.5 * (y + x/y)
          goodEnough y = y*y ~= x
```

λ

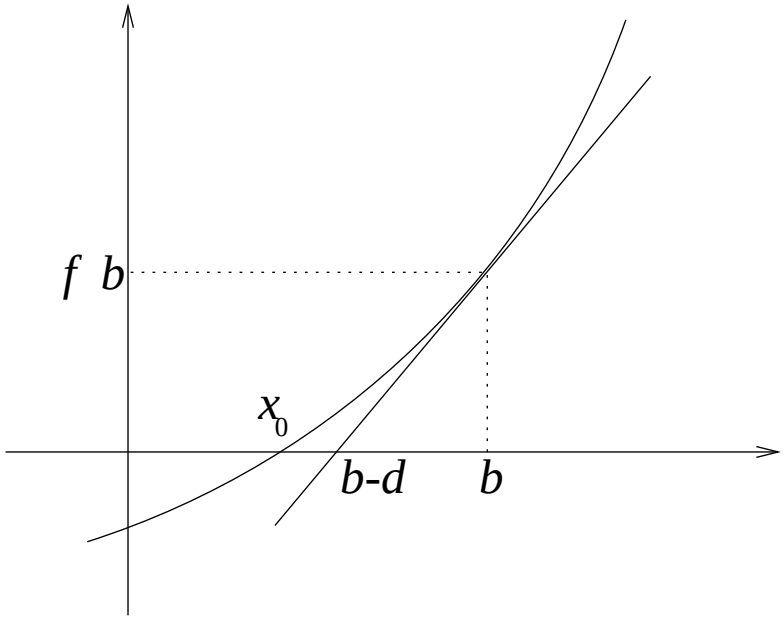
Оператор `~=` (приблизительно равно), может быть определен следующим образом:

```
infix 5 ~=

a ~= b = a - b < h && b - a < h
   where h = 0.000001
```

λ

В этом определении функция высшего порядка `until` оперирует *функциями* `improve` (улучшать) и `goodEnough` (достаточно хорошо), используя начальное значение 1.0.



Приближение методом Ньютона

Хотя в записи функции `root` сразу за `improve` располагается 1.0, функция `improve` не применяется непосредственно к числу 1.0; вместо этого оба эти объекта передаются функции `until`. С точки зрения процесса карринга, это эквивалентно следующей расстановке скобок `(((until goodEnough) improve) 1.0)`. Только заглянув в определении функции `until` можно увидеть, как `improve` применяется к 1.0.

2.3. Нули функции

Другой вычислительной проблемой, которая может быть решена с помощью итерации, является нахождение нулей функции.

Рассмотрим функцию f, корень x_0 которой требуется найти. Возьмем b в качестве первого приближения для нуля функции. Тогда касательная, проведенная к функции f в точке b пересекает ось x в точке, являющейся лучшим приближением нуля, чем b (см. рисунок).

Точка пересечения касательной с осью x находится на расстоянии `d` от первого приближения `b`. Величина `d` может быть вычислена следующим образом. Тангенс угла наклона касательной к функции `f` в точке `b` равен $f'(b)$. С другой стороны, это значение равно `f(b)/d`, поэтому $d = f(b)/f'(b)$.

Данное замечание позволяет нам сделать следующий вывод: если `b` есть первое приближение нуля функции `f`, то $b - f(b)/f'(b)$ —

есть лучшее приближение. Этот метод известен, как «метод Ньютона» (этот метод не всегда работает для функций с локальными экстремумами: вы можете «ходить вокруг» нужного корня и никогда в него не попасть).

Для получения функции, вычисляющей нули этом методом, также можно использовать функцию `until`. В качестве параметра будем использовать приближение (`improve`), задаваемое приведенной выше формулой (для чего воспользуемся ранее определенной функцией `diff`). Условием окончания итераций будет служить достаточно малое отклонение функции от нулевого значения.

λ
```
zero f = until goodEnough improve 1.0
         where improve b    = b - f b / diff  f b
               goodEnough b = f b ~= 0.0
```

Мы выбрали 1.0 в качестве первого приближения, но с таким же успехом, это могло быть и число 17.93. Вообще, это может быть любая точка из области определения функции `f`.

2.4. Обратная функция

Ноль функции f, где $fx = x^2 - a$, равен $\sqrt{a}$. С учетом этого замечания мы можем вычислить $\sqrt{a}$ как ноль функции f. Применив функцию `zero`, `root` можно определить так:

λ
```
root a = zero f
       where f x = x*x - a
```

Кубический корень можно вычислить так:

λ
```
cubic a = zero f
       where f x = x*x*x - a
```

Аналогично, любая функция, обратная к заданной, может быть вычислена с использованием этой функции в определении функции `f`, например,

λ
```
arcsin a = zero f
       where f x = sin x - a

arccos a = zero f
       where   f x = cos x - a
```

Наличие общности в определениях этих функций является сигналом для определения функции высшего порядка, которая обобщает эти случаи (по аналогии с определением функции `foldr`, которая появилась как обобщение `sum`, `product` и `and`). В данном случае функция `inverse` является функцией высшего порядка, имеющей дополнительный параметр — функцию g, инверсию которой требуется вычислить:

λ
```
inverse   g a = zero f
      where f x = g x - a
```

Если вы случайно заметили некую закономерность, то следует попытаться определить функцию высшего порядка так, чтобы другие функции стали просто частными случаями ее применения. При этом все частные случаи следует определить через полученную функцию, которая будет частично параметризова различными параметрами:

λ
```
arcsin = inverse sin
arccos = inverse cos
ln     = inverse exp
```

Функцию `inverse` можно рассматривать как функцию с двумя параметрами (функция и `Float`) и `Float` в качестве результата, или как функцию с *одним* параметром (функция) и функцией как результатом. Это следует из эквивалентности следующих объявлений типа функции `inverse`:

λ
```
inverse   ::   (Float -> Float) -> Float -> Float
```

и

λ
```
inverse   ::   (Float -> Float) -> Float -> Float
```

(вспомним о правой ассоциативности `->`).

Функция нахождения корня, использующая метод Ньютона, может быть получена подобным путем. Рассмотрим определение функции `root`:

λ
```
root a = zero f
       where   f x = x*x - a
```

Заменив вызов функции `zero f` его определением, получим:

λ
```
root a = until goodEnough improve 1.0
      where improve b     = b - f b / diff f b
            goodEnough b  = f b ~= 0.0
            f x = x*x - a
```

В данном случае нет необходимости определять `diff` численно: производная `f` есть функция (`2*`), поэтому формула в функции `improve b` может быть упрощена:

$$
\begin{aligned}
& b - \frac{fb}{f'b} \\
&= b - \frac{b^2 - a}{2b} \\
&= b - \frac{b^2}{2b} + \frac{a}{2b} \\
&= \frac{b}{2} + \frac{a/b}{2} \\
&= 0.5 * (b + a/b)
\end{aligned}
$$

Это и есть формула, использованная нами еще на стр. 103.

Вопросы и задания

V.2.1

Дайте определение функции `cubicRoot`, которая не использует численное дифференцирование (даже косвенное, с помощью `zero`).

V.2.2

Определите функцию `inverse`, используя лямбда нотацию.

V.2.3

Объясните расположение скобок в выражении:
`(f (x+h) - f x) / h`.

Глава VI
Структуры данных

1. Списки

Списки являются одной из самых распространенных структур данных, используемых в функциональном программировании. Список — коллекция элементов *одного* типа. Тип списка задается типом элементов, содержащихся в нем.

1.1. Определение списка

Существуют несколько способов определить список: явное перечисление его элементов, конструирование списка при помощи оператора `:`, указание числовых интервалов.

Перечисление элементов списка — самый простой способ его создания. Элементы заключают в квадратные скобки и разделяют запятыми. Элементами списка могут быть как константы, так и выражения, которые будут определены в процессе вычисления, например,

```
[1 + 2, 3*x, length [1, 2]] :: [Int]
[3 < 4, a == 5, p && q]     :: [Bool]
[diff sin, inverse cos]   :: [Float -> Float]
```

Ограничений на длину списка нет, он может содержать, например, лишь один элемент (такие списки носят название *singleton*). Так, список `[[1, 2, 3]]` является синглетоном — он содержит один элемент, являющийся списком.

Список, не содержащий элементов, играет особую роль в определениях функций для работы со списками. Такой список называется пустым. Его объявление полиморфно — пустой список может быть представителем списка любого типа. Напомним, что в подобных случаях в объявлениях типа используются, так называемые, переменные типа (ти́повые переменные), например, `[a]`. Пустой

список может встречаться в любом месте, где можно разместить обычный список. Его тип определяется исходя из контекста: в записи `sum []` используется пустой список чисел; в `and []` — пустой список типа `[Bool]`.

Конструирование с помощью оператора : — другой распространенный способ определения списков. Этот оператор помещает элемент в начало списка, результат его применения есть список большей длины.

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

λ

Так, если `xs` есть список `[2, 3, 4]`, то `1 : xs` есть список `[1, 2, 3, 4]`. Используя пустой список и оператор `:`, можно получить любой список. Например, `1 : (2 : (3 : [ ]))` есть список `[1, 2, 3]`. Учтя правую ассоциативность этого оператора, опустим скобки `1 : 2 : 3 : [ ]`.

Указание числовых интервалов — третий способ, позволяющий определить список чисел. В этом случае в квадратных скобках размещаются два числа, разделенные двоеточием:

```
---> [1 .. 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
---> [2.6 .. 6.0]
[2.6, 3.6, 4.6, 5.6]
```

H

Хотя символ точка и может использоваться в имени оператора, комбинация символов `..` не является оператором — это зарезервированная последовательность символов языка Haskell (см. стр. 71).

Выражение `[x .. y]` вычисляется путем вызова `enumFromTo x y`. Функция `enumFromTo` использует рекурсивный вызов самой себя: добавив левую границу к списку, она вызывается с аргументом большим на единицу. Если правая граница меньше левой, то срабатывает терминальный случай: выдается пустой список.

▶ *Упражнение VI.1.1*

Создайте файл, в который поместите нижеприведенный скрипт.

```
import Prelude hiding (enumFromTo)

enumFromTo x y | y <x      = []
               | otherwise = x : enumFromTo (x+1) y
```

λ

Загрузив определение функции `enumFromTo`, примените ее к различным спискам.

◀

1.2. Функции на списках

При определении функций на списках часто применяется сопоставление входных параметров с образцами (паттернами): функция задается при помощи отдельных правил для пустого списка и для списка вида `x:xs`. Любой список, отличный от пустого, представим в виде первого элемента (головы списка) `x` и, возможно пустого, списка `xs` (хвост списка).

Некоторые функции, определенные на списках, уже обсуждались ранее (`head`, `tail`, `sum`, `length`, `map`, `filter`, `foldr` и `foldl`). Даже для тех функций, которые определены в прелюдии и доступны для вызова без определения их в пользовательском скрипте, требуется понимание того, как они работают и каково их определение. Прежде все потому, что их определения являются замечательными примерами функций, определенных на списках, и их текст помогает понять, как все же они работают.

В этом параграфе мы познакомимся с некоторыми из определений функций для работы со списками. Большинство из них являются рекурсивными, т. е. в правиле, соответствующем образцу вида `x:xs`, обязательно будет присутствовать вызов с параметром `xs`.

Сравнение и упорядочивание списков

Два списка равны, если они содержат одни и те же элементы в том же порядке. Вот определение функции `eq`, проверяющей два списка на равенство:

λ
```
[ ]    `eq` [ ]     = True
[ ]    `eq` (y:ys)  = False
(x:xs) `eq` [ ]     = False
(x:xs) `eq` (y:ys)  = (x == y) && xs `eq` ys
```

Здесь как первый так и второй параметры могут быть как пустыми списками, так и не пустыми, поэтому для определения функции потребовалось задать четыре правила: по одному на каждую из возможных комбинаций. Рекурсивный вызов присутствует в последнем правиле: если оба списка не пусты, то их первые элементы сравниваются на равенство, после чего функция вызывается снова, но аргументами уже являются хвосты исходных списков.

Так как в определении используется оператор сравнения `==`, то нельзя применять эту функцию к спискам произвольного типа.

Используя в объявлении класс Eq, мы ограничим область определения этой функции такими типами данных, которые допускают проверку на равенство:

```
eq    :: Eq a => [a] -> [a] -> Bool
```

λ

Функция eq не определена в прелюдии. Для проверки списков на равенство можно использовать оператор ==. Заметим, что далеко не все списки содержат элементы, допускающие проверку на равенство, например, нельзя сравнить на равенство два списка функций.

▶ *Упражнение VI.1.2*

Создайте скрипт, в который включите определение функции eq. Проверьте, как ведет себя эта функция на списках различных типов:

```
---> [1, 2] `eq` [1, 2]
True
---> [1, 2] `eq` [2, 1]
False
---> [sin pi, 2.0] `eq` [sin pi, sqrt 4]
True
---> [ 2 > 1, False] `eq` [True, 1 > 2]
True
```

H

Попробуйте применить эту функцию к спискам, содержащим функции.

◀

Если элементы списка допускают упорядочение с помощью операций сравнения $<$, $\leqslant$ и т. п., то такие списки также можно сравнить, определяя какой из двух списков «не больше другого». При этом сравнение производится *в лексикографическом порядке* (как в словарях): сравниваются первые элементы списка, если они равны, тогда вторые и так далее, пока текущий элемент одного списка не будет отличаться от соответствующего элемента другого списка. Например, [2,3] < [3,1] и [2,1] < [2, 2]. Если один из списков в процессе сравнения элементов закончился, а другой еще нет, то более короткий список «*меньше*» более длинного. Тот факт, что в таком определении присутствуют слова «и так далее» означает наличие рекурсии в теле определения функции:

```
se                  :: Ord a => [a] -> [a] -> Bool
[ ]    `se` [ ]      = True
```

λ

```
[ ]    `se` (y:ys)  = False
(x:xs) `se` [ ]     = False
(x:xs) `se` (y:ys)  = x < y  || (x == y && xs `se` ys)
```

Теперь, когда функция `se` задана, не составит труда определить другие функции сравнения: `ne` (не равны), `ge` (больше или равны), `st` (строго меньше) и `gt` (строго больше).

```
xs `ne` ys = not (xs `eq` ys)
xs `ge` ys = ys `se` xs
xs `st` ys = xs `se` ys && xs `ne` ys
xs `gt` ys = ys `st` xs
```

Конечно, и эти функции можно определить рекурсивно, по аналогии с функцией `se`. В прелюдию определения этих функций не включены. Для сравнения списков можно использовать стандартные операторы сравнения (`>=`, `<=` и т. д.)

▶ *Упражнение VI.1.3*

Добавьте определения функций `se`, `ne`, `ge`, `st` и `gt` в скрипт. Приведите примеры использования этих функций.

◀

Объединение списков

Два списка одного и того же типа могут быть объединены в один с помощью оператора `++`. Этот процесс носит название *конкатенация* (concatenation, сцепление вместе). Например, `[1, 2, 3] ++ [4, 5]` даст список `[1, 2, 3, 4, 5]`. Конкатенация с пустым списком не изменяет исходный список: `[1, 2] ++ [ ]` дает `[1, 2]`.

Оператор `++` является стандартной функцией, определение которой размещено в файле `Prelude.hs`. Ее определение таково:

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

В этом определении имеются два правила сопоставления с образцом для левого аргумента. В случае непустого списка функция применяются рекурсивно к меньшему аргументу — хвосту левого списка.

Имеется еще одна функция, объединяющая списки. Это функция `concat`, с единственным аргументом — *списком* списков. Она соединяет все элементы всех списков в один список, например,

```
---> concat [ [1, 2, 3], [4, 5], [], [6] ]
[1, 2, 3, 4, 5, 6]
```

Н

Определение функции `concat` таково:

```
concat          :: [[a]] -> [a]
concat []        = []
concat (xs:xss)  = xs ++ concat xss
```

Первый образец, `[]`, есть пустой список, в данном случае это пустой список списков. Результатом также является пустой список. Во втором случае список не пуст, он содержит первый элемент список `xs` и (возможно пустой) хвост `xss`, являющийся списком списков. Именно к этому списку меньшей длины и применяется снова функция `concat`, результат применения которой будет добавлен при помощи оператора `++` к списку `xs`.

Обратите внимание на различие между `++` и `concat`: оператор `++` работает с двумя списками, в то время как у функции `concat` имеется лишь один аргумент — список списков. Но обе эти функции в разговорной речи называют конкатенацией.

▶ *Упражнение VI.1.4*

Объединим списки `[1, 2, 3, 4]` и `[5, 6, 7]` двумя способами (с помощью функций `++` и `concat`):

```
--->  [1, 2, 3, 4] ++ [5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
---> concat [ [1, 2, 3, 4],  [5, 6, 7]]
[1, 2, 3, 4, 5, 6, 7]
```

Н

◀

Разбиение списка на части

В прелюдии определен целый ряд функций, позволяющих выделять ту или иную часть списка. Результатом работы некоторых является лишь один элемент, другие функции возвращают список.

Приведем еще раз определения функций, выделяющих голову и хвост списка:

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

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

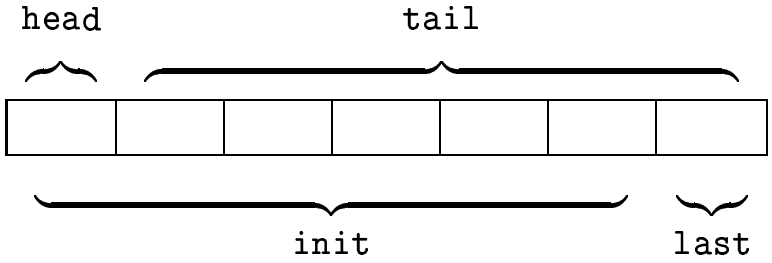
λ

В этих определениях присутствует сопоставление с образцом. Однако мы не видим правила для образца `[]`. Это объясняется тем, что понятия головы и хвоста введены лишь для непустых списков. Поэтому при вызове любой из этих функций с аргументом, являющимся пустым списком, будет получено сообщение об ошибке.

Несколько сложнее записывается функция, которая выделяет *последний* элемент списка. В ее определении уже присутствует рекурсия:

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

И снова эта функция не определена на пустом списке, так как этот случай не входит ни в один из двух паттернов, присутствующих здесь. По аналогии с парой функций `head` и `tail`, парой для `last` является `init`. На рисунке схематично представлены все четыре функции:

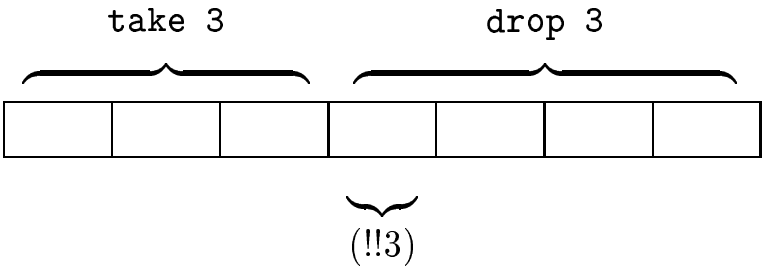


Функция `init` выделяет все кроме последнего элемента. В ее определении также присутствует рекурсия:

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

Образец `x:[]` может быть записан (и обычно записывается) в виде `[x]`.

На странице 100 было рассмотрено определение функции `take`. Кроме самого списка эта функция требует указания параметра, определяющего количество элементов в итоговом списке. В противоположность `take`, функция `drop` удаляет указанное число элементов из начала списка. Имеется и оператор `!!`, который выделяет один единственный элемент списка с указанным номером (нумерация элементов списка начинается с нуля). Схематично эти три функции представлены на следующей схеме:



Определены они следующим образом:

```
take, drop        :: Int -> [a] -> [a]
take 0     xs     = [ ]
take n     [ ]    = [ ]
take (n+1) (x:xs) = x : take n xs
drop 0     xs     = xs
drop n     [ ]    = [ ]
drop (n+1) (x:xs) = drop n xs
```

λ

В случае, если количество элементов в списке меньше целочисленного аргумента этих функций, функция `take` вернет весь список целиком, а функция `drop` возвратит пустой список. Это следует из второго правила каждой из функций.

Оператор `!!` выделяет один элемент из списка. Голова списка имеет номер 0, а вызов `xs !! 3` вернет *четвертый* элемент списка `xs`. Этот оператор не должен применяться к коротким спискам, так как не указано, что делать в таком случае. Познакомьтесь с определением этого оператора:

```
infixl 9 !!
(!!)    :: [a] -> Int -> a

(x:xs) !! 0 = x
(x:xs) !! (n+1) = xs !! n
```

λ

▶ *Упражнение VI.1.5*

Создайте скрипт с определениями этих трех функций. Для того чтобы скрыть определения, загруженные из прелюдии, добавьте в начало скрипта строку

```
import Prelude hiding (take, drop, (!!))
```

H

Примените эти функции к различным спискам.

◀

С использованием оператора `!!` можно определить более компактную функцию `dayofweek`, взамен функции `weekday`, рассматриваемой на стр. 99:

λ
```
dayofweek d = [ "Sunday", "Monday", "Tuesday",
                "Wednesday", "Thursday", "Friday",
                "Saturday"] !! d
```

▶ *Упражнение VI.1.6*

Определите функцию `day'`, отличающуюся от функции `day` (см. стр. 180) вызовом функции `dayofweek` вместо `weekday`.

◀

Обращение списка

Функция `reverse`, включенная в прелюдию, инвертирует порядок элементов в списке. Эта функция легко определяется с помощью рекурсии. Для пустого списка результат совпадает с исходным — он также является пустым списком. В случае непустого списка следует инвертировать его хвост, а голову переместить в конец:

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

Получение характеристик списка

Одной из наиболее часто используемых характеристик списка является его длина, вычисляемая путем применения функции `length`. В прелюдии она определена следующим образом:

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

Там же определена функция `elem`, проверяющая, содержится ли в списке тот или иной элемент:

λ
```
elem    :: Eq a => a -> [a] -> Bool
elem e xs = or (map (== e) xs)
```

Эта функция сравнивает все элементы списка `xs` в `e` (используя частично параметризованный оператор `==`). Промежуточный

результат — список величин типа `Bool`, к которому затем применяется функция `or`, пытающаяся найти хотя бы один элемент, равный `True`. С использованием оператора композиции функций определение можно переписать в виде:

```
elem e = or . (map (== e))
```
λ

В противоположность этой функции `notElem` истинна, если ее первый параметр не содержится в списке:

```
notElem e xs = not (elem e xs)
```
λ

▶ *Упражнение VI.1.7*

Рассмотрим функцию `and`, включенную в файл `Prelude.hs` и проверяющую, все ли элементы списка истинны. Определение функции содержит два образца, с которыми сопоставляется список-аргумент. Как обычно, один из них — пустой список, а второй — список, содержащий голову:

```
import Prelude  hiding(and)
```
λ

```
and       :: [Bool] -> Bool
and [ ]    = True
and (x : xs) = x && and xs
```

Убедитесь в корректной работе определенной вами функции:

```
---> and [ 1>2, 3<2]
False
---> and [ 1<=2, 3<2]
False
---> and [ 1<=2, 3>2]
True
---> and [ 1<=2, 3>2, True]
True
```
H

◀

С использованием функции `and` функцию `notElem` можно представить и в таком виде:

```
notElem e = and . (map (/= e))
```
λ

1.3. Сортировка списков

Сортировка списка, т. е. размещение его элементов в неубывающем порядке, может осуществляться с помощью функций, реализующих различные *алгоритмы* сортировки. В этом разделе будут

проанализированы два алгоритма. Оба алгоритма предполагают, что элементы списка могут быть упорядоченены. Так, это могут быть списки целых чисел или списки списков чисел типа `Float`, но не могут быть, например, списки функций. Данный факт следующим образом отражен в объявлении типа функции, осуществляющей сортировку:

λ
```
sort   :: Ord a => [a] -> [a]
```

Это означает, что функция `sort` может применяться к спискам, элементы которого относятся к классу `Ord`, т. е. допускают сравнение.

Сортировка вставкой

Предположим, что у нас уже имеется отсортированный список. Тогда новый элемент может быть вставлен на подходящее место при помощи такой функции:

λ
```
insert              ::  Ord a => a -> [a] -> [a]
insert e [ ]         = [e]
insert e (x:xs)
        | e <= x     = e : x : xs
        | otherwise = x : insert e xs
```

Если список был пуст, то новый список содержит единственный элемент `e`. Если список не пуст, и `x` — его первый элемент, то `x` сравнивается с `e`. В случае, когда элемент `e` меньше или равен `x`, он вставляется в начало списка; в противном случае, головой списка остается `x` и осуществляется рекурсивная вставка `e` в список `xs`:

H
```
---> insert 3 [1, 2, 4]
[1,2,3,4]
```

Напомним, что для применения функции `insert` нужен отсортированный список, только в этом случае список с добавленным элементом, также будет отсортирован. Тем не менее, функция `insert` может использоваться для сортировки списков.

Предположим, что требуется отсортировать список `[a, b, c, d]`. Возьмем пустой список и поместим в него элемент `d`. Результатом будет отсортированный список из одного элемента. Теперь добавим в него элемент `c`, затем `b` и, наконец, `a`. На каждом шагу мы получали отсортированный список и результат — тоже отсортирован. Выражение, вычисление которого привело к сортировке списка, таково

```
a `insert` (b `insert` (c `insert` (d `insert` [ ])))
```

Структура этого выражения полностью соответствует применению свертки с функцией **insert**, рассмотренной выше, в качестве оператора и пустого списка в качестве стартового значения. Отсюда получаем один из возможных алгоритмов сортировки:

```
isort  = foldr insert [ ]
```

λ

Этот алгоритм называется *сортировка вставкой*.

Сортировка слиянием

Другой алгоритм сортировки использует возможность слияния двух отсортированных списков в один. Это осуществляется с помощью функции **merge**:

```
merge              :: Ord a => [a] -> [a] -> [a]
merge []   ys       = ys
merge xs   []       = xs
merge (x:xs) (y:ys)
       |  x <= y    = x : merge xs (y:ys)
       | otherwise  = y : merge (x:xs) ys
```

λ

В случае, когда один из списков пуст, результатом является другой список. Если оба списка не пусты, то меньший из двух головных элементов становится головой нового списка, а оставшиеся элементы продолжают участвовать в рекурсивном вызове функции **merge**.

Аналогично **insert**, функция **merge** предполагает, что ее параметры уже отсортированы. Только в этом случае можно быть уверенным в том, что в результате слияния будет получен отсортированный список.

Воспользовавшись функцией **merge**, можно построить алгоритм сортировки списка. Этот алгоритм основывается на том факте, что пустой список и синглетон (список, содержащий один элемент) уже отсортированы. Любой список большей длины может быть разделен на две части, каждая из которых содержит меньшее число элементов. Полученные две половинки можно отсортировать при помощи рекурсивного вызова алгоритма сортировки, после чего отсортированные части сливаются в один список функцией **merge**:

```
msort   :: Ord a => [a] -> [a]
msort  xs
     | len <= 1   = xs
```

λ

```
    | otherwise  = merge (msort ys) (msort  zs)
    where ys     = take half xs
          zs     = drop half xs
          half   = len `div` 2
          len    = length xs
```

Этот алгоритм носит название *сортировка слиянием.*

1.4. Функции высшего порядка на списках

Одной из особенностей функционального программирования является то, что функции могут выступать в роли данных для других функций. Многие функции для работы со списками являются функциями высшего порядка: их аргументами выступают другие функции.

Функции `map` и `filter`

Ранее уже обсуждались функции `map` и `filter`. Функция `map` применяет функцию-параметр к каждому элементу списка:

```
              xs  = [ 1 , 2 , 3 , 4 , 5 ]
                    ↓   ↓   ↓   ↓   ↓
map square xs  = [ 1 , 4 , 9 , 16 , 25 ]
```

Функция `filter` удаляет из списка все элементы, не удовлетворяющие условию:

```
               xs  = [ 1 , 2 , 3 , 4 , 5 ]
                     ×   ↓   ×   ↓   ×
filter even xs  = [     2 ,     4     ]
```

Рекурсивные определения этих функций, включенные в файл `Prelude.hs`, обсуждались на стр. 85.

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

filter  :: (a -> Bool) -> [a] -> [a]
filter p [ ]    = [ ]
filter p (x:xs) |  p x       = x : filter p xs
                |  otherwise = filter p xs
```

Свертки

Свертка справа `foldr` вставляет оператор между всеми элементами списка, начиная с правой стороны:

```
            xs  = [  1   ,  2   ,  3   ,  4   ,  5  ]
                     ↓      ↓      ↓      ↓      ↓
foldr (+) 0 xs  =  (1  +  (2  +  (3  +  (4  +  (5  + 0)))))
```

Функция `foldl` вставляет оператор между всеми элементами списка, начиная с левой стороны:

```
            xs  =        [ 1   ,  2   ,  3   ,  4   ,  5 ]
                           ↓      ↓      ↓      ↓      ↓
foldl (+) 0 xs  =  (((((0 +  1)  +  2)  +  3)  +  4)  +  5)
```

Вспомним приводимые ранее определения этих функций (стр. 85):

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

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

λ

С помощью этих стандартных функций можно избавиться от явной рекурсии в определениях большинства других функций. Например, функция `or`, выясняющая, имеется ли в списке хотя бы один элемент, равный `True`, может быть определена так:

```
or  = foldr (||) False
```

λ

Вот рекурсивное определение, в котором не участвует свертка:

```
or [ ]      = False
or (x:xs)   = x || or xs
```

λ

Многие функции могут быть переписаны в терминах `foldr` и `map`. Хорошим примером для иллюстрации такой возможности является функция `elem`:

```
elem e = foldr (||) False . map (== e)
```

λ

Конечно, и эта функция может быть определена явно, без использования стандартных функций. В этом случае определение будет рекурсивно:

```
elem e [ ]     = False
elem e (x:xs) = x == e || elem e xs
```

λ

Функция `foldr` используется для определения самых разных функций на списках. Вот еще несколько примеров:

```
reverse   :: [a] -> [a]
reverse    = foldr snoc [ ]
    where snoc x xs = xs ++ [x]

length    ::[a] -> Int
length     = foldr oneplus 0
    where oneplus x n = 1 + n

sum, product     :: Num a => [a] -> a
sum        = foldr (+) 0
product    = foldr (*) 1

and        :: [Bool] -> Bool
and         = foldr  (&&) True

map        :: (a -> b) -> [a] -> [b]
map f       = foldr (cons . f) [ ]
    where cons x xs = x : xs
```

Не каждая функция на списках может быть определена через `foldr`. Например, рассматриваемая далее функция `zip` может быть выражена только через свертку слева.

Свертки без начальных значений

Предположим, что нам требуется найти максимальный элемент списка чисел. Вспомнив про свертку, мы могли бы попробовать нечто вроде следующего определения:

```
maxlist :: Num a => [a] -> a
maxlist = foldr max e
```

Но что выбрать в качестве начального элемента `e`? Если бы было известно, что список содержит неотрицательные значения, то можно было бы считать `e` равным нулю. Но нас не устраивает такое ограничение. Итак, чему должно равняться `maxlist [ ]`? Ведь нам хотелось бы иметь `maxlist [x] = x` для любого `x`. Но каково самое меньшее из всех возможных чисел? В языке Haskell предусмотрен специальный класс `Bounded`, содержащий только две величины: `minBound` — минимальное и `maxBound` — максимальное из возможных чисел.

С учетом вышесказанного, теперь возможно определить функцию `maxlist` так:

```
maxlist ::  [Int] -> Int
maxlist = foldr max minBound
```

λ

Альтернативным решением, без использования величины `minBound`, явилось бы введение новой функции свертки, такой чтобы ей не требовалось инициализирующее значение. В действительности можно определить две таких функции, `foldr1` и `foldl1`.

`foldr1` $(\oplus)$ $[x_1, x_2, \ldots, x_{n-1}, x_n] = x_1 \oplus (x_2 \oplus (\ldots (x_{n-1} \oplus x_n) \ldots))$,
`foldl1` $(\oplus)$ $[x_1, x_2, x_3, \ldots, x_n] = (\ldots ((x_1 \oplus x_2) \oplus x_3) \ldots) \oplus x_n$,
где $\oplus$ — бинарный оператор.

Записав определение на языке Haskell, получим

```
foldr1           :: (a -> a -> a) -> [a] -> a
foldr1 f [x]      = x
foldr1 f (x:xs)   = f x (foldr1 f xs)
```

λ

Соответствующее определение через функцию `foldl` таково:

```
foldl1           :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)   = foldl f x xs
```

λ

Теперь функцию `maxlist` можно определить так:

```
maxlist = foldr1 (max)
```

λ

Так как функция `max` ассоциативна, то в этом определении можно использовать как `foldr1`, так и `foldl1`.

▶ *Упражнение VI.1.8*

Подготовьте скрипт, содержащий разнообразные определения функции `maxlist`, после чего убедитесь в их эквивалентности.

◀

Функции `takeWhile` и `dropWhile`

Рассмотрим еще две полезные функции высшего порядка на списках: `takeWhile` и `dropWhile`.

Функция `takeWhile` очень напоминает функцию `filter`: ее аргументами также являются предикат и список. Но в отличии от `filter`, которая просматривает весь список в поисках подходящих элементов, `takeWhile`, просматривая список с начала, останавливается сразу, как только найдется элемент, не удовлетворяющий заданному предикату. Например,

H
```
---> takeWhile even [2, 4, 6, 7, 8]
[2,4,6]
---> filter even [2, 4, 6, 7, 8]
[2,4,6,8]
```

В отличии от `filter` функция `takeWhile` не поместила число 8 в результирующий список, так как она завершила свою работу после проверки числа 7. Ознакомимся с ее определением:

λ
```
takeWhile           :: (a -> Bool) -> [a] -> [a]
takeWhile p [ ]       = []
takeWhile p (x:xs)
         | p x        = x : takeWhile p xs
         | otherwise = [ ]
```

Так же как `take` соседствует с `drop`, функция `takeWhile` связана с функцией `dropWhile`. Она удаляет из списка все начальные элементы, удовлетворяющие условию. Начиная с первого элемента, который не удовлетворил условию, все оставшиеся элементы помещаются в результирующий список:

H
```
---> dropWhile even [2, 4, 6, 7, 8, 9]
[7,8,9]
```

Определение этой функции таково:

λ
```
dropWhile           :: (a -> Bool) -> [a] -> [a]
dropWhile p [ ]       = [ ]
dropWhile p (x:xs)
         | p x        = dropWhile p xs
         | otherwise = x:xs
```

Функция `zip`

Функция `zip` берет два списка и возвращает список пар соответствующих элементов; если списки имеют разную длину, то длина результирующего списка совпадет с длиной более короткого.

H
```
---> zip [0 .. 4] "hello"
[(0,'h'),(1,'e'),(2,'l'),(3,'l'),(4,'o')]
---> zip [0 .. 1] "hello"
[(0,'h'),(1,'e')]
```

Полное определение `zip` таково:

λ
```
zip              :: [a] -> [b] -> [(a,b)]
zip [ ]    ys     = [ ]
```

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

Образцы соответствия, применяемые в этом определении, аналогичны образцам функций `take` и `drop`.

Функция `zip` имеет множество применений. Следующие два упражнения демонстрируют полезность этой функции.

▶ *Упражнение VI.1.9*

Как известно, скалярное произведение двух векторов $\bar{x}$ и $\bar{y}$ равно сумме попарных произведений координат векторов:

$$\bar{x} \cdot \bar{y} = \sum_{i=1}^{n} x_i \cdot y_i.$$

Функция `sp`, вычисляющий скалярное произведение, может быть определена так

```
sp       :: Num a => [a] -> [a] -> a
sp xs ys  = sum (map times (zip xs ys))
    where times (x, y) = x * y
```

λ

Сохраните определение функции `sp` в файле и примените ее к векторам различной размерности.

Обратите внимание, что используемая здесь функция `times` является всего лишь некарринговой версией оператора умножения `*`, другими словами `times = uncurry (*)` (см. стр. 79). Внесите изменение в скрипт, после чего проверьте его работоспособность.

◀

▶ *Упражнение VI.1.10*

Рассмотрим функцию `nondec`, определяющую, является ли последовательность $[x_0, \ldots, x_{n-1}]$ неубывающей. Другими словами, требуется проверить, верно ли, что $x_k \leqslant x_{k+1}$ для всех k из диапазона $0 \leqslant k \leqslant n-2$. Тип функции `nondec` таков

```
nondec :: Ord a => [a] -> Bool
```

λ

Конечно, эта функция допускает рекурсивное определение, однако ознакомьтесь с ее определением через функцию `zip`:

```
nondec xs  = and ( map leq (zip xs (tail xs)))
     where leq (x, y) = x <= y
```

λ

Напомним, что функция `and` берет список из логических величин и возвращает `True`, если все они равны `True`, и `False` в противоположном случае. Два ее определения рассмотрены на страницах 117 и 121.

Выражение `zip xs (tail xs)` возвращает список пар соседних элементов списка. В частности, из ленивости вычислений следует, что

```
zip [ ] (tail [ ]) = [ ]
zip [x] (tail [x]) = zip [x] [ ] = [ ]
```

Заметим также, что функция `leq` есть некаррингова форма оператора сравнения «меньше или равно», т. е. `leq = uncurry (<=)`.

◀

Эти примеры проиллюстрировали, что комбинации `map`, `filter` и `zip` могут применяться в самых различных вариантах. Для того чтобы сделать определения короче можно ввести функцию `zipWith` следующим образом:

λ
```
zipWith          :: ( a -> b ->c) -> ([a] -> [b] ->[c])
zipWith f [] ys = []
zipWith f xs [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

Например,

λ
```
sp xs ys  = sum (zipWith (*) xs ys)
nondec xs = and (zipWith (<=) xs (tail xs))
```

Для еще большей краткости мы можем ввести некаррингову версию `zip` и `zipWith`:

λ
```
zipp   ::([a], [b]) ->[(a, b)]
zipp    = uncurry zip
zippWith :: (a -> b -> c) -> ([a], [b]) -> [c]
zippWith f = map (uncurry f) . zipp
```

Для `spp`, некарринговой версии `sp`, имеем

λ
```
spp  = sum . zippWith (*)
```

Мы получили чисто функциональное определение `spp`: определение совсем не содержит параметров. Ниже приводится аналогичное определение для функции `nondec`:

λ
```
nondec   :: Ord a => [a] -> Bool
nondec = and . zipWith (<=) . pair (id, tail)
```

В нем используется функция `pair`, заданная так:

```
pair   :: (a -> b, a -> c) -> a -> (b, c)
pair (f, g) x = (f x, g x)
```
λ

Функция `unzip`

В противоположность `zip` функция `unzip` преобразует список пар в пару списков:

```
unzip   :: [(a, b)] ->([a], [b])
unzip    = pair (map fst, map snd)
```
λ

Действие этой функции проиллюстрируем примером:

```
---> unzip [(1, True), (2, True), (3, False)]
([1,2,3], [True,True,False])
```
H

Функция `zipp` (некарринговая версия `zip`) и `unzip` связаны уравнением `zipp (unzip xys) = xys`:

```
---> zipp (unzip [(1, True), (2, True), (3, False)])
[(1,True),(2,True),(3,False)]
```
H

Определение этой функции с помощью свертки справа таково:

```
unzip    :: [(a, b)] -> ([a], [b])
unzip    = foldr conss ([ ], [ ])
   where conss (x, y) (xs, ys) = (x:xs, y:ys)
```
λ

Сканирование слева

Иногда возникает задача применения функции `foldl` к каждому начальному сегменту списка. Эта задача решается с помощью функции «сканирования слева» `scanl`.

`scanl` $(\oplus)\ e\ [x_1, x_2, \ldots, x_{n-1}, x_n]] =$
$[e, e\oplus x_1, (e\oplus x_1)\oplus x_2, \ldots, ((\ldots(e\oplus x_1)\oplus x_2 \ldots x_{n-1})\oplus x_n]$
где $\oplus$ бинарный оператор, а `e` — стартовое значение.

В частности, `scanl (+) 0` вычисляет список частичных сумм списка чисел, а `scsnl (*) 1 [1 .. n]` вычисляет список первых `n` факториалов чисел:

```
---> scanl (+) 0 [1, 1, 1, 1]
[0, 1, 2, 3, 4]
---> scanr (*) 1 [1 .. 5]
[120, 120, 60, 20, 5, 1]
```
H

В этом разделе мы рассмотрим два определения этой функции: первое — ясное, и второе — эффективное.

Для первой программы нам требуется функция `inits`, возвращающую список начальных сегментов списка. Так для списка, состоящего из трех элементов, должно выполняться следующее соотношение:

`inits` $[x_0, x_1, x_2]$ = $[[\,], [x_0], [x_0, x_1], [x_0, x_1, x_2]]$.

Пустой список состоит только из одного сегмента, также являющегося пустым списком. Результат применения функции `inits` к списку вида `(x:xs)` содержит пустой список в качестве самого короткого сегмента, а все другие начальные сегменты начинаются с `x` и содержат все начальные сегменты `xs`. Соответствующее рекурсивное определение таково:

λ
```
inits   :: [a] -> [[a]]
inits [ ]  = [[ ]]
inits (x:xs) = [ ] : map (x:) (inits xs)
```

Убедимся в корректности нашего определения:

H
```
---> inits [1,2,3,4]
[[],[1],[1,2],[1,2,3],[1,2,3,4]]
```

Теперь можно дать определение функции `scanl`:

λ
```
scanl    :: (a -> b -> a) -> a -> [b] -> [a]
scanl f e = map (foldl f e ) . inits
```

▶ *Упражнение VI.1.11*

Подготовьте скрипт, содержащий приведенное выше определение функции `scanl`. Не забудьте, что определение этой функции включено в прелюдию. Получите список сумм чисел от 1 до 20 и список факториалов от 1 до 20.

◀

Рассмотренная реализация `scanl` соответствует определению функции, но приводит к неэффективной программе. Функция `f` применяется `k` раз во время вычисления `foldl f e` для списка длины `k`, а так как начальные сегменты списка длины `n` есть списки, длины которых равны `0, 1, ..., n`, то функция `f` применяется всего $n^2/2$ (докажите!) раз. Версия этой функции, размещенная в файле `Prelude.hs`, менее понятна, но значительно эффективнее:

λ
```
scanl f e [ ]    =  [e]
scanl f e (x:xs) = e: scanl f (f e x ) xs
```

Эта программа более эффективна потому, что функция `f` применяется точно `n` раз для списка длины `n`.

Сканирование справа

Двойственное вычисление дает **scanr**, определенная следующим образом:

```
scanr f e = map (foldr f e) . tails
```
λ

Функция `tails` возвращает хвостовые сегменты списка. Например,

`tails` $[x_0, x_1, x_2]$ = $[[x_0, x_1, x_2], [x_1, x_2], [x_2], []]$

Заметим, что `inits` производит список начальных сегментов с возрастающей длиной, а `tails` — хвостовые сегменты убывающей длины. Мы определим `tails` так

```
tails [] = [[]]
tails (x:xs) = (x:xs) : tails xs
```
λ

Применим ее к списку, состоящему из трех элементов:

```
---> tails [1,2,3]
[[1,2,3], [2,3], [3], []]
```
H

Рекурсивное определение функции **scanr** может быть, например, таким

```
scanr f e [ ]    = [e]
scanr f e (x:xs) = f x (head ys) : ys
   where  ys = scanr f e xs
```
λ

Посмотрим на результат ее применения:

```
---> scanr (*) 1 [1 .. 5]
[120,120,60,20,5,1]
```
H

Сканирование без пустых списков

По аналогии с функциями `foldr1` и `foldl1` мы можем определить еще пару сканирующих список функций — **scanl1** и **scanr1**.

```
scanl1 f = map (foldl1 f) . inits1
scanr1 f = map (foldr1 f) . tails1
```
λ

Здесь `inits1` и `tails1` возвращают *непустые* начальные и конечные сегменты непустых списков. Например,

```
inits1 (x:xs) =  map (x:) (inits xs)
```
λ

▶ *Упражнение VI.1.12*

Определите функцию `tails1`.

◀

Используя тот факт, что `foldl1 f . (x:) = foldl f x`, получим более эффективное определение функции `scanl1`:

λ
```
scanl1          :: (a -> a -> a) -> [a] ->[a]
scanl1 f (x:xs) = scanl f x xs
```

Составление более эффективной версии `scanr1` предлагается в качестве упражнения.

Вопросы и задания

VI.1.1

Пусть имеются три списка `xs`, `ys`и `zs`. Объедините эти списки при помощи функций `++` и `concat`. Является ли оператор `++` ассоциативным оператором?

VI.1.2

Вычислите `map (map square) [[1,2], [3,4,5]]`.

VI.1.3

Функция `filter` может быть определена в терминах `concat` и `map` следующим образом:

```
filter p = concat . map box
     where  box x = ...
```

Дайте определение для функции `box`.

VI.1.4

Определите функцию `filter p` с помощью функции `foldr`.

VI.1.5

Вспомним определения функций `takeWhile` и `dropWhile`, рассмотренные на стр. 124. Определите `takeWhile` в терминах `foldr`. Можно ли определить `dropWhile` через `foldr`?

VI.1.6

Какое из двух следующих определений верно?

```
foldl (-) x xs = x - sum xs
foldr (-) x xs = x - sum xs
```

VI.1.7

Рассмотрим следующее определение функции *insert*:

```
insert x xs = takeWhile (<= x) xs ++ [x] ++
              dropWhile (<= x) xs
```

Покажите, что если элементы `xs` расположены в неубывающем порядке, то тоже самое верно и для `insert x xs` при любом `x`. Используйте `insert` для определения функции `isort`, которая сортирует список в неубывающем порядке. Примените `foldr`.

VI.1.8

Функция `remdups` удаляет копии одинаковых соседних элементов из списка. Например, `remdups [1, 2, 2, 3, 3, 3, 1, 1] = [1, 2, 3, 1]`. Определите `remdups`, используя `foldl` или `foldr`.

VI.1.9

Дан список чисел `xs=[`x_1`, `x_2`, ..., `x_n`]`, последовательность следующих один за другим максимумов (*sequence of successive maxima*) `ssm xs`, есть наибольшая подпоследовательность вида `[`x_{j_1}`, `x_{j_2}`, ..., `x_{j_m}`]`, где $j_1 = 1$ и $x_j < x_{j_k}$ для $j < j_k$. Например, последовательность следующих максимумов для `[3, 1, 3, 4, 9, 2, 10, 7]` есть `[3, 4, 9, 10]`. Определите `ssm` в терминах `foldl`.

VI.1.10

Предложите эффективную программу для `scanr`.

VI.1.11

Какой список получится в результате `scanl (/) [1 .. n]`?

VI.1.12

Математическая константа e определена как $e = \sum_{n=0}^{\infty} \frac{1}{n!}$. Напишите выражение, которое можно использовать для вычисления e с некоторой разумной степенью точности.

2. Абстракции списков

В теории множеств широко применяется нотация задания множеств в виде, аналогичном следующему:

$V = \{x^2 \mid x \in N,\ x \text{ четно}\}$

По аналогии с этой записью, носящей название «включения множеств», Haskell позволяет задавать списки при помощи так называемых *включений списков* (называемых также *абстракциями списков*). Примером такой нотации является выражение вида `[ x*x | x <- [1 .. 10]]`, которое определяет список квадратов целых чисел, принадлежащих интервалу от 1 до 10. Абстракции списков содержат некоторое выражение, размещенное слева от вертикальной черты, причем в это выражение могут входить переменные. Ограничения на эти переменные (`x` в предыдущем примере) размещаются справа от вертикальной черты.

Абстракции списков, *называемые также* **включениями списков**, *дают элегантный метод сжатого описания определенных операций обработки списков.*

Нотация `x <- xs` означает, что `x` последовательно принимает все значения из списка `xs`. Для каждого такого значения `x` вычисляется выражение, размещенное перед чертой. Так, в примере, приведенном выше, получается такой же список, что и в результате выполнения команды `map square [1 .. 10]`, где функция `square` определена следующим образом:

```
square x = x*x
```

Преимущество использования абстракций списков состоит в том, что можно обойтись без именования функции, которую требуется вычислить (в нашем примере `square`).

Нотация абстракций списка очень разнообразна. После вертикальной черты могут указываться диапазоны изменения нескольких переменных. Выражения, размещенные до черты, также могут иметь самый разнообразный вид. Например,

H
```
---> [(x, y)| x <- [1 .. 2], y <- [4 .. 6] ]
[(1,4), (1,5), (1,6), (2,4), (2,5), (2,6)]
```

Вторая переменная изменяется быстрее: для каждого значения `x`, `y` пробегает все значения из списка `[4 .. 6]`.

Кроме диапазонов изменения переменных справа от черты могут находится логические выражения (типа `Bool`). В следующем

примере в результирующий список будут помещены пары только с четными `x` (для которых значение `even x` равно `True`).

```
---> [(x, y) | x <- [1 .. 5], even x, y <- [1 .. x] ]
[(2,1), (2,2), (4,1), (4,2), (4,3), (4,4)]
```

Н

Для каждой стрелочки (`<-`) переменная, указанная с ее помощью, может использоваться во всех последующих выражениях и в выражении левее черты. Так в примере `x` используется в тьюпле `(x, y)` и в выражениях `even x` и `[1 .. x]`. Однако `y` может использоваться только в `(x, y)`, так как его определение размещено последним.

Стрелочка (`<-`), используемая во включениях списков, является зарезервированным символом языка, а не оператором.

Формально включение списка имеют форму `[e | Q]`, где `e` — выражение, а `Q` — *квалификатор*. Квалификатор — это, возможно пустая, последовательность *генераторов* и *охранных* выражений, разделенных запятыми.

Правило генерации

```
[e | x <- xs, Q] = concat (map f xs) where f x = [e | Q]
```

Охранное правило

```
[e | p, Q]       = if p then [e | Q] else [ ]
```

Генераторы имеют форму `x <- xs`, где `x` есть переменная или кортеж переменных, а `xs` — список. Охранное выражение — это некоторое логическое условие. Квалификатор `Q` может быть пустым, в этом случае мы запишем просто `[e]`.

Используя эти правила, мы можем вычислить

```
  [x * x | x <- [1 .. 5], odd x]
=     {правило генерации}
  concat (map f [ 1 .. 5]) where f x = [x*x| odd x]
=     {охранное правило }
  concat (map f [ 1 .. 5])
              where f x = if odd x then [x*x] else [ ]
=     { применение map }
  concat [[1], [ ], [9], [ ], [25]]
=     { применение concat }
  [1, 9, 25]
```

▶ *Упражнение VI.2.1*

Выполните несколько команд, содержащих абстракции списков:

H
```
---> [(a, b) | a <- [1 .. 3], b <- [1 ..2]]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]
---> [(a, b) | a <- [1 .. 2], b <- [1 ..3]]
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]
```

Как генераторы могут зависеть от переменных показывает следующий пример:

H
```
---> [(i, j) | i <- [1 .. 4], j <- [i+1 .. 4]]
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```

Мы можем легко разнообразить генераторы с помощью охранных условий:

H
```
---> [(i, j)| i <- [1..4], even i, j <- [i+1..4], odd j]
[(2,3)]
```

В следующем примере генератор имеет форму `(x, y) <- xs`. Сначала разместим в скрипте определение списка пар:

λ
```
pairs = [(i, j) | i <- [ 1 .. 2], j <- [1 .. 3]]
```

После загрузки скрипта можно использовать этот список в различных генераторах:

H
```
---> [i+j | (i,j) <- pairs]
[2,3,4,3,4,5]
```

◀

В качестве более интересного примера рассмотрим программу, дающую список пифагоровых троек в данном диапазоне.

H
```
---> triads 13
[(3,4,5),(4,3,5),(5,12,13),(6,8,10),(8,6,10),(12,5,13)]
```

Пифагоровыми тройками называют кортежи чисел `(x,y,z)` таких что, $x^2 + y^2 = z^2$. Мы определим функцию `triads` так:

λ
```
triads   :: Int -> [(Int, Int, Int)]
triads n  = [(x,y,z) | (x,y,z) <- triples n, pyth (x,y,z)]

triples  :: Int -> [(Int, Int, Int)]
triples n = [(x,y,z) |
              x <- [1..n], y <- [1..n], z <- [1..n]]

pyth        :: (Int, Int, Int) -> Bool
pyth (x,y,z) = (x*x + y*y == z*z)
```

Так как `triads` находит одни и те же триады двумя различными путями, то следующее определение функции `triples` лучше:

```
triples n  = [(x,y,z)|x<-[1..n], y<-[x..n], z<-[y..n]]
```

λ

В новом определении `y` находится в диапазоне `x` $\leqslant$ `y` $\leqslant$ `n`, а величина `z` — в диапазоне `y` $\leqslant$ `z` $\leqslant$ `n`, таким образом `z` не может быть меньше, чем наибольшее значение среди `x` и `y`. Все триады теперь будут встречаться только один раз. Например,

```
---> triads 13
[(3,4,5),(5,12,13),(6,8,10)]
```

Н

Абстракции списков обеспечивают заманчивую альтернативу стилю, основанному на комбинировании `map`, `filter` и `concat`. Среди правил, связывающих эти методы, отметим следующие:

`[ f x | x <- xs]`	=	`map f xs`
`[ x | x <- xs, p x]`	=	`filter p xs`
`[ e | Q, p]`	=	`concat [[ e | p ] | Q]`
`[ e | Q, x <- [ d | P]]`	=	`[e[x:=d] | Q, P]`

В последнем уравнении `e[x:=d]` обозначает выражение, которое получается, если все вхождения `x` в выражение `e` заменить на `d` (подстановка).

Строго говоря, нотация абстракции списков является излишней. Ведь таких же эффектов можно добиться с использованием комбинаций таких функций, как `map`, `filter` и `concat`. Тем не менее, особенно в сложных случаях, включения списков значительно легче для понимания. Ниже список пар получен двумя способами: без абстракции списков и с их применением:

```
---> concat (map f (filter even [1 .. 5]))
     where f x = map g [1 .. x]
           where g y = (x, y)
[(2,1), (2,2), (4,1), (4,2), (4,3), (4,4)]

---> [(x, y)| x <- [1 .. 5], even x, y <- [1 .. x] ]
[(2,1), (2,2), (4,1), (4,2), (4,3), (4,4)]
```

Н

Заметьте, что первая запись уже не может считаться интуитивно ясной. Интерпретатор же непосредственно переводит нотацию абстракции списков в соответствующие выражения с использованием `map`, `filter` и `concat`. Таким образом, абстракция списков

есть просто средство, позволяющее сделать текст определения более понятным, а процесс разработки программ более простым. Выбор между включениями списков и стилем комбинирования `map` и `filter` определяется предпочтениями пользователя.

Вопросы и задания

VI.2.1

Определите функцию `pairs` такую, что `pairs n` есть список всех различных пар чисел в интервале $1 \leqslant x, y \leqslant n$.

VI.2.2

Напишите программу, находящую все различные четверки (a, b, c, d) в диапазоне $0 < a, b, c, d \leqslant n$ такие, что $a^2 + b^2 = c^2 + d^2$.

VI.2.3

Преобразуйте следующие включения списков в стиль комбинирования функций `map`, `filter` и `concat`

```
[(x, y) | x <- [1 .. n], odd x, y <- [1 ..n]]
[(x, y) | x <- [1 .. n], y <- [1 ..n], odd x]
```

3. Бесконечные списки

3.1. Создание бесконечных списков

Число элементов в списке может быть бесконечным. Следующая функция `from` теоретически может использоваться для получения такого списка:

λ
```
from n  = n: from (n+1)
```

Конечно, компьютер не сумеет выдать бесконечный список элементов. Фактически мы можем наблюдать только его начало. Если не прервать работу функции, нажав комбинацию клавиш *Ctrl+C*, то процесс вычисления будет продолжаться и продолжаться:

H
```
---> from 5
[5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,

{Interrupted!}
```

Бесконечный список используется в качестве промежуточного результата, в то время как окончательный результат будет конечным. Проиллюстрируем сказанное следующим примером. Пусть требуется найти все степени числа три, не превышающие 1000. Первые десять степеней мы можем найти с помощью вызова:

```
---> map (3^) [1 .. 10]                                          H
[3,9,27,81,243,729,2187,6561,19683,59049]
```

Элементы полученного списка, не превышающие 1000, найдем так:

```
---> takeWhile (<1000) (map (3^) [1 .. 10])                      H
[3,9,27,81,243,729]
```

Но как мы могли заранее быть уверены в том, что десяти элементов первоначального списка достаточно? Правильное решение — воспользоваться бесконечным списком, полученным с помощью вызова `from 1`, вместо списка `[ 1 .. 10]`, и получить таким образом *все* степени числа три:

```
---> takeWhile (<1000) (map (3^) (from 1))                       H
[3,9,27,81,243,729]
```

Такой подход возможен лишь потому, что интерпретатор «достаточно ленив»: выполнение предстоящей работы откладывается до тех пор, пока это возможно. Вызов `map (3^) (from 1)` не вычисляется полностью (это потребовало бы бесконечного времени). Сначала вычисляется только первый элемент, который затем передается функции `takeWhile`. Только после обработки первого элемента, функция `takeWhile` запросит следующий элемент и второй элемент списка `from 1` будет определен. Продолжая таким образом, `takeWhile` прекратит свою работу, т. е. не станет требовать следующий элемент от функции `map`, как только очередной элемент удовлетворит условию $\geqslant 1000$.

Использование бесконечных списков возможно только благодаря ленивым вычислениям. Языки, реализующие стратегию энергичных вычислений (большинство императивных языков и некоторые из языков функционального программирования), не могут оперировать бесконечными структурами данных.

Функция, которой требуются все элементы списка, не может применяться к бесконечным спискам. Примером таких функций являются `sum` и `length`. При вызове функций `sum (from 1)` или `length (from 1)` даже «ленивое» вычисление не поможет найти

ответ за конечное время. В этом случае компьютер никогда не выдаст конечный ответ.

Тот факт, что язык Haskell реализует стратегию отложенных вычислений практически постоянно учитывается при проектировании программ. Вспомним, к примеру, функцию `prime`, проверяющую является ли целое число простым (стр. 98).

λ
```
prime   :: Int -> Bool
prime x = denominators x == [1, x]
```

Находит ли эта функция все делители числа `x` для того, чтобы затем сравнить их список со списком `[1, x]`? Нет, так как на это потребовалось бы достаточно много времени. Посмотрим, что происходит при вычислении, например, `prime 30`. Сначала находится первый делитель числа 30 — это 1. Это значение сравнивается с началом списка `[1, 30]`. Выясняется, что первые элементы списка совпали. Определяется второй делитель 30 — это число 2. Оно сравнивается со вторым элементом списка `[1, 30]`, после чего выясняется, что они *не равны*. Оператор проверки на равенство `==` «знает», что списки не могут совпасть, если на одном и том же месте в них находятся различные элементы. Поэтому возвращается значение `False`. Остальные делители числа 30 не вычисляются!

Ленивое поведение оператора `==` следует из его определения (стр. 111). Рекурсивный случай в его определении выглядит так:

λ
```
(x:xs) == (y:ys) = x == y && xs == ys
```

Если `x == y` возвращает значение `False`, то `xs == ys` вычислять уже не нужно: конечным результатом всегда будет `False`. Это следствие ленивого поведения оператора `&&`, отраженного в его определении:

λ
```
False && x = False
True  && x = x
```

Если левый параметр имеет значение `False`, то значение правого параметра вычислять не нужно.

▶ *Упражнение VI.3.1*

Рассмотрим задачу построения функции `position` такой, что `position x xs` возвращает номер позиции первого вхождения элемента `x` в список `xs` (нумерация элементов списка начинается с 0), и -1, если `x` не встречается в списке. Тип функции `position`:

λ
```
position   :: Eq a => a -> [a] -> Int
```

Зачастую при решении задач бывает проще справится с более общей задачей, а исходную рассмотреть как ее частный случай. И в этом случае, мы сначала найдем позиции *всех* вхождений x в xs:

```
positions   :: Eq a => a -> [a] -> [Int]
positions x xs = [ i | (i, y) <- zip [0 ..] xs, x == y]
```

λ

Выражение `zip [0 ..] xs` возвращает пары каждого элемента списка `xs` с его порядковым номером в списке. Хотя `[0 ..]` есть бесконечный список, `zip [0 ..] xs` вернет конечный список, потому что `xs` — конечный список. Теперь мы можем определить `position` так:

```
position x xs = head (positions x xs ++ [-1])
```

λ

Заметим, что простота этого определения достигается не за счет увеличения времени вычисления. Сначала вычисляется голова списка и нет необходимости продолжать определять остальные элементы списка.

```
---> position 3 [1, 3, 4, 3]
1
```

H

◀

3.2. Функции на бесконечных списках

В прелюдии многие функции определены с помощью бесконечных списков. Так действительное имя функции `from` (стр. 136) есть `enumFrom`. Эту функцию обычно не используют в такой форме, так как вместо `enumFrom n` проще записать `[n .. ]` (сравните такую запись с записью `[n .. m]` вместо `enumFromTo n m`, обсуждаемой на стр. 109).

Бесконечный список, который содержит в себе повторение одного и того же элемента, может быть создан при использовании функции `repeat`:

```
repeat   :: a -> [a]
repeat x  = x : repeat x
```

λ

Вызов `repeat 't'` дает бесконечный список "tttttttt..."

Бесконечный список, генерируемый функцией `repeat`, используется в роли промежуточного результата для той или иной функции, имеющей конечный результат. Например, функция `copy` создает конечное число копий элемента:

```
copy     :: Int -> a -> [a]
copy n x  = take n (repeat x)
```

λ

Благодаря стратегии отложенных вычислений функция `copy` может использовать бесконечный результат функции `repeat`. Функции `repeat` и `copy` определены в прелюдии.

Наиболее гибкими в смысле использования того или иного вида списков являются функции высшего порядка, т. е. использующие другие функции в качестве параметра. Функция `iterate` получает функцию и начальный элемент в качестве параметров. Результатом является бесконечный список, в котором каждый элемент получается посредством применения функции к предыдущему элементу. Например:

H

```
---> iterate (+1) 3
[3,4,5,6,7,8, ...
    {Interrupted!}

---> iterate (*2) 1
[1,2,4,8,16,32,64,128,256, ...
    {Interrupted!}

---> iterate (`div` 10) 5678
[5678,567,56,5,0,0,0, ...
    {Interrupted!}
```

Определение `iterate`, приводимое в прелюдии, таково:

λ

```
iterate          :: (a -> a) -> a -> [a]
iterate f x       = x : iterate f (f x)
```

Эта функция напоминает функцию `until`, описанную на стр. 90. Она также получает функцию и начальный элемент в качестве параметров. Различие состоит в том, что `until` прекращает свою работу, как только значение удовлетворит основному условию (которое также является параметром). К тому же `until` возвращает только конечный результат (который удовлетворяет заданному условию), тогда как `iterate` сохраняет всю последовательность результатов в списке. Это возможно, потому что бесконечные списки не имеют последнего элемента.

В следующих примерах обсуждается как функцию `iterate` можно использовать для решения практических задач: вывода числа в виде строки, получение списка всех простых чисел и получение «случайной» последовательности чисел.

3.3. Примеры применения бесконечных списков

Преобразование числа в строку

Функция `intString` преобразует положительное целое число в строку, которая состоит из цифр этого числа. Например, `intString 5678` возвращает строку "5678". С помощью этой функции вы можете объединять результат вычислений со строкой, например, `intString (3*17) ++ " линий"`.

Функция `intString` может быть получена как последовательность выполнения ряда операций. Сначала число должно быть несколько раз разделено нацело на 10 с помощью функции `iterate` (также, как в третьем примере использования функции `iterate`). Не интересующую нас бесконечную последовательность нулей в конце списка нужно убрать, применив функцию `takeWhile`. Теперь желаемые цифры могут быть найдены как последние цифры чисел, содержащихся в списке; последняя цифра числа равна остатку от его деления нацело на десять. Полученные цифры располагаются в обратном порядке, что легко поправило с помощью функции `reverse`. В завершение, элементы списка (цифры, т.е. величины типа `Int`) нужно преобразовать в символы (тип `Char`), что мы осуществим при помощи функции `digitChar`, рассмотренной на странице 52. Ниже приводится пример, включающий в себя все вышеперечисленные шаги:

```
                5678
              ↓  iterate (‘div‘ 10)
[5678, 567, 56, 5, 0, 0, 0, ...
              ↓ takeWhile (/= 0)
      [5678, 567, 56, 5]
              ↓ map (‘rem‘ 10)
         [8, 7, 6, 5]
              ↓ reverse
         [5, 6, 7, 8]
              ↓ map digitChar
   ['5', '6', '7', '8']
```

Функция `intString` может быть определена просто как объединение этих пяти шагов. Заметим, что функции написаны в обратном

порядке, потому что оператор последовательного выполнения (.) есть композиция функций (знакомая нам функция ‘after‘):

λ
```
intString :: Int -> [Char]
intString  = map digitChar
             . reverse
             . map (‘rem‘ 10)
             . takeWhile (/=0)
             . iterate (‘div‘ 10)
```

Этот промер еще раз подчеркивает, что *функциональное программирование — программирование посредством функций*!

Получение списка всех простых чисел

На странице 98 была определена функция `prime`, предназначенная для проверки простоты числа. С ее помощью можно получить бесконечный список простых чисел следующим образом:

H
```
filter prime [2 ..]
```

Функция `prime` предварительно находит все делители числа. Если число достаточно большое, то для проверки того факта, что число не является простым, потребуется значительное время

Разумнее будет использование функции `iterate` для получения более эффективной алгоритма. Этот метод известен с древних времен и носит название «решето Эратосфена». Работа метода также начинается с построения бесконечного списка `[2 ..]`:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...
```

Первое число в этом списке — 2, является простым и должно быть помещено в список простых чисел. Затем число 2 и все кратные ему числа (4, 6, 8, 10 ...) удаляются из списка:

```
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21, ...
```

И снова, первое число в списке — 3 — простое. Это число и все числа, кратные ему удаляются из списка:

```
[5, 7, 11, 13, 17, 19, 23, 25, 29, 31,  ...
```

Повторяем процесс, начав с 5:

```
[7, 11, 13, 17, 19, 23, 29, 31, 37, 41, ...
```

И так далее. Функция «вычеркивания кратных первому элементу» всегда применяется к предыдущему результату. Процесс итерации начинается со списка `[2 ..]`:

```
iterate crossout [2 ..]
   where crossout (x:xs) = filter (not . multiple x ) xs
         multiple x y    = divisible y x
```

λ

(Число `y` кратно `x`, если `y` делится на `x`). Входной аргумент есть бесконечный список, а результат является *бесконечным списком бесконечных списков*. Получаемый суперсписок выглядит примерно так:

```
[ [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...
, [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, ...
, [5, 7, 11, 13, 17, 19, 23, 25, 29, 31,  ...
, [7, 11, 13, 17, 19, 23, 29, 31, 37, 41, ...
, ...
```

Этот список никогда не удастся увидеть целиком — при попытке получить его, вы сможете увидеть только начало первого списка. Но можно увидеть фрагмент списка: нужные нам простые числа являются первыми элементами списков и могут быть получены при помощи функции `head`, примененной к каждому подсписку:

```
primenums   :: [Int]
primenums = map head (iterate crossout [2 ..])
   where crossout (x:xs) = filter (not . multiple x ) xs
         multiple x y = divisible y x


divisible     :: Int -> Int -> Bool
divisible m n  = m `rem` n == 0
```

λ

Благодаря отложенным вычислениям только та часть каждого списка обрабатывается, которая требуется для получения желаемого ответа. Для того чтобы найти следующее простое число, находятся дополнительные элементы каждого списка, но ровно столько, сколько необходимо.

```
---> primenums
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,
73,79,83,89,97,101,103,107,109,113,127,131,137,139,149,
151,157,163,167,173,179,181,191,193,197,199,211,223,227,
229,233,239,241,251,257,263,269,271,277,281,283,293,307,
311,313,317,331{Interrupted!}
```

H

Зачастую бывает сложно (как в этом примере) определить, что вычисляется в текущий момент. Но в этом и нет необходимости: при разработке программ достаточно знать, что бесконечные списки

существуют; порядок вычисления же будет автоматически оптимизирован с помощью стратегии отложенных вычислений.

«Псевдослучайные» числа

Числа, которые выбираются случайным образом, находят множество полезных применений: при моделировании на компьютере естественных явлений (например, случайные столкновения частиц в ядерной физике или появление людей в очереди через случайные промежутки времени), для решения сложных задач численного анализа (таких, как метод Монте-Карло нахождения площадей), при тестировании компьютерных программ в качестве исходных данных.

Сразу после появления компьютеров начались исследования эффективного способа получения случайных чисел. Были предложены десятки методов генерации таких чисел. Многие из ранних генераторов (датчиков) случайных чисел работали недостаточно эффективно (то есть, получаемая с их помощью последовательность чисел не удовлетворяла разнообразным статистическим критериям). Дональд Кнут в своей широко известной книге «Искусство программирования» [4] приводит исчерпывающий обзор и анализ различных генераторов. Там же приводится достаточно простой, но вполне качественный алгоритм для получения чисел, «случайно» распределенных на интервале от нуля до 2147483647. Ниже приводится его реализация на языке Haskell:

```
next_seed :: Int -> Int
next_seed n =
      case test>0 of
          True  -> test
          False -> test + 2147483647
      where
        test = 48271 * lo - 3399 * hi
        hi   = n `div` 44488
        lo   = n `mod` 44488
```

Начальное значение `n` может быть выбрано произвольно. Для получения различных последовательностей следует применить данный алгоритм с другому начальному значению. Вызов функции `next_seed` дает первое «псевдослучайное» число, передаваемое ей снова в качестве нового аргумента. Для реализации этого подхода удобно использовать функцию `iterate`:

```
rand :: [Int]
```

```
rand  = iterate next_seed 23765492
```

Из бесконечного списка, генерируемого функцией **rand**, следует выбрать требуемое число начальных элементов при помощи функции **take**, например,

```
---> take 5 rand                                                    Н
[23765492,427796834,2125708109,1139992232,1444060144]
```

Для получения списка случайных чисел, меньших некоторого заданного значения (например, 100), можно брать остаток от деления случайного числа на данное.

```
rand :: [Int]                                                       λ
rand = [ x `mod` 100 | x <- iterate next_seed 23765492 ]
```

В результате вызова так определенной функции получается бесконечных список чисел в диапазоне от 0 до 99.

```
---> take 20 rand                                                   Н
[92,34,9,32,44,51,46,21,5,54,19,70,70,99,5,69,88,3,17,3]
```

▶ *Упражнение VI.3.2*

Существует большое число научно обоснованных критериев добротности генераторов случайных чисел, описанных, например, в [**4**]. Но даже без соответствующей теории интуитивно ясно, что в случайной последовательности количество четных и нечетных чисел должно быть приблизительно равно. Напишем программу, подсчитывающую долю четных и нечетных чисел в последовательности, получаемой с помощью описанного выше генератора случайных чисел.

Сначала определим функцию **quantum**, возвращающую пару чисел — число нечетных и четных чисел в списке, являющимся аргументом данной функции:

```
quantum       :: [Int] ->(Int, Int)                                 λ

quantum [ ]    = (0, 0)
quantum (x:xs) =
          case odd x of
                True  -> (u+1,   v)
                False -> (  u, v+1)
          where (u, v) =  quantum xs
```

Подсчитаем с ее помощью число четных и нечетных чисел среди первой тысячи целых чисел:

H
```
---> quantum [1 .. 1000]
(500,500)
```

Функция `proportion` подсчитывает долю каждого элемента пары от их суммарного количества:

λ
```
proportion  :: (Int, Int) -> (Float, Float)

proportion (0, 0) = (0.0, 0.0)
proportion (0, _) = (0.0, 1.0)
proportion (_, 0) = (1.0, 0.0)
proportion (a, b) = (fromInt a / c, fromInt b / c)
  where c = fromInt(a+b)
```

Остается применить композицию этих функций к списку случайных чисел, полученного с помощью функции `rn`:

λ
```
rn   :: Int -> [Int]
rn n  = take n rand

test :: Int -> (Float, Float)
test  = proportion . quantum . rn
```

Обратите внимание, что функция `test` определена в чисто функциональном стиле — как композиция трех функций. Теперь можно провести тестирование полученной последовательности:

H
```
---> test 500
(0.508,0.492)
---> test 5000
(0.5008,0.4992)
```

Как видим, доли четных и нечетных чисел достаточно близки.

◀

▶ *Упражнение VI.3.3*

Рассмотрим еще один алгоритм сортировки списков — так называемую «быструю» сортировку (*quicksort*):

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

Получите 400 чисел при помощи генератора случайных чисел, описанного на стр. 144. Включите статистическую опцию интерпретатора и убедитесь в том, что название этой функции отражает ее производительность, сравнив ее с другими сортировками, рассмотренными ранее:

```
---> quicksort (take 400 rand)
 . . .
(85142 reductions, 114637 cells)
---> msort (take 400 rand)
 . . .
(141649 reductions, 189366 cells)
---> isort (take 400 rand)
(403054 reductions, 526123 cells, 2 garbage collections)
```

Н

При обработке последнего вызова Hugs дважды использовал механизм «встроенной сборки мусора» (garbage collection), т. е. удалял из памяти данные, которые не требуются для дальнейших вычислений. Подобного рода технику автоматического освобождения не используемых участков памяти используют многие современные языки программирования, например, Java, Ruby.

◀

Вопросы и задания

VI.3.1

Измените функцию `rand` таким образом, чтобы можно было бы задавать начальное значение датчика в качестве параметра функции. Проведите несколько экспериментов, определяя доли четных и нечетных чисел в последовательностях, полученных при разных начальных значениях.

4. Кортежи

4.1. Использование кортежей

Все элементы списка обязаны иметь один и тот же тип. Для хранения величин различных типов используют кортежи (тьюплы). Часто о кортежах говорят как об упорядоченных множествах, так как для них важен порядок размещения элементов, например,

`(True, 23)` и `(23, True)` являются совершенно различными кортежами. В прелюдии определены несколько функций для работы с тьюплами:

```
fst             :: (a,b) -> a
fst (x,_)        = x

snd             :: (a,b) -> b
snd (_,y)        = y
```

По аналогии можно задать функции для выделения составных частей троек:

```
fst3          :: (a, b, c) -> a
fst3(x, _, _) = x
snd3          :: (a, b, c) -> b
snd3(_, y, _) = y
thd3          :: (a, b, c) -> c
thd3(_, _, z) = z
```

Эти функции полиморфны, так как они могут применяться к двойкам и тройкам, содержащим данные любого типа. Однако можно определять функции на тьюплах фиксированного типа, например:

```
f          :: (Int, Char) -> [Char]
f (n, c)    = intString n ++ [c]
```

Если группируются вместе две величины, имеющие одинаковый тип, то вы можете использовать список для их хранения. Тем не менее, ничто не мешает объединить их в кортеж. Точка на плоскости, например, описывается двумя числами типа `Float`. Ее можно представить и как список из двух элементов и как пару чисел. В обоих случаях можно определить функции для работы с точками, например, функцию, определяющее расстояние от точки до начала координат. Ниже представлены две функции: `distanceL`, определенную на списке, и `distanceT`, использующую кортежи:

```
distanceL       :: [Float] -> Float
distanceL [x, y] = sqrt (x*x + y*y)

distanceT       :: (Float, Float) -> Float
distanceT (x, y) = sqrt (x*x + y*y)
```

При корректном вызове этих функций нет никаких различий в их использовании. Но может так случиться, что в результате ошибки

ввода или логической ошибки функция будет вызвана с тремя параметрами. В случае `distanceT` ошибка будет обнаружена еще на этапе анализа программы: кортеж из трех чисел относится совсем к другому типу данных.

```
---> distanceL [3, 4]                                          Н
5.0
---> distanceT (3, 4)
5.0
---> distanceT (3, 4, 6)
ERROR - Type error in application
*** Expression     : distanceT (3,4,6)
*** Term           : (3,4,6)
*** Type           : (a,b,c)
*** Does not match : (Float,Float)
```

При использовании `distanceL` ошибка также будет обнаружена, но уже на этапе выполнения функции. При сопоставлении входных данных с образцом списка (содержащим всего два элемента!) в определении функции будет выявлено их несоответствие:

```
---> distanceL [3, 4, 6]                                       Н

Program error: {distanceL [Num_fromInt instNum_v47 3,
Num_fromInt instNum_v47 4,Num_fromInt instNum_v47 6]}
```

В этом примере использование кортежа вместо списка является более предпочтительным, так как при вызове функции `distanceT` ошибка выявлялись на более ранней стадии.

Использование кортежей позволяет легко определять функции, возвращающие сразу несколько значений. В случае с несколькими аргументами еще может выручить использование механизма карринга, но для функций с несколькими результатами альтернативы использования кортежей нет! Примером подобной функции может служить следующая функция, разбивающая список на две части:

```
splitAt     :: Int -> [a] -> ([a], [a])                        λ
splitAt n xs = (take n xs, drop n xs)
```

Следующая версия этой функции более эффективна:

```
splitAt              :: Int -> [a] -> ([a], [a])               λ
splitAt 0 xs          = ([], xs)
splitAt n []          = ([], [])
splitAt (n+1) (x:xs) = (x:ys, zs)
```

```
    where (ys, zs) = splitAt n xs
```

Вызов `splitAt 2 "Haskell"` возвратит тьюпл `("Ha", "skell")`. В рекурсивном правиле этой функции показано, как можно использовать образцы при работе с картежами.

4.2. Кортежи и списки

Кортежи часто являются элементами списка. Список, содержащий пары, используют для организации поиска (словари, телефонные справочники и т.п.). Функция поиска `search` может быть легко записана с помощью образца, являющегося непустым списком с первым элементом в виде пары:

λ
```
search   :: Eq a => [(a, b)] -> a -> b
search ((x, y):ts) s
                | x == s   = y
                | otherwise = search ts s
```

Заданная таким образом функция полиморфна, так как может работать с списком пар любого произвольного вида. Однако, элементы приходится сравнивать между собой, поэтому требуется принадлежность первых элементов пар классу `Eq`. Функция поиска может быть легко частично параметризована указанием специфического списка, например,

λ
```
telephoneNr  = search telephoneDirectory
translation  = search dictionary
```

Списки `telephoneDirectory` и `dictionary` могут определяться отдельно.

Другая функция, использующая пары, нам уже известна — это функция `zip` (стр. 124):

λ
```
zip              :: [a] -> [b] -> [(a,b)]
zip [ ]    ys     = [ ]
zip (x:xs) [ ]    = [ ]
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

Примером функции высшего порядка для работы 2-тьюплами служит функция `zipWith` (стр. 126):

λ
```
zipWith        :: ( a -> b ->c) -> ([a] -> [b] ->[c])
zipWith f [] ys = []
zipWith f xs [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

Эта функция применяет функцию (с двумя параметрами) ко всем элементам двух списков. Отметим, что функции `zip` и `zipWith` в некоторой степени являются аналогами функции `map`, которая применяет функцию (с одним параметром) ко всем элементам списка.

Можно дать другое определение функции `zip` — как частично параметризованную функцию `zipWith`:

```
zip  = zipWith makePair
  where makePair x y = (x, y)
```

λ

▶ *Упражнение VI.4.1*

Пусть пара вида `(String, Int)` хранит информацию об имени и возрасте человека. Напишем программу, определяющую по данному списку таких пар имя самого юного человека.

Сначала напишем функцию, которая из двух таких пар выбирает ту, которая содержит информацию о более юном человеке:

```
younger :: (String, Int) -> (String, Int) -> (String, Int)
younger p1@(_, age1) p2@(_, age2) = if age1 < age2
                                    then p1 else p2
```

λ

Обратите внимание на использование `@`-образцов и анонимных переменных в образцах соответствия данной функции.

Рекурсивная функция `youngest` определяет тьюпл, содержащий данные самого молодого человека в непустом списке таких пар:

```
youngest :: [(String, Int)] -> (String, Int)
youngest (p : []) = p
youngest (p:ps) = younger p (youngest ps)
```

λ

Теперь создадим список с данными о членах некой семьи и определим основную функцию `main`, печатающую имя самого молодого члена этой семьи:

```
family :: [(String, Int)]
family = [("Степан", 7), ("Ксения", 3),
          ("Павел", 39), ("Нина", 37)]

main =
   let (name, _) = youngest family
   in putStr ("Самый молодой член семьи - " ++ name ++ ".")
```

λ

Функция `putStr` печатает свой аргумент — строку. Посмотрите на результат работы функции:

H
```
---> main
Самый молодой член семьи - Ксения.
```

◀

4.3. Кортежи и карринг

Кортежи позволяют использовать несколько параметров у функции без привлечения механизма карринга. Посмотрим на следующее определение:

λ
```
add (x, y) = x + y
```

Запись этой функции выглядит как запись функции, принятая в математике. Большинство людей, глядя на такое определение, скажут, что задана функция от двух аргументов и что параметры «естественно» расположены внутри скобок. Но мы то знаем — у этой функции один параметр: кортеж из двух чисел, в ее определении присутствует шаблон кортежа.

Тем не менее, использование механизма карринга предпочтительнее работы с тьюплами. Каррированную функцию можно частично вычислить (параметризовать), а функцию, использующую тьюпл — нет. Поэтому все стандартные функции, зависящие более чем от одного параметра, используют в каррингговой форме.

Напомним, что в прелюдии содержаться две функции преобразования функций (т. е. берущие функции в качестве аргумента и возвращающие функцию):

λ
```
curry             :: ((a,b) -> c) -> (a -> b -> c)
curry f x y        = f (x,y)

uncurry           :: (a -> b -> c) -> ((a,b) -> c)
uncurry f (x, y)   = f x y
```

5. Синонимы

Перед тем как продолжить, уместно ввести простой способ записи, позволяющий давать альтернативные имена типам данных. Задаваемые с помощью этого способа имена яснее отражают цели, ради которых вводится тип данных.

Когда в нескольких функциях используется один и тот же тип кортежа или списка, становится несколько утомительным описание этих типов в объявлениях функций. Предположим, что нам требуется написать набор функций для операций с точками на плоскости: `distance` — расстояние от точки до начала координат, `difference` — расстояние между двумя точками, `area_polygon` — площадь многоугольника, и `transf_polygon` — функцию трансформации многоугольника. Их объявления таковы:

```
distance     :: (Float, Float) -> Float                              λ
difference   :: (Float, Float) ->(Float, Float) -> Float
area_polygon :: [(Float, Float)] -> Float
transf_polygon :: ((Float, Float) -> (Float, Float))
                  -> [(Float, Float)] -> [(Float, Float)]
```

Как видим в объявлениях у нас постоянно повторяется описание пары, содержащей координаты отдельной точки. В таком случае удобно ввести новое *определение типа*. Оно позволит яснее отразить назначение функций:

```
type Point = (Float, Float)                                          λ
```

Мы дали новое определение `Point` (точка) для пары чисел типа `Float`. С использованием такого определения типа объявления наших функций стали яснее отражать назначение этих функций:

```
distance       :: Point -> Float                                     λ
difference     :: Point -> Point -> Float
area_polygon   :: [Point] -> Float
transf_polygon :: (Point -> Point) -> [Point] -> [Point]
```

Можно еще определить и «многоугольник»:

```
type Polygon = [Point]                                               λ

area_polygon   ::  Polygon -> Float
transf_polygon :: (Point -> Point) -> Polygon -> Polygon
```

Есть несколько вещей, о которых нужно помнить при определениях типа:

- слово `type` специально зарезервировано для этого случая;
- имя нового определения типа должно начинаться с заглавной буквы (так как это константа, а не переменная);
- *объявление* типа определяет тип функции, *определение* типа определяет новое имя для типа.

Интерпретатор рассматривает новое определение типа просто как аббревиатуру (сокращение). Он не считает, что любое выражение тип `(Float, Float)` есть тип `Point`. Возможно использование и двух различных имен для одного и того же типа, например,

λ
```
type Point   = (Float, Float)
type Complex = (Float, Float)
```

Здесь `Point` в точности тоже самое, что и `Complex` и `(Float, Float)`. Поэтому такие определения типа и называют **синонимами**. В дальнейшем вы узнаете как действительно ввести *новый* тип данных.

Пожалуй наиболее известный тип синоним — это `String`. Определение этого синонима в преамбуле вводит тип `String` как список элементов типа `Char`:

λ
```
type String = [Char]
```

Строки записываются в специальном виде: символы, составляющие строку, заключаются в двойные кавычки. Различие между 'a' и "a"в том, что первое есть символ, а последнее — список символов, состоящий только из одного элемента.

Строки являются типом синонимом и не являются отдельным, самостоятельным типом данных. Они наследуют свойства, присущие всем спискам. Сравнение строк следует нормальному лексикографическому порядку. Например,

H
```
---> "hello" < "hallo"
False
---> "Jo" <"Joanna"
True
```

Чаще всего нам хочется печатать строки *посимвольно*. Это означает, что (1) двойные кавычки не выводятся и (2) некоторые символы (такие как `\n`) интерпретируются как специальные команды. Особенности ввода и вывода будут рассмотрены в главе 8, сейчас достаточно отметить, что Haskell обладает встроенной функцией `putStr`, предназначенной для печати строк. В случае применения функции `putStr` к строке, она будет печататься посимвольно.

H
```
---> putStr "Hello World"
Hello World
---> putStr "Hello \nWorld"
Hello
World
```

▶ *Упражнение VI.5.1*

Вернемся к задаче нахождения корней квадратного уравнения. На странице 47 дано определение функции `abcFormula`, предназначенной для тех же целей:

```
abcFormula       :: Float -> Float -> Float -> [Float]
abcFormula a b c  = [(-b + sqrt(b*b - 4.0*a*c))/(2.0*a),
                     (-b - sqrt(b*b - 4.0*a*c))/(2.0*a)
                    ]
```

Эта функция представлена в карринговой форме, она последовательно берет три аргумента и возвращает список корней. Определим новую функцию `roots`, предназначенную для тех же целей, но получающую тройку чисел — коэффициентов уравнения, и возвращающую пару корней. Особо выделим случай, когда уравнение не имеет решений.

Объявление типа этой функции будет таким:

```
roots    ::  (Float, Float, Float) ->  (Float, Float)
```

Для лучшего отражения сути нашей задачи введем два типа синонима — для коэффициентов уравнения и его корней:

```
type Coeffs = (Float, Float, Float)
type Roots  = (Float, Float)
```

Это описание дает новые имена для уже существующих типов. Объявление функции `roots` теперь можно записать так:

```
roots    :: Coeffs -> Roots
```

Новый тип описания короче и более информативен. Причем определение типа `Roots` показывает, что у квадратного уравнения не может быть больше двух корней.

В определении функции с помощью охранных выражений выделим случай, когда уравнение не имеет корней. Если корни совпадут, то оба элемента пары будут одинаковы. Для того, чтобы избежать повторных вычислений введем локальные определения для некоторых выражений. Ниже приведен скрипт, содержащий полностью определение функции `roots`:

```
type Coeffs = (Float, Float, Float)
type Roots  = (Float, Float)

roots    :: Coeffs -> Roots

roots (a, b, c)
```

```
      | d >= 0  = ( (-b + d')/a', (-b - d')/a')
      | otherwise     = error "Корней нет!"
       where d = b*b - 4.0*a*c
             d' = sqrt d
             a' = 2.0 * a
```

Теперь можно убедится в корректной работе полученной программы. Сначала попробуем предложить программе данные неподходящего типа, затем различные тройки коэффициентов:

H
```
---> roots (1.0, 2.0)
ERROR - Type error in application
*** Expression     : roots (1.0,2.0)
*** Term           : (1.0,2.0)
*** Type           : (a,b)
*** Does not match : (Float,Float,Float)
---> roots (1.0, 2.0, 1.0)
(-1.0,-1.0)
---> roots (1.0, -6.0, 5.0)
(5.0,1.0)
---> roots (1.0, 2.0, 5.0)

Program error: Корней нет!
```

◀

▶ *Упражнение VI.5.2*

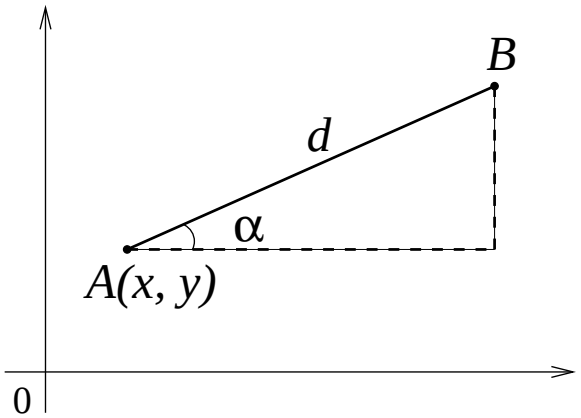
Определим функцию `move`, которая берет число, являющееся расстоянием, угол и пару координат точки на плоскости, после чего вычисляет координаты новой точки плоскости, положение которой определяется расстоянием и углом.

Мы можем ввести типы синонимы для величин, используемых в задаче следующим образом:

λ
```
type Position = (Float, Float)
type Angle    = Float
type Distance = Float
```

Заметьте, что и `Angle` и `Distance` являются синонимами `Float`. Теперь можно определить `move` так:

λ
```
move     :: Distance -> Angle -> Position ->Position
move d a (x, y) = (x + d * cos a, y + d * sin a)
```



Тригонометрические функции `cos` и `sin` являются примитивами. Так же как тип описания функции `roots`, такое описание `move` короче и помогает понять назначение функции `move`.

◀

Тип синоним может быть параметризован одним или большим числом типов переменных, аналогично определениям других типов данных. Вот несколько корректных примеров использования синонимов:

```
type Pairs a     = (a, a)
type Antomorph a = a -> a
type Flag a      =  (a, Bool)
```

λ

Два типа синонима не могут быть объявлены в терминах друг друга, так как каждый синоним должен выражаться в терминах существующих типов. С другой стороны, вполне законно объявление одного синонима в терминах другого. Например,

```
type Bools = Pairs Bool
```

λ

Здесь `Bools` — тип синоним для `Pairs Bool`, т. е возвращает синоним для пары `(Bool, Bool)`.

5.1. Рациональные числа

Одной из задач, в которых удобно использовать пары чисел, является задача осуществления математических действий с рациональными числами. Как известно, число представимое в виде несократимой дроби $\frac{p}{q}$, где p и q — целые числа, называется рациональным. Множество всех рациональных чисел обозначается как **Q**. Невозможно использование типа `Float` для представления

дробей, ведь все вычисления должны быть абсолютно точными: так $\frac{1}{2}+\frac{1}{3}$ должно равняться $\frac{5}{6}$, а не 0.833333.

Дроби могут быть представлены как пара целых чисел: числитель и знаменатель. Отсюда очевидное определение типа:

λ
```
type Fraction = (Int, Int)
```

Для наиболее часто используемых дробей можно ввести специальные имена:

λ
```
qZero   = (0, 1);   qOne   = (1, 1);   qTwo     = (2, 1)
qHalve  = (1, 2);   qThird = (1, 3);   qQuarter = (1, 4)
```

Нам требуется написать несколько функций, позволяющих осуществлять основные арифметические операции с дробями:

λ
```
qMul     :: Fraction -> Fraction -> Fraction
qDiv     :: Fraction -> Fraction -> Fraction
qAdd     :: Fraction -> Fraction -> Fraction
qSub     :: Fraction -> Fraction -> Fraction
```

Основная проблема состоит в том, что одна и та же дробь может представляться в виде различных дробей, например, половина может быть представлена не только как `(1, 2)`, но и виде `(2, 4)` или `(17, 34)`. При умножении четверти на два результат должен равняться половине. Для решения этой проблемы определим функцию `simplify`, предназначенную для упрощения дробей. Применяя эту функцию после любой операции с дробями, мы будем получать результат в одном и том же виде: результат сравнения половинки с произведением четверти на два должен равняться `True`.

Функция `simplify` делит числитель и знаменатель дроби на их *наибольший общий делитель* (НОД, gcd, greatest common divisor). Наибольший общий делитель двух чисел есть наибольшее число, на которое делится как числитель, так и знаменатель дроби. Для учета знака дроби воспользуемся функцией `signum`. Определение `simplify` таково:

λ
```
simplify (n,d) = ((signum d*n) `div` g, abs d `div` g)
    where g = gcd n d
```

Простое определение `gcd x y` мы получим, если среди делителей числа `x` выберем наибольший, являющийся при этом делителем числа `y`, для чего воспользуемся функциями `divisors` и `divisible`, рассмотренными на стр. 97.

λ
```
gcd x y = last (filter (divisible y') (divisors x'))
```

```
      where x' = abs x
            y' = abs y
```

Полученное определение функции gcd не эффективно, и в прелюдии приводится другая реализация этой функции, использующая алгоритм Евклида нахождения НОД. Алгоритм Евклида основан на следующих свойствах: для всех a и b, больших или равных 0, выполнены соотношения

```
НОД(a, b) = НОД(a - b, b)= НОД(a, b - a);
НОД(a, 0) = НОД(0, a) = a
НОД(0, 0) не определен
```

Определение gcd, учитывающее алгоритм Евклида, таково:

```
gcd 0 0          = error "gcd 0 0 is undefined"
gcd x y          = gcd' (abs x) (abs y)
                   where gcd' x 0 = x
                         gcd' x y = gcd' y (x `rem` y)
```

λ

Подготовив функцию simplify, можно приступать к разработке функций, предназначенных для арифметических операций с дробями. Для умножения двух дробей следует перемножить их числители и знаменатели ($\frac{2}{3} \cdot \frac{5}{4} = \frac{10}{12}$) и упростить результат:

```
qMul (x, y) (p, q)  = simplify (x*p, y*q)
```

λ

Деление сводится к умножению на обратную величину:

```
qDiv (x, y) (p, q)  = simplify (x*q, y*p)
```

λ

Перед сложением двух дробей их сначала приводят к общему знаменателю, после чего складывают числители полученных дробей ($\frac{1}{4} + \frac{3}{10} = \frac{1 \cdot 10}{4 \cdot 10} + \frac{3 \cdot 4}{4 \cdot 10} = \frac{10}{40} + \frac{12}{40} = \frac{22}{40}$). Отсюда получаем определения

```
qAdd (x, y) (p, q)  = simplify (x*q + y*p, y*q)
qSub (x, y) (p, q)  = simplify (x*q - y*p, y*q)
```

λ

Результат вычисления выводится в виде кортежа из двух элементов. Так как это не очень удобно, то можно определить функцию fractionString, преобразующую кортеж в строку:

```
fractionString          :: Fraction -> String
fractionString  (x, y)
    | y' == 1   = intString x'
    | otherwise = intString x' ++ ``/'' ++ intString y'
      where (x', y') = simplify (x, y)
```

λ

Учтите, что функция `fractionString` в таком виде способна преобразовывать только положительные дроби (см. определение функции `intString` на стр. 142).

▶ *Упражнение VI.5.3*

Изменим определение `fractionString` так, чтобы функция могла оперировать любыми рациональными числами. Для этого добавим функцию `signIntString`, преобразующую целое число в строку:

λ
```
signIntString :: Int ->  String
signIntString x
    | x == 0      = "0"
    | otherwise   = signumChar x ++ intString (abs x)
      where signumChar x
              | signum x == -1 = "-"
              | otherwise      =   []
```

Теперь мы можем представить в виде строки любое целое число:

H
```
---> signIntString 0
"0"
---> signIntString (-1234450)
"-1234450"
```

Полностью скрипт выглядит так:

λ
```
type Fraction = (Int, Int)

qZero, qOne, qTwo, qHalve, qThird, qQuarter :: Fraction
qZero   = (0, 1);   qOne   = (1, 1);   qTwo     = (2, 1)
qHalve  = (1, 2);   qThird = (1, 3);   qQuarter = (1, 4)

simplify :: Fraction -> Fraction
simplify (n,d) = ((signum d*n) `div` g, abs d `div` g)
               where g = gcd n d

qMul, qDiv, qAdd, qSub :: Fraction -> Fraction -> Fraction
qMul (x, y) (p, q)      = simplify (x*p, y*q)
qDiv (x, y) (p, q)      = simplify (x*q, y*p)
qAdd (x, y) (p, q)      = simplify (x*q + y*p, y*q)
qSub (x, y) (p, q)      = simplify (x*q - y*p, y*q)

fractionString         :: Fraction -> String
fractionString  (x, y)
    | x' == 0    = signIntString x'
```

```
    | y' == 1    = signIntString x'
    | otherwise  = signIntString x' ++ "/" ++ intString y'
        where (x', y') = simplify (x, y)

intString :: Int -> String
intString  = map digitChar
             . reverse
             . map (`rem` 10)
             . takeWhile (/=0)
             . iterate (`div` 10)

digitChar   :: Int -> Char
digitChar n  | n >= 0 && n < 10 = chr (n +ord '0')

signIntString :: Int -> String
signIntString x
    | x == 0      = "0"
    | otherwise   = signumChar x ++ intString (abs x)
      where signumChar x
              | signum x == -1 = "-"
              | otherwise      =   []
```

Теперь можно проводить арифметические операции с дробями:

```
---> fractionString (qMul qThird (-3,1))
"-1"
---> fractionString (qSub (1, 2) (1, 20))
"9/20"
---> fractionString (qAdd (1, 100) (1, 101))
"201/10100"
---> fractionString (qDiv qQuarter (91, -40))
"-10/91"
```

H

◀

Вопросы и задания

VI.5.1

Как известно, делить на ноль запрещено. Измените функцию `qDiv` так, чтобы при попытке разделить на ноль появлялось бы сообщение об ошибке.

VI.5.2

Многочлен степени n от переменной x может быть представлен, как список его коэффициентов. Предположим, что коэффициенты размещаются в списке в порядке убывания степеней числа x. Например, многочлен $3.1x^4+4.2x^3+9.3x+7.4$, который есть в точности многочлен $3.1x^4 + 4.2x^3 + 0.0x^2 + 9.3x + 7.4$, представляется списком `[3.1, 4.2, 0.0, 9.3, 7.4]`. Введите тип синоним `Poly` для списка коэффициентов многочлена, после чего

(а) напишите функцию, складывающую два многочлена вместе

```
padd  :: Poly -> Poly -> Poly;
```

(б) напишите функцию, вычисляющую значение многочлена для произвольного значения x

```
eval  :: Poly -> Float -> Float;
```

(в) напишите функцию для умножения двух многочленов

```
ptimes  :: Poly -> Poly -> Poly.
```

Глава VII
Haskell как язык ООП

1. Объектно-ориентированное программирование

Одной из сфер применения языка Haskell является использование его для быстрого прототипирования сложных программных систем. Как показал опыт таких компаний, как Ericsson, применившей этот язык при разработке телекоммуникационных приложений, и Software AG (экспертная система Natural Expert написана на Haskell), скорость создания ПО возрастала в 9–25 раз по сравнению с аналогичными проектами, где использовались C/C++. Написанный на Haskell прототип программной системы, позволяет проанализировать плюсы и минусы того или иного проекта и внести неизбежные изменения в проект системы еще на ранней стадии.

Отличительная особенность Haskell — возможность использования собственного синтаксиса для формального описания требований к проекту, что способствует безболезненному переходу от проектирования к программированию.

$$\boxed{\textit{ФП} + \textit{ООП} = \textit{Haskell}}$$

Haskell дает возможность писать простой, краткий и наглядный код, обеспечивая несложными средствами инкапсуляцию и полиморфизм. Так, рекурсивное описание процедуры быстрой сортировки, рассмотренной на стр. 163, занимает всего несколько строк. Использовать такую процедуру можно с данными любых типов, допускающими сравнение.

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

Аналогичный алгоритм, написанный, допустим, на языке C, будет ориентирован на работу со списками определенного типа, например, массивом чисел, в то время как приведенная выше функция на Haskell более универсальна: она отсортирует и списки чисел, и списки строк или символов.

Все задачи управления памятью при выполнении программы, написанной на Haskell, возлагаются на исполняющую среду. Программисту не надо заботиться о выделении и освобождении динамической памяти. Помимо этого среда реализует так называемое ленивое вычисление — в ходе выполнения программы происходит вычисление только тех данных, которые реально запрашиваются.

Недостатки Haskell, а именно, повышенные требования к памяти (из-за рекурсии) и не очень высокое быстродействие, не являются критичными на этом этапе разработки ПО.

Практически все современные программные разработки основываются на принципах объектно-ориентированного программирования, позволяющих повысить надежность программных компонент и их повторное использование, что сокращает суммарные затраты на их разработку. Сокрытие информации и абстрактность данных упрощают использование программного кода, написанного кем-то ранее. Именно Haskell, сочетающий в себе элементы функционального и объектно-ориентированного стиля разработки программ, является самым подходящим инструментом для создания прототипов программных систем.

Язык Haskell поддерживает объектно-ориентированную парадигму программирования. Одним из основных отличий ООП от других стилей программирования является то, что в обычном программировании программист ограничен встроенными типами данных, в то время как в случае ООП программист может определять свои собственные абстрактные типы данных.

1.1. Инкапсуляция и наследование

Объектно-ориентированные языки предоставляют два главных механизма абстракции: *инкапсуляцию* и *наследование*, которые могут использоваться при разработке программного обеспечения.

Благодаря инкапсуляции можно достичь *модульности*, позволяющей рассматривать объекты в качестве строительных блоков

сложной системы. Это свойство ООП способствует быстрому построению прототипа системы, причем окончательная доводка «быстрых и схематичных решений» откладывается на более позднее время.

Другое преимущество объектно-ориентированного подхода, которое часто считают главным, — возможность повторного использования кода. Наследование в этом отношении является очень ценным инструментом. Механизм наследования позволяет программисту изменять и настраивать поведение класса объектов без необходимости получать доступ к исходному коду реализации этого класса.

Объект можно рассматривать как воплощение (элемент) абстрактного типа данных. Чтобы использовать объект, клиент должен лишь знать, *что* объект делает, а не *как* это поведение реализуется. С точки зрения реализации на определенном языке программирования объект есть не что иное, как высокоуровневая организация данных.

Наследование представляет собой общий и мощный механизм повторного использования кода. Абстрактный тип данных обусловливает поведение некоторой совокупности сущностей. Используя механизм наследования для расширения определения некоторого типа, мы либо задаем новое поведение в дополнение к заданному, либо изменяем унаследованное, либо делаем и то и другое.

Условие, которое должно непременно соблюдаться при изменении унаследованного поведения, состоит в том, что измененные подобным образом объекты обязательно должны допускать их использование везде, где могут применяться объекты исходного типа.

Полиморфизм
- Абстрактное наследование
- Отношение уточнения и подтипы
- Различные типы полиморфизма
- Типовая абстракция
- Сокрытие представления
- Рекурсивные определения типов данных

Итак, можно сказать, что назначение объектов — реализация некоторых соглашений. Соглашение фиксируется в определении класса, экземпляром которого является данный объект. Наследование можно понимать как механизм реализации уточнения поведения, которое в конечном счете означает усовершенствование соглашения, регламентирующего поведение объекта.

1.2. Классы и перегрузка

С теоретической точки зрения сущность объектно-ориентированного программирования состоит в объединении *концепции абстрактных типов данных* с концепцией *полиморфизма*. Эти понятия можно считать абстрактным дополнением к более конкретным понятиям *инкапсуляции* и *наследования*.

Типизация, как мы уже отмечали ранее, представляет собой мощное средство для борьбы с ошибками в программах. Типизация может различаться по виду (*статическая* или *динамическая*) и по уровню (*сильная* или *слабая*). При статической типизации контроль типов осуществляется на этапе компиляции, а при динамической — на этапе выполнения. Haskell относится к языкам, поддерживающим статическую типизацию. Сильно типизированные языки связывают программиста довольно жесткими ограничениями. При моделировании реальной предметной области зачастую очень сложно «втиснуть» требуемую систему типов в рамки имеющихся типов. Однако, когда речь идет о разработке высоконадежных программ, нетипизированные средства, как правило, считают неудовлетворительными.

Тем не менее, даже в рамках строго типизированных средств, для того, чтобы сделать систему типов более удобной для практического использования, делают отступления от строгой типизации, разрешая применение перегруженных функций, приведение типов и значений, общих для нескольких типов.

Различают два вида полиморфизма.

Общий полиморфизм, в свою очередь, может принимать форму *включающего* (связанного с механизмом наследования) и *параметрического полиморфизма* (основанного на поддержке родовых типов).

Другой род полиморфизма, называемый *ad hoc* полиморфизм, представляет собой надстройку над неполиморфной моделью типизации и известен как перегрузка (*overloading*). Вот несколько примеров полиморфизма этого вида.

- Литералы `1`, `2`, ..., часто используют для представления целых чисел как с фиксированной, так и с произвольной точностью.
- Арифметические операторы, такие как `+`, определяются для работы с различными видами чисел.
- Операция проверки на равенство `==` работает как с числами, так и со многими другими (но не со всеми) типами.

Отметим, что такое перегруженное поведение различно для каждого типа (иногда поведение не определенно, что приводит к ошибке), тогда как при параметрическом полиморфизме тип не имеет никакого значения. В Haskell **класс** обеспечивает структурный способ управления *ad hoc*-полиморфизмом или перегрузкой.

Синтаксис языка Haskell предоставляет возможность определять классы. Типы данных в Haskell рассматриваются как экземпляры (instance) того или иного класса.

Объявление класса `class` вводит новый класс типов и перегружает операции, поддерживаемые в любом типе, являющимся экземпляром данного класса.

Класс (type class) в Haskell есть коллекция типов, распределяющих между собой набор функций и/или операторов.

Объявление экземпляра `instance` декларирует, что тип является экземпляром класса и включает определения перегруженных операций, называемых в соответствии с традициями ООП, *методами класса.*

Допустим, мы решили перегрузить операторы `(+)` и `negate` для типов `Int` и `Float`. Тогда следует ввести новый класс `Num`:

```
class Num a where
  (+)     :: a -> a -> a
  negate  :: a -> a
```

Подобное объявление читается так: «Тип `a` есть экземпляр класса `Num`, если он содержит (перегруженные) методы класса `(+)` и `negate`, соответствующих типов, определенные в нем.»

После этого мы можем определить `Int` и `Float` как экземпляры этого класса:

```
instance Num Int where
  x + y    = addInt x y
  negate x = negateInt x

instance Num Float where
  x + y    = addFloat x y
  negate x = negateFloat x
```

Функции `addInt`, `negateInt`, `addFloat` и `negateFloat` есть примитивы, но в другим случаях это могут быть и определяемые пользователем функции. Первая декларация в приведенном выше примере читается как: «Тип `Int` есть экземпляр класса `Num` с

соответствующими определениями (т. е. методами класса) для `+` и `negate`.»

Прежде чем приступать к рассмотрению вопроса о реализации новых абстрактных классов в языке Haskell, рассмотрим некоторые базовые классы этого языка (описание этих классов находится в файле `Prelude.hs`).

2. Встроенные классы

2.1. Класс `Eq`

Обзор классов начнем с простого, но очень важного класса `Eq`, известного как *класс равенства* (*equality*). Данный класс является коллекцией множества типов, допускающих проверку на равенство. Другими словами, если возможно использование операции сравнения на равенство (`==`) двух элементов указанного типа, то этот тип входит в состав класса `Eq`. Не все типы допускают проверку на равенство. Например, рассмотрим список типа `[Char -> Char]`, т. е. последовательность функций для работы с символами. Понятно, что проверить на равенство два таких списка невозможно. Несмотря на то, что существует математическая концепция равенства двух функций, в языке Haskell не допустимо применение оператора `==` к функциям, так как невозможно описать процесс их сравнения на равенство.

В прелюдии класс `Eq` вводится следующим определением:

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

Такое определение может быть прочитано как «тип `a` есть экземпляр класса `Eq`, если он поддерживает использование (перегруженного) оператора `==`, соответствующего типа, определенного в нем». Соглашение о том, что тип `a` должен быть экземпляром класса `Eq`, записывается в виде `Eq a`. Выражение `Eq a` носит название *контекст* (context). Контексты размещают перед типовыми выражениями. Например, в следующем объявлении типа оператора `==`, которое читается так: «Для любого типа `a`, являющегося экземпляром класса `Eq` функция `==` имеет тип `a -> a -> Bool`»:

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

▶ *Упражнение VII.2.1*

Определим функцию `elem`, выясняющую, входит ли тот или иной элемент в список. Это определение рекурсивно: в пустой список никакой элемент не входит, а в непустом списке проверяется на равенство первый элемент и, если сравнение неуспешно, то сравнение проводится со следующим элементом списка. Так как в определении участвует сравнение на равенство, то величины, участвующие в определении функции должны принадлежать типу, который есть экземпляр класса `Eq`:

```
elem                 :: (Eq a) => a -> [a] -> Bool
x `elem`  []          = False
x `elem` (y:ys)       = x == y || (x `elem` ys)
```

λ

Мы использовали здесь инфиксную форму функции `elem`.

Примените функцию к спискам различного типа. Что произойдет, если убрать из объявления функции контекст?

◀

Каким же образом указать, что тот или иной тип является экземпляром класса `Eq`? Это осуществляется с помощью *объявления (декларации) экземпляра* (*instance declaration*). Например,

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

λ

Подобная запись читается так «Тип `Integer` есть экземпляр класса `Eq`, с определением метода, соответствующего оператору `==`». Функция `integerEq` является примитивной функцией, проверяющей, равны ли два целых числа, но в общем случае любое выражение может использоваться в правосторонней части определения метода, а не только та или иная функция. Аналогичное определение позволяет сравнивать числа с плавающей точкой.

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

λ

▶ *Упражнение VII.2.2*

Создадим класс, аналогичный `Eq` с методом класса `eq`, проверяющим на равенство, и функцию `elem`, определенную на типах, входящих в этот класс. Так как функция `elem` уже определена в прелюдии, то первая строка скрипта должна скрыть ее прежнее определение:

```
import Prelude hiding (elem)
```

λ

Имена базовых классов (объявленных в прелюдии) *не могут быть «скрыты»*, поэтому мы дадим нашему классу другое имя:

λ
```
class Eq' a where
  eq :: a -> a -> Bool
```

Теперь можно определить функцию `elem`, используя объявленный выше метод `eq`:

λ
```
elem :: (Eq' a) => a -> [a] -> Bool
x `elem` [] = False
x `elem` (y:ys) = x `eq` y || x `elem` ys
```

Чтобы воспользоваться этой функцией определим несколько экземпляров нашего класса `Eq'`:

λ
```
instance Eq' Int where
 x `eq` y = abs (x-y) < 3  -- "приблизительно равны"

instance Eq' Float where
 x `eq` y = abs (x-y) < 0.1
```

Ограничения на объявления экземпляров класса

- Программа не может содержать более одного объявления экземпляра для данной комбинации типа и класса.
- Если тип объявлен как член класса, то он должен быть объявлен во всех суперклассах этого класса.
- Объявление экземпляра не требует обязательного задания методов для всех операторов класса. Когда метод не предоставлен при объявлении экземпляра и не получен по умолчанию в объявлении класса, выдается ошибка времени выполнения при вызове этого метода.
- Вы должны обеспечивать корректный контекст при объявлении экземпляра, так как контекст не выводится по умолчанию.

Создадим два списка, элементы которых принадлежат только что определенным типам (`Int` и `Float`), предназначенные для проверки работы функции `elem`:

λ
```
list1 :: [Int]
list1 = [1, 5, 9, 23]

list2 :: [Float]
list2 = [0.2, 5.6, 33, 12.34]
```

Теперь можно применить функцию `elem`, которая проверяет, имеется ли в списке число, «приблизительно» равное заданному:

Н

```
---> 2 `elem` list1
True
---> 100 `elem` list1
False
---> 0.22 `elem` list2
True
```

Внимание! Типы `Int` и `Float` теперь перегружены — без сигнатуры типа, указывающей, например, что целые числа являются экземплярами нашего типа `Int`, выражения, аналогичные `3 `eq` 3` не определены.

◀

2.2. Класс `Ord`

Класс `Ord` содержит в себе типы, допускающие сравнение своих элементов при помощи операций $(<)$, $(\leqslant)$, $(>)$ и $(\geqslant)$. Этот класс *расширяет* понятие класса `Eq`: он *наследует* (*inherits*) все операции класса `Eq`, добавляя вышеперечисленное множество операторов сравнения, а также функции нахождения максимального и минимального значения:

λ

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

Обратите внимание на контекст этого определения. Мы говорим, что `Eq` является *суперклассом* (или родительским классом) для класса `Ord` (в свою очередь `Ord` есть *подкласс* класса `Eq`) и любой тип, являющийся экземпляром класса `Ord`, также является экземпляром класса `Eq`. Есть определенное преимущество в том, что при определении этого класса использовался контекст: при объявлении типа функции, использующей операции как из класса `Ord`, так и из класса `Eq` можно указывать в качестве контекста только `(Ord a)`, так как `Ord` «подразумевает использование». Более важным преимуществом является то, что в качестве операций подкласса выступают уже существующие методы суперкласса. Например, в прелюдии метод `(<)` определяется так:

λ

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

Примером использования класса `Ord` служат функции сортировки, рассмотренные на стр. 118 и 163. Проверим, что они работают с различными типами данных, допускающих сравнение:

H
```
---> quicksort [34, 21, 456, 2, 8,-89]
[-89,2,8,21,34,456]
---> quicksort [True, False, False, 3<2]
[False,False,False,True]
---> quicksort "qwerty"
"eqrtwy"
---> quicksort ["qwer", "fght", "ss"]
["fght","qwer","ss"]
```

▶ *Упражнение VII.2.3*

Создадим класс, позволяющий сравнивать между собой величины данного класса и являющийся подклассом `Eq'`, с операторами `lt` и `le`:

λ
```
class (Eq' a) => Ord' a where
  lt, le :: a -> a -> Bool
  x `le` y = x `eq` y || x `lt` y
```

Заметим, что тип операторов `lt` и `le` есть

```
le, lt :: (Ord' a) => a -> a -> Bool
```

или, что эквивалентно,

```
le, lt :: (Eq' a, Ord' a) => a -> a -> Bool
```

Объявим `Int` экземпляром `Ord'` и определим для него оператор `lt` (отличающийся от стандартного!):

λ
```
instance Ord' Int where
 x `lt` y = x < y+1
```

Сравним между собой величины определенного нами типа `Int` (указывая их тип для устранения неопределенности):

H
```
---> (6 :: Int) `lt` (6 :: Int)
True
---> (6 :: Int) `le` (5 :: Int)
True
```

◀

2.3. Классы Show и Read

В зависимости от требований, предъявляемых к задаче, программисту может понадобиться распечатать числа в виде таблицы, картинки различного вида или отформатированный текст. Для достижения этих эффектов Haskell использует специальный класс `Show`, минимальное объявление которого выглядит так

```
class Show a where
    showsPrec       :: Int -> a -> String -> String
```

λ

Функция `showsPrec` предназначена для вывода величины типа `a`. Эта функция разработана гибкой и эффективной, что отразились на ее сложности. Мы не будем сейчас подробно останавливаться на возможностях этой функции, отметим только, что с ее помощью определяется более простая функция `show`:

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

λ

Функция `show` берет величину типа `a` и преобразует ее в строку. Например, тип `Bool` объявлен как экземпляр класса `Show` и функция `show` для логических величин выглядит так:

```
show False = "False"
show True = "True"
```

λ

Запрос на вычисление логического выражения и вывода его на терминал приводит к молчаливому применению функции `putStr.show`. Логическая величина преобразуется в строку посредством `show`, а результат печатается функцией `putStr`. Посмотрим как это применяется:

```
---> putStr (show True)
True
```

H

Подобное вычисление и печать логической величины возможна потому, что тип `Bool` объявлен как экземпляр класса `Show`. Типы `Char` и `String` также являются экземплярами класса `Show`, например,

```
---> show 'a'
"'a'"
```

H

Применение `putStr` к полученному результату приведет к удалению пары двойных кавычек и комбинация из трех символов 'a' будет напечатана:

```
---> putStr "'a'"
'a'
```

H

Некоторые подклассы `Show` уже обеспечены встроенными примитивами для печати. Например, целые числа можно печатать и величина `show n` для элемента `n` типа `Integer` есть список символов, являющийся десятичным представлением числа `n`. Тоже самое верно и для других типов чисел. Например,

H
```
---> show 42
"42"
---> show 43.2
"43.2"
```

Использование функции `show` позволяет нам контролировать внешний вид вывода:

H
```
---> putStr ("The year is " ++ show (2*1001))
The year is 2002
---> putStr (show 1 ++ show 2 ++ show 3)
123
---> putStr (show 1 ++ "\n" ++ show 2 ++ "\n" ++ show 3)
1
2
3
```

Если класс `Show` позволяет отобразить объект в виде строки, то класс `Read` решает обратную задачу. Основная идея использования данного класса состоит в том, что определяется парсер (лексический анализатор) для типа `a`, являющийся функцией, берущей строку и возвращающей список пар `(a, String)`. В прелюдии для таких функций введен тип-синоним:

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

Обычно парсер возвращает список, состоящий из одного элемента и содержащий величину типа `a`, которая была прочитана из входной строки, и оставшуюся часть строки, снова передаваемую затем парсеру. Если синтаксический анализ невозможен, то результатом будет пустой список, а если допускается более одного парсера (что теоретически возможно), то результирующий список будет содержать уже несколько пар. Стандартная функция `reads` является парсером для любого экземпляра класса `Read`:

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

Все встроенные классы обеспечены также более простой функцией `read`, объявленной так:

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

Строка, передаваемая в качестве аргумента функции `read` должна полностью обрабатываться парсером:

```
---> read "34" +3
37
---> head (read "[1,2]") +3
4
---> read ("1" ++"2"++"3") - 3
120
---> read "True" == True
True
---> read "34xx" +3

Program error: Prelude.read: no parse
```


В языке Haskell допускается **множественное наследование** — классы могут иметь более одного родительского класса (суперкласса). Следующее объявление создает класс `C`, наследующий операции классов `Eq` и `Show`:

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

Методы класса относятся к верхнему уровню объявлений в Haskell. Они разделяют пространство имен аналогично обычным переменным: одно и то же имя не может использоваться для обозначения и метода класса и переменной или для методов, принадлежащим различным классам.

Кроме ограничений, заданных для текущего класса, методы класса могут иметь свои дополнительные ограничения на используемый тип переменных. Например, в классе

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

метод `m` требует, чтобы тип `b` являлся экземпляром класса `Show`.

2.4. Класс `Enum`

Рассмотрим еще один класс, носящий название `Enum` (от enumerate — перечислять), который содержит типы, чьи элементы могут быть перечислены. Его минимально возможное определение таково:

```
class Enum a where
    toEnum              :: Int -> a
    fromEnum            :: a -> Int
```

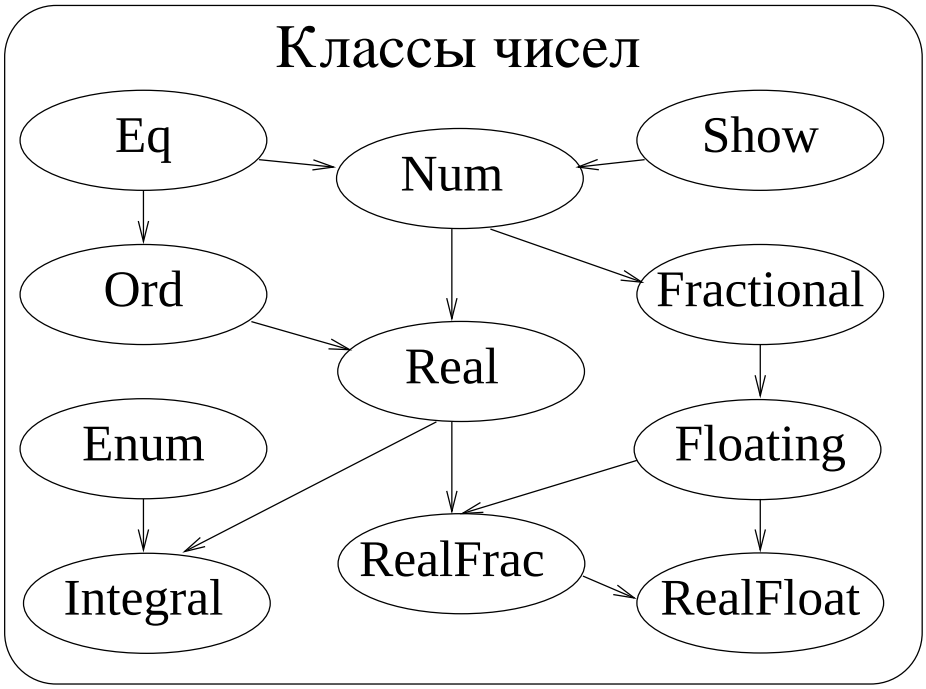
Тип, объявленный как экземпляр класса `Enum`, использует функции `toEnum` и `fromEnum` для преобразования между его элементами и типом `Int`. Функция `fromEnum` должна быть левообратимой (left-inverse) по отношению к функции `toEnum`, иначе говоря, должно выполняться соотношение

$$fromEnum(toEnum\ x) = x$$

Проверим это свойство для типа `Bool`, который объявлен в прелюдии как экземпляр класса `Enum`:

H
```
---> fromEnum(toEnum 1 :: Bool)
1
```

2.5. Классы чисел



Класс `Num` объединяет в себе все числа. Мы уже рассматривали его определение на стр. 167. В прелюдии он объявлен как подкласс классов `Eq` и `Show`:

λ
```
class (Eq a, Show a) => Num a where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a
    fromInt        :: Int -> a
```

На рисунке изображена иерархия классов чисел языка Haskell. Класс `Real` включает в себя типы действительных чисел, `Fractional` — дробные числа. Класс `Integral` содержит два типа целых чисел — `Int` и `Integer`, а `RealFloat` — типы `Float` и `Double`. Для более близкого знакомства с классом `Num` следует обратиться к файлу `Prelude.hs`

3. Алгебраические типы данных

3.1. Определение новых типов данных

В любом языке программирования, предназначенном для моделирования реальных объектов, обязательно должна быть возможность *определения новых, структурированных типов данных*, часто называемых *алгебраическими* типами данных. Современные функциональные языки программирования, и Haskell в том числе, также имеют такую возможность.

Отметим, что типы-синонимы, рассмотренные нами ранее, не создают новых типов, они лишь дают уже существующим типам более удобные, «говорящие» имена, которые делают объявления функций более короткими и информативными. Тип-синоним сохраняет все свойства породивших его типов. Например,

```
type Position = (Float, Float)
type Angle    = Float
type Distance = Float

move         :: Distance -> Angle -> Position -> Position
move d a (x,y) = (x + d * cos a, y + d * sin a)
```

Значение оператора `==` на `Angle` означает тоже самое, что и `==` на `Float`. На практике же, возможно нам потребуется какое-либо другое определение. Например, кто-то захочет объявить равенство на `Angle`, как равенство по модулю 2π. Конечно, можно было бы написать специальную функцию для сравнения двух величин типа `Angle`, но удобнее было бы использовать именно оператор `==`, что невозможно сделать оставаясь в рамках типа-синонима.

3.2. Перечисляемые типы

В языке Haskell имеется возможность определения новых структурированных типов данных (*datatype*) путем перечисления

всех их возможных значений. Так тип `Bool` может быть задан с помощью следующего *определения типа данных*:

λ
```
data Bool = False | True
```

Эта декларация закрепляет имя `Bool` за типом данных, который может быть представлен двумя величинами `True` и `False`. Структурированные величины «строятся» с помощью функций-конструкторов. В нашем примере два конструктора — константные функции (т. е. имеющие нулевое число параметров) `True` и `False`. Их можно использовать в качестве образцов соответствия в функциях. При определении функций с логическими аргументами `True` и `False` выступают в качестве образцов сопоставления, например, функция отрицания определяется так:

λ
```
not       :: Bool -> Bool
not False  = True
not True   = False
```

Два уравнения, задающие функцию `not` используются компьютером как правила переписывания для упрощения выражений вида `not e`. Сначала упрощается выражение `e`. Если результат равен `False`, то используется первое уравнение, а если `True` — то второе.

При объявлении нового типа данных мы использовали *конструктор типа* `Bool`. В этом примере `Bool` есть нульмерный конструктор, он не зависит от параметров. Конструктор типа `Bool` может появляться *только* в объявлениях типа и не может быть частью того или иного выражения. Запись, подобная `x = Bool`, не имеет смысла. С другой стороны, `True` и `False` являются (нульмерными) *конструкторами данных*. Они появляются в выражениях или в шаблонах (образцах соответствия). Запись, аналогичная, `x :: True` — ошибочна. Конструкторы данных могут использоваться только при определении данных.

Подобным образом можно было бы определить и тип `Char`, перечислив все его значения. Однако задание функций, использующих сопоставление с образцом для величин полученного типа, заняло бы значительное время, так как количество возможных значений данного типа велико.

Рассмотрим другой пример. Предположим, что программа должна оперировать с днями недели. Возможным вариантом решения задачи является представление дней недели целыми числами от 0 до 6 с дальнейшим введением типа-синонима

```
type Day = Int
```

Для любой функции, использующей определенный таким образом тип `Day`, нам придется включать предварительную проверку аргумента функции на принадлежность указанному выше диапазону:

```
f :: Day -> String
-- pre 0 <= d <= 6
```

Подобное представление имеет несколько очевидных недостатков, а именно:

— величина типа `Int`, используемая в этом случае, может быть неверно истолкована при попытке кем-либо разобраться в программе (например, можно забыть, что `d` представляет величину `Day`, или что 4 означает четверг);
— иногда при использовании `f` можно забыть о предусловии;
— можно по невнимательности «разрушить» договоренность об использовании таких величин (например, умножить величину типа `Day` на 10);
— можно случайно передать такую величину вместо числа типа `Int` (например, если в функции, где `Int` представляет температуру или расстояние).

▶ *Упражнение VII.3.1*

Определите тип-синоним `Day` и величину `d` этого типа:

```
type Day = Int

d :: Day
d  = 1
```

λ

Установив с помощью команды `:set +t` вывод интерпретатором Hugs информации о типах величин, выполните следующие вычисления:

```
---> d
1 :: Day
---> d-10
-9 :: Int
---> d*10
10 :: Int
```

H

Примеры показывают, что величины этого типа ничем не отличаются от целых чисел.

◀

Haskell предоставляет возможность определить тип `Day` более подходящим образом — определить совершенно *новый* тип с помощью декларации `data`:

λ
```
data Day =  Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

В результате такого определения будет создан новый тип данных `Day`. Семь константных функций (постоянных значений) являются *конструкторами данных* типа данных `Day`. Обычно имена конструкторов отличаются тем, что начинаются с заглавной буквы, так же как и имя создаваемого типа данных. Конструкторы служат лишь для создания величин нового типа данных. Они неявно определяются при появлении в выражениях `data`. Конструкторы часто называются *конструирующими функциями*, а конструкторы без аргументов *константами данных*.

▶ *Упражнение VII.3.2*

Пусть нам требуется написать функцию `analyse`, берущую три положительных целых числа `a` `b` и `c`, расположенные в неубывающем порядке. Эти три числа представляют собой длины сторон некоторого треугольника. Функция `analyse` определяет, возможно ли существование треугольника с такими сторонами и если да, то является ли полученный треугольник треугольником общего типа (т. е. с тремя различными сторонами), равнобедренным или равносторонним. Напомним, что три отрезка образуют треугольник, тогда и только тогда, когда длина наибольшей стороны не превосходит сумму двух других (неравенство треугольника).

Таким образом `analyse` должна возвращать одно из четырех возможных значений. Можно было бы использовать четыре различных числа для кодирования этих случаев, но гораздо лучше ввести специальный тип данных, необходимый для этой задачи:

λ
```
data Triangle = Failure       -- фиктивный
              | Isosceles     -- равнобедренный
              | Equilateral   -- равносторонний
              | Scalene       -- общего вида
```

Это определение вводит новый тип данных `Triangle`, имеющий четыре значения. Теперь программа может выглядеть так:

λ
```
analyse  :: (Int, Int, Int) -> Triangle
analyse (a, b, c)
          | a + b <= c          = Failure
          | a == c              = Equilateral
```

```
        | (a == b) || (b == c) = Isosceles
        | otherwise            = Scalene
```

Ветка `otherwise` принимает значение `True`, если все предыдущие условия возвратили значения `False` (отметим, что такое определение функции `alalyse` корректно только при предположении $0 \leqslant a \leqslant b \leqslant c$). Обратите внимание, что операторы `==` и `<=` (меньше или равно) используются здесь для операций сравнения переменных типа `Int`.

```
---> analyse (2,2,2)
Equilateral
---> analyse (2,2,9)
Failure
---> analyse (12,12,19)
Isosceles
---> analyse (3,4,5)
Scalene
```

◀

Одной из мощных концепций объектно-ориентированного программирования является реализация механизма наследования. Тип данных, определенный с помощью декларации `data`, также может наследовать свойства того или иного класса. Для автоматического наследования следует указать после перечисления конструкторов данных ключевое слово `deriving` (дословно: всасывание) и в круглых скобках перечислить имена требуемых классов.

При наследовании методов класса `Eq` величины, задаваемые различными конструкторами данных, не равны между собой. При наследовании свойств класса `Ord` устанавливается лексикографический порядок на величинах данного класса: если один конструктор данных указан в декларации `data` ранее другого, то величина, задаваемая им будет считаться меньшей, чем другая. Любой перечисляемый тип данных, задаваемый с помощью декларации `data`, можно также объявить как экземпляр класса `Enum`, например,

```
data Car = Ford | Jag | Merc | Porsche
   deriving (Show, Read, Ord, Eq, Enum)
```

```
---> Merc == Jag
False
---> Jag > Ford
True
```

```
---> Jag > Merc
False
---> [Jag .. ]
[Jag,Merc,Porsche]
```

Если по каким-либо причинах разработчика не устраивает наследуемый метод, то с помощью декларации `instance` его можно переопределить.

▶ *Упражнение VII.3.3*

Величины типа `Bool`, заданного в прелюдии, могут сравниваться между собой и выводиться на печать. Определим тип `Bool'`, подобный `Bool`, определив его с помощью деклараций `instance` как экземпляр классов `Eq`, `Ord` и `Show`. Для этого следует перегрузить методы этих классов:

λ
```
data Bool' = True' | False'

instance Eq Bool' where
  True'  == True'  = True
  True'  == False' = False
  False' == True'  = False
  False' == False' = True
  x /= y           = not (x == y)

instance Ord Bool' where
  True'  < True'  = False
  True'  < False' = True
  False' < True'  = False
  False' < False' = False
  x <= y          = x < y || x == y
  x > y           = not (x <= y)
  x >= y          = x > y || x == y

instance Show Bool' where
  show True'  = "Истина"
  show False' = "Ложь"
```

Теперь величинами типа `Bool'` можно сравнивать между собой, аналогично стандартным логическим значениям, но их отображение при выводе на печать будет отличаться от стандартного:

H
```
---> True' < False'
```

```
True
---> True' > False'
False
---> True' == False'
False
---> True' /= False'
True
---> putStr (show True')
Истина
```

Отметим, что можно было бы сделать доступными все, определенные на классах `Eq`, `Ord` и `Show` операторы и функции, указав, что данный тип данных наследует свойства перечисленных классов:

```
data Bool' = True' | False'
        deriving (Eq, Ord, Show)
```

λ

◀

3.3. Полиморфные типы данных

Кроме объявления типа при помощи перечисления констант, входящих в него, мы можем объявлять типы, чьи величины зависят от других типов. Например,

```
data Either = Left Bool | Right Char
```

λ

Это объявление типа данных `Either` (от анг. either — один из двух, тот или другой), чьи значения определяются в форме `Left b`, где `b` — логическая величина, и `Right c`, где `c` — символ. Тип `Either` комбинирует логические и символьные величины в один общий тип. Мы можем обобщить эту идею, определив *полиморфный* тип:

```
data Either a b  = Left a | Right b
```

λ

С этим обобщением предыдущий тип можно задать как `Either Bool Char`. Полиморфизм позволяет параметризовать определение типа данных, т. е. вводить в него тип хранимых объектов в качестве параметра. Напомним, что `a` и `b` называются *переменными типа*, являясь некоторыми идентификаторами, обозначающими тип.

Имена `Left` и `Right` задают конструкторы для построения величин типа `Either`, являющиеся нестрогими функциями:

```
Left   :: a -> Either a b
Right  :: b -> Either a b
```

Не следует объявлять тип функции-конструктора, так как он автоматически выводится из декларации типа, использующего данный конструктор.

▶ *Упражнение VII.3.4*

Создайте скрипт, содержащий объявление рассмотренного выше полиморфного типа. Так как тип `Either` уже введен в преамбуле, а «скрыть» определенные там типы невозможно, то используйте имя `Either'` для создаваемого типа данных и имена `Left'` и `Right'` для его конструкторов.

◀

Конструкторы данных лишь создают величину указанного типа, так `Left 3` есть выражение в его простейшей возможной форме — его невозможно упростить. Выражения, включающие конструкторы данных могут появляться в образцах соответствия левосторонней части определения функции. Например,

λ
```
case'         :: (a -> c, b -> c) -> Either a b -> c
case' (f, g) (Left x)  = f x
case' (f, g) (Right y) = g y
```

Функция `case'` получает два аргумента: пару функций и величину типа `Either`. Если эта величина создана при помощи конструктора `Left`, то применяется первая функция из пары, а если при помощи конструктора `Right`, то вторая. Посмотрите на примеры использования этой функции:

H
```
---> case' ((>2), not) (Left 2.3)
True
---> case' ((>2), not) (Right True)
False
```

▶ *Упражнение VII.3.5*

В скрипт с определением типа данных `Either'` добавьте функцию `case'`. Выполните приведенные выше примеры. Параметризуйте тип `Either'` такими значениями, чтобы результатом вызова функции `case'` являлось бы число типа `Float`:

H
```
--> case' ((/2) . fromInt , (*3)) (Left' 2)
1.0
--> case' ((/2) . fromInt , (*3)) (Right' 1.5)
4.5
```

◀

С помощью функции `case'`, определим функцию `plus'`:

```
plus'   :: (a -> b, c -> d) -> Either a b -> Either c d
plus' (f, g) = case' (Left.f, Right.g)
```

λ

Для функций `case'` и `plus'` верны следующие соотношения:

```
case' (f, g) . Left  = f
case' (f, g) . Right = g
h . case' (f, g) = case' (h.f, h.g)
case' (f, g) . plus' (h, k) = case' (f.h. g.k)
```

Предположив, что величины типов `a` и `b` могут быть сравниваемы между собой (т. е. являются экземплярами классов `Eq` и `Ord`), можно определить операторы сравнения для величин типа `Either a b`. Синтаксис подобного рода определений следующий:

```
instance (Eq a , Eq b) => Eq (Either a b) where
   Left  x == Left  y  =  (x == y)
   Left  x == Right y  =  False
   Right x == Left  y  =  False
   Right x == Right y  =  (x == y)
instance (Ord a, Ord b) => Ord (Either a b) where
   Left  x < Left  y   =  (x < y)
   Left  x < Right y   =  True
   Right x < Left  y   =  False
   Right x < Right y   =  (x < y)
```

λ

Указанные определения позволяют расширить область применения операторов `==` и `<` на величины типа `Either`. Этого же эффекта можно добиться, указав с помощью ключевого слова `deriving`, что определяемый тип данных наследует свойства классов `Eq` и `Ord`:

```
data Either a b = Left a | Right b deriving (Eq, Ord)
```

λ

3.4. Объявление рекурсивных типов данных

Итак, мы научились объявлять новые типы данных. Однако подобным образом можно объявлять и *рекурсивные* типы данных.

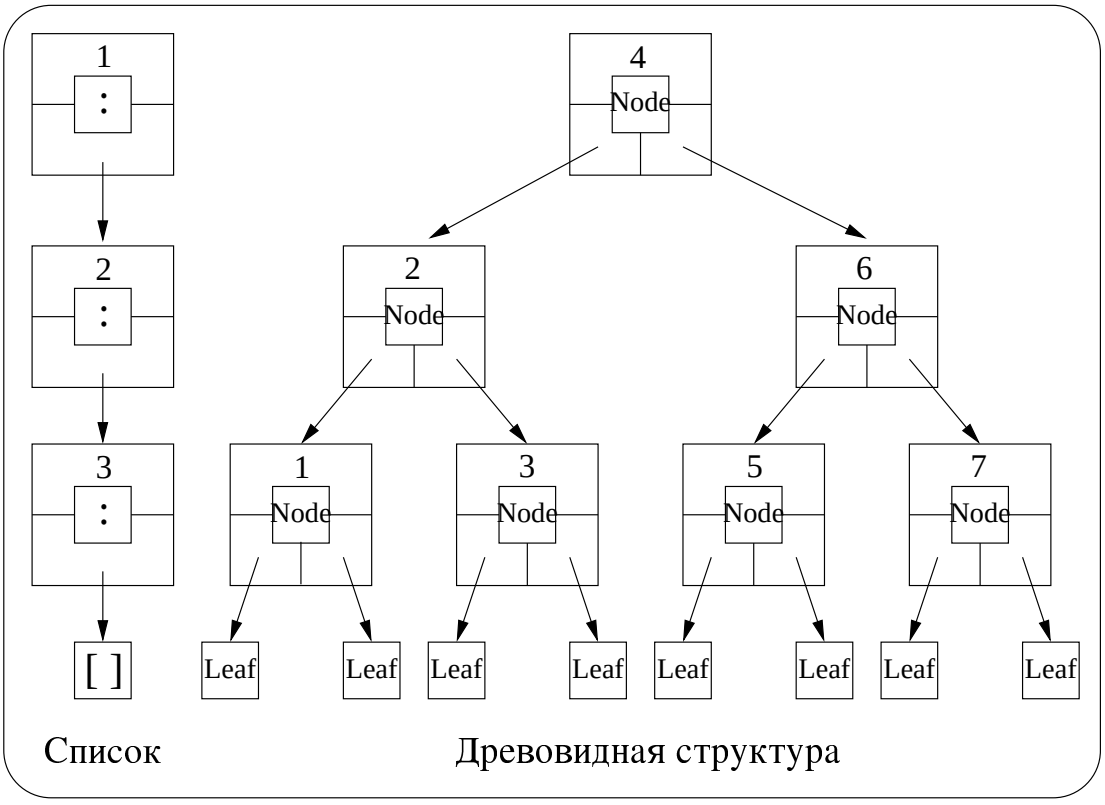
Любой тип данных всегда задает способ создания (конструирования) величин данного типа. Например, список является типом данных и, как нам уже известно, величины типа список могут быть построены двумя способами:

- либо создается пустой список;
- либо с помощью оператора : новый элемент добавляется к уже существующему более короткому списку.

Таким образом, имеется два конструктора для создания списка. Если функция для работы со списками определяется с помощью шаблонов соответствия, то нужно задать два уравнения: по одному на каждый способ создания списка, например,

λ
```
length []     = 0
length (x:xs) = 1 + length xs
```

Задавая функцию при помощи правил для пустого списка и оператора :, примененного к элементу и списку, мы тем самым полностью определяем ее.



Список является линейной структурой: чем больше элементов в него добавляют, тем длиннее он становится. В некоторых случаях такого рода линейная структура неприменима и куда больше подходит *древовидная структура*. Существуют различные виды древовидных структур. На рисунке список сравнивается с древовидной структурой, имеющей ровно по две ветви в каждом внутреннем узле. В маленьких квадратах показано при помощи какого конструктора построена структура. В случае со списком мы

используем оператор : с двумя аргументами (указанными по обе стороны оператора) или `[ ]` без параметров.

В качестве имен древовидных типов данных и их конструкторов часто употребляют слова: Tree — дерево, Node — узел, Fork — развилка, Branch — ветвь, Leaf — лист. При конструировании дерева, изображенного на рисунке, использовались функции (вместо операторов) `Node` (с тремя параметрами) или `Leaf` (без аргументов).

Функции, применяемые для построения структуры данных, называются конструкторами. `Node` и `Leaf` являются конструкторами данных древовидной структуры. Как обычно, имена конструкторов начинаются с заглавной буквы для того, чтобы выделить их из ряда обычных функций.

Имена функций-конструкторов, реализованных в виде операторов, должны начинаться с двоеточия. Конструктор `(:)` является примером конструирующей функции для списка; единственным исключением из этого правила является конструктор `[ ]`. То, какие конструирующие функции можно использовать для нового типа, задается в определении типа данных. В нем же задаются типы параметров конструкторов функций и полиморфичность нового типа. Так, определение приведенной на рисунке древовидной структуры данных задается следующим объявлением типа:

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

λ

Такое определение читается следующим образом:

«Древовидная структура `Tree` с элементами типа `a` может быть построена двумя путями:

(1) с помощью функции `Node` и трех ее параметров (один типа `a` и два типа `Tree`);
(2) с помощью константы `Leaf`».

При задании так определенного древовидного типа структуры используются конструкторы данных `Node` и `Leaf`. Древовидная структура, изображенная на рисунке, описывается следующим выражением:

```
Node  4 ( Node 2 ( Node 1 Leaf Leaf)
                 ( Node 3 Leaf Leaf)
        )
( Node 6 ( Node 5 Leaf Leaf)
                 ( Node 7 Leaf Leaf)
        )
```

λ

Можно не разносить выражения, определяющие дерево, по строкам. Так, допустима следующая форма записи:

```
Node 4 ( Node 2 ( Node 1 Leaf Leaf) ( Node 3 Leaf Leaf))
       ( Node 6 ( Node 5 Leaf Leaf) ( Node 7 Leaf Leaf))
```

Однако первое из приведенных определений представляется более «прозрачным». Не забывайте о двумерном синтаксисе языка Haskell: добавляйте отступы в тех строках, которые являются продолжением определения, начатого ранее.

Функции с аргументами, являющимися древовидной структурой, определяются путем указания шаблонов для каждого конструктора этого типа данных. Следующая функция, например, подсчитывает число элементов `Node` в древовидной структуре:

```
size               ::   Tree  a -> Int
size  Leaf          =  0
size  (Node x p q)  =  1 + size p + size q
```

Сравните эту функцию с `length`, вычисляющей длину списка.

Можно определить огромное количество различных типов древовидных структур. Рассмотрим лишь несколько примеров:

• Деревья, содержащие информацию в листьях, а не в узлах:

```
data Tree2 a  =  Node 2 (Tree2 a) (Tree2 a) | Leaf2  a
```

• Деревья, в которых информация типа `a` размещена в узлах, а информация типа `b` листьях:

```
data Tree3 a b = Node3 (Tree3 a b) (Tree3 a b) | Leaf3 b
```

• Деревья, в которых из каждого узла выходит три ветви:

```
data  Tree4  a  =  Node4 (Tree4 a) (Tree4 a) (Tree4 a)
                 | Leaf4
```

• Деревья, в которых число ветвей, исходящих из узла является переменной величиной:

```
data  Tree5 a  =  Node5 a [Tree5 a]
```

В данном примере дерева нет необходимости отдельно определять листья, поскольку в качестве таковых могут быть использованы узлы, не содержащие исходящих ветвей.

• Деревья, в которых каждый узел имеет только одну ветвь:

```
data Tree6 a  =  Node6 a (Tree6 a) | Leaf6
```

Такой пример дерева по сути является списком, поскольку имеет линейную структуру.

• Деревья, с различными видами узлов:

```
data Tree7 a b = Node7a Int a (Tree7 a b)  (Tree7 a b)
               | Node7b Char (Tree7 a b)
               | Leaf7a b
               | Leaf7b Int
```

Деревья поиска

Хорошим примером ситуации, в которой деревья предпочтительнее, чем списки, есть поиск (присутствующих) элементов в большой коллекции. С этой целью используются *деревья поиска.*

На странице 117 была рассмотрена функция `elem`, которая возвращала `True`, если элемент встречался в списке. Эта функция определялась с помощью функций `map` и `or`:

```
elem    :: Eq a => a -> [a] -> Bool                    λ
elem e xs = or (map (== e) xs)
```

Другое ее определение использует рекурсию:

```
elem e []     = False                                  λ
elem e (x:xs) = x == e || elem e xs
```

В обоих случаях все элементы списка просматриваются один за одним. Как только элемент будет найден, функция немедленно возвратит `True` (благодаря ленивости вычислений), но если элемент не содержится в списке, то функция все равно проверит их все один за другим.

Удобнее было бы работать с отсортированным списком, т. е. таким, все его элементы которого располагаются в неубывающем порядке. Поиск можно прекращать, если ожидаемый элемент уже пройден. Для реализации такого алгоритма требуется, чтобы элементы списка допускали не только проверку на равенство, но и сравнение (т. е. являлись бы величинами класса `Ord`):

```
elem'   :: Ord a => a -> [a] -> Bool                   λ
elem' e []               = False
elem' e (x:xs) | e < x   = False
               | e == x  = True
               | e > x   = elem' e xs
```

Значительным улучшением программы является использование не списка, а *дерева поиска*. Дерево поиска есть род отсортированного дерева. Это дерево может быть определено с помощью объявления, рассмотренного нами ранее:

λ
```
data  Tree a = Node a (Tree a) (Tree a) | Leaf
```

В таком дереве в каждом узле содержится элемент и два (меньших) поддерева: левое и правое (см. рисунок на стр. 186). Основным требованием, предъявляемым к дереву поиска, является расположение в левом поддереве величин меньших, чем в узле, а в правом — больших. Значения в узлах дерева, приведенного на рисунке, выбраны таким образом, что оно является деревом поиска.

Поиск элементов в дереве поиска очень прост. Если величина равна значению, хранящемуся в узле, то элемент обнаружен. Если он меньше, то поиск продолжается в левом поддереве (правое поддерево содержит элементы с бóльшим значением). С другой стороны, если величина больше, чем хранящаяся в узле, то поиск продолжается в правом поддереве. Поэтому функция `elemTree` выглядит так:

λ
```
elemTree   :: Ord a => a -> Tree a -> Bool
elemTree e Leaf                    = False
elemTree e (Node x l r) | e == x = True
                        | e < x  = elemTree e l
                        | e > x  = elemTree e r
```

Если дерево хорошо сбалансировано, т. е. не содержит больших «дыр», то число элементов, предназначенных для поиска, уменьшается приблизительно в два раза на каждом шаге. Поэтому требуемый элемент будет найден достаточно быстро: так в коллекции из 1000 элементов потребуется сделать около десяти шагов, а в коллекции из миллиона элементов около двадцати. В общем, можно сказать, что поиск в коллекции из `n` элементов при помощи функции `elem` потребует около `n` шагов, а при помощи функции `elemTree` только $\log_2$ `n` шагов.

Структура дерева поиска

Форма дерева поиска для некоторой коллекции элементов может быть определена «вручную». Затем дерево поиска можно с помощью клавиатуры ввести как одно большое выражение с множеством конструирующих функций. Тем не менее может возникнуть потребность в автоматизации этого процесса.

Аналогично функции `insert`, добавляющей элементы в отсортированный список (стр. 118), функция `insertTree` добавляет элемент к дереву поиска таким образом, что результат добавления остается деревом поиска, т. е. вставка элемента производится в нужное место:

```
insertTree  :: Ord a => a -> Tree a -> Tree a
insertTree e Leaf          = Node e Leaf Leaf
insertTree e (Node x l r)
               | e <= x  = Node x (insertTree e l) r
               | e > x   = Node x l (insertTree e r)
```

λ

Если элемент добавляется к листу (т. е. к пустому дереву), то строится небольшое дерево, состоящее из одного узла, в котором хранится элемент `e`, и двух пустых деревьев. В противном случае дерево не пусто и оно содержит в корневом узле элемент `x`. Это значение и используется для определения, в какое поддерево помещать новый элемент.

С помощью функции `insertTree` все элементы списка можно поместить в дерево поиска:

```
listToTree   :: Ord a => [a] -> Tree a
listToTree = foldr insertTree Leaf
```

λ

Сравните полученную функцию с функцией `isort` (стр. 119).

Недостатком использования функции `listToTree` является то, что в результате ее применения не всегда получается хорошо сбалансированное дерево. Эта проблема не стоит так остро, когда информация добавляется в случайном порядке. Если же список, который преобразуют в дерево уже отсортирован, то полученное дерево поиска будет «перекошенным».

```
---> listToTree [1 ..7]
Node 7 (Node 6 (Node 5 (Node 4 (Node 3 (Node 2
(Node 1 Leaf Leaf) Leaf) Leaf) Leaf) Leaf) Leaf) Leaf
```

Н

Хотя данная структура и является деревом поиска (величина каждого элемента заключена между значениями левого и правого поддерева) структура его по сути линейна. В таком дереве недостижимо логарифмическое время поиска. Значительно лучшим (не «линейным») деревом поиска с теми же значениями будет дерево

```
Node 4 (Node 2 (Node 1 Leaf Leaf)
               (Node 3 Leaf Leaf))
       (Node 6 (Node 5 Leaf Leaf)
               (Node 7 Leaf Leaf))
```

Сортировка с помощью дерева поиска

Функции, рассмотренные выше, могут служить основой для построения нового алгоритма сортировки. Нам потребуется лишь еще одна функция, которая помещает элементы дерева поиска в список в нужном порядке. Определение ее таково:

```
labels   :: tree a -> [a]
labels leaf = []
labels (Node x l r) = labels l ++ [x] ++ labels r
```

В отличии от функции `insertTree`, эта функция производит рекурсивную обработку *как левого, так и правого* поддерева. Это значит, что каждый элемент дерева будет обработан.

Произвольный список следует сначала превратить в дерево поиска с помощью функции `insertTree`, а затем собрать их в список в правильном порядке с помощью `labels`:


```
sort :: Ord a => [a] -> [a]
sort = labels . listToTree
```

Удаление из дерева поиска

Деревья поиска можно использовать в роли базы данных. Кроме операций перенумеровки, вставки и построения, которые уже описаны, нам потребуется еще функция, удаляющая элементы из базы данных. Она похожа на функцию `insertTree`: в зависимости от своего аргумента она вызывается рекурсивно либо для левого, либо для правого поддерева.

```
deleteTree              :: Ord a => a -> Tree a -> Tree a
deleteTree e Leaf        = Leaf
deleteTree e (Node x l r)
               | e < x  = Node x (deleteTree e l) r
               | e == x = join l r
               | e > x  = Node x l (deleteTree e r)
```

Если величина будет найдена в дереве (случай `x == e`), то она не может быть просто опущена, так как в этом месте осталась бы «дыра». Поэтому необходима функция `join`, соединяющая два поддерева в одно.

Эта функция берет наибольший элемент левого поддерева в качестве нового узла. Если левое поддерево пусто, то в слиянии поддеревьев уже нет проблемы.

```
join          :: tree a -> Tree a -> Tree a
```

```
join Leaf b2 = b2
join b1 b2   = Node x b1' b2
               where (x, b1') = largest b1
```

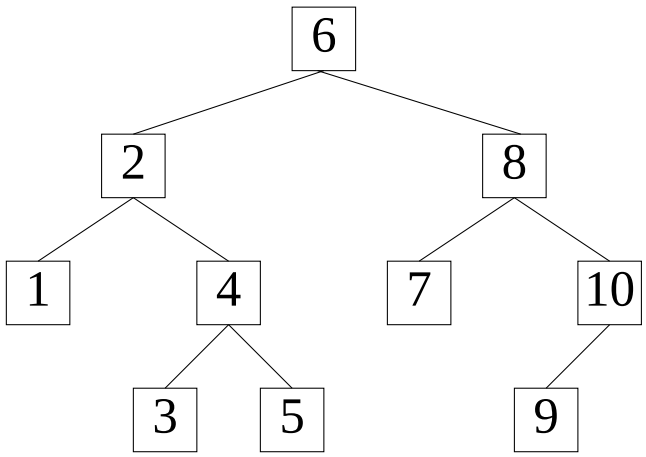
Функция `largest` кроме наибольшего элемента поддерева также возвращает дерево, остающееся после изъятия наибольшего элемента. Оба результата соединяются в пару. Наибольший элемент находится при рекурсивном просмотре правых поддеревьев:

```
largest                  :: Tree a -> (a, Tree a)
largest (Node x b1 Leaf) = (x, b1)
largest (Node x b1 b2)   = (y, Node x b1 b2')
               where (y, b2') = largest b2
```

λ

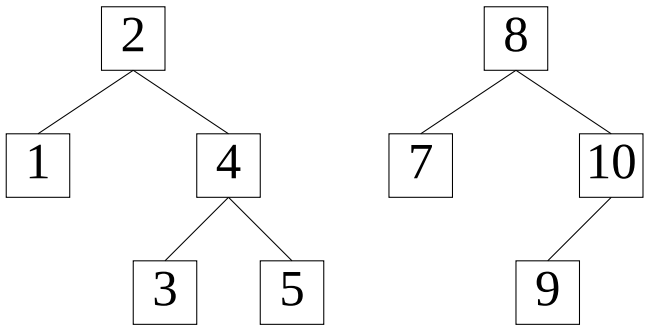
Проиллюстрируем работу функции `deleteTree` следующим примером.

`deleteTree 6`

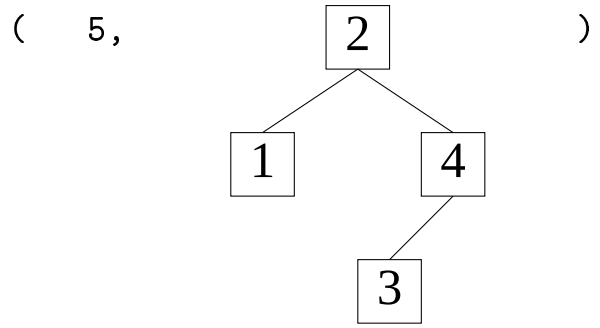


Этот вызов приведет к применению функции `join` с левому и правому поддеревьям в качестве параметров:

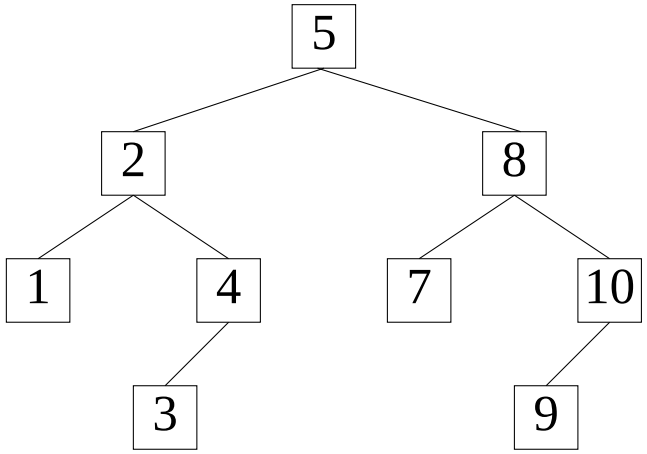
`join` `b1` `b2`



Функция `largest` будет вызвана функцией `join` в поддеревом `b1` в качестве параметра. В результате будет получена пара `(x, b1')`:



Деревья `b1'` и `b2` выступят в качестве поддеревьев для нового дерева поиска:



Функция `largest` вызывается функцией `join` только в том случае, если полученное дерево не пусто (т. е. не является `Leaf`-деревом). Для обработки пустого дерева в определение функции `join` включено отдельное правило.

▶ *Упражнение VII.3.6*

Подготовьте скрипт, содержащий вышеописанные функции для работы с деревьями поиска. При объявлении типа `Tree` не забудьте указать, что он является экземпляром класса `Show` (без подобного указания вы не сумеете увидеть результат применения функций к деревьям):

λ
```
data  Tree a = Node a (Tree a) (Tree a) | Leaf
        deriving (Show)
```

Включите в скрипт описание дерева поиска из рассмотренного примера, задаваемое выражением:

λ
```
tree1 :: Tree Int
tree1 = Node 6 (Node 2 (Node 1 Leaf Leaf)
                       (Node 4 (Node 3 Leaf Leaf)
                               (Node 5 Leaf Leaf)))
```

```
(Node 8 (Node 7 Leaf Leaf)
        (Node 10 (Node 9 Leaf Leaf)
                 Leaf))
```

Теперь можно работать с величиной `tree1`, как с базой данных. Выполните следующие вызовы функций:

```
---> elemTree 4 tree1
True
---> elemTree 14 tree1
False
---> elemTree 14 (insertTree 14 tree1)
True
---> elemTree 4 (deleteTree 4 tree1)
False
```

◀

3.5. Метки полей

Поля в пределах декларации `data` могут быть доступны либо позиционно, либо по имени с использованием *меток полей*. Рассмотрим тип данных, представляющих точку на плоскости:

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

Две компоненты `Point` есть первый и второй аргумент конструктора `Pt`. Функция, аналогичная,

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

может использоваться для ссылки на первую компоненту точки в достаточно наглядном виде, но для больших структур довольно утомительно создавать такие функции вручную.

Конструктор в декларации типа `data` может быть объявлен с ассоциированным *именем поля*, заключенным в скобки. Такое имя поля отождествляет компоненты конструктора с именем яснее, чем указание позиции поля в структуре. Вот альтернативный способ определения типа `Point`:

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

Такое определение идентично приведенному ранее определению типа `Point`. Конструктор `Pt` один и тот же в обоих случаях. Однако, эта декларация задала имена двум полям: `pointx` и

pointy. Данные имена могут использоваться в качестве *селекторных* функций, позволяющих выделять компоненты из структуры. В этом примере тип селекторных функций таков:

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

А вот функция, использующая эти селекторы:

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

Метки полей можно использовать и для получения новых величин: выражение `Pt {pointx = 1, pointy = 2}` идентично `Pt 1 2`. Использование имен полей не вытесняет употребление позиционного стиля при задании величин, так обе формы, приведенные ранее, доступны пользователю, однако, *с помощью меток полей можно задать величину, не все поля которой будут определены*, что невозможно при позиционной форме определения величины.

Образцы соответствия, включающие имена полей, используют такой же синтаксис, что и конструктор `Pt`:

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

Функция, изменяющая значение структуры (update function), использует значения поля в существующей структуре для заполнения компонент новой структуры. Если `p` есть величина типа `Point` (т. е. точка), то `p {pointx = 2}` есть точка с тем же значением поля `pointy`, что и у `p`, но с `pointx`, замененным на `2`. Такая операция не является деструктивной: обновляющая функция лишь создает новую копию объекта, заполняя некоторые его поля новыми значениями.

▶ *Упражнение VII.3.7*

Создайте скрипт, содержащий следующий код:

λ
```
data Point = Pt {px, py :: Float} deriving Show

absPoint :: Point -> Float
absPoint p = sqrt (px p * px p + py p * py p)

e1 :: Point
e1 = Pt {px = 1, py = 2}
e2 :: Float
e2 = absPoint e1
```

```
e3 :: Float
e3 = px e1
e4 :: Point
e4 = e1 {px = 4}
```

Загрузите скрипт и выполните вычисления:


```
---> e1
Pt{px=1.0,py=2.0}
---> e2
2.23607
---> e3
1.0
---> e4
Pt{px=4.0,py=2.0}
```

Вычислите расстояние от точки `e4` до начала координат. Определите функцию, вычисляющую расстояние между двумя точками.

◀

Скобки, используемые при задании меток полей, имеют специальное значение: несмотря на то, что синтаксис языка Haskell обычно разрешает опускать скобки, заменяя их соответствующим выравниванием текста (*layout rule*), при использовании меток полей скобки *необходимы*.

Метки полей могут использоваться не только в типах данных с единственным конструктором (обычно называемые записями, records). Однако, следует отметить, что в типах с несколькими конструкторами применение выделяющих или обновляющих функций, использующих имена полей, может иногда привести к ошибке времени выполнения, аналогичной той, к которой приводит вызов функции `head` с пустым списком в качестве аргумента.

То или иное имя поля не может использоваться более, чем в одном типе данных. Тем не менее в пределах одного типа данных, это имя может использоваться в различных конструкторах. Так, в следующем типе данных метка `f` используется сразу в двух конструкторах:


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

Если `x` есть величина типа `T`, то выражение `x {f = 5}` допустимо использовать независимо от того, каким из конструкторов данных типа `T` была создана величина `x`.

▶ *Упражнение VII.3.8*

Создайте скрипт, содержащий следующий код:

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

e5 :: T
e5 = C1 {f = 1, g = 2}
e6 :: T
e6 = C2 {f = 3, h = False}
e7 :: Int
e7 = f e5
e8 :: Int
e8 = f e6
e9 :: Float
e9 = g e6
```

Выполните вычисления и объясните полученные результаты:

H
```
---> e5
C1{f=1,g=2.0}
---> e6
C2{f=3,h=False}
---> e7
1
---> e8
3
---> e9

Program error: {T_C1_g e6}
```

◀

Метки полей не меняют природу алгебраического типа данных, а лишь являются удобной синтаксической возможностью обращения к компонентам структуры данных. Они делают конструкторы с несколькими компонентами более удобными для практического использования, так как поля могут добавляться или удаляться без упоминания конструктора.

3.6. Строгий конструктор в data декларации

Структуры данных в языке Haskell в общем случае являются ленивыми: компоненты не вычисляются до тех пор, пока явно не

потребуются. Это дает возможность применения структур, содержащих элементы, вычисление которых, если оно потребуется, приведет к ошибке или прекращению выполнения. Использование ленивых структур данных усиливает выразительность языка Haskell и является неотъемлемой чертой стиля программирования на данном языке.

Внутреннее представление каждого поля объекта, задаваемого ленивой структурой данных напоминает некоторую большую и тяжелую упаковку, которая скрывает вычисление величины, до того момента, когда она потребуется. Упаковки, содержащие неопределенные элементы ($\perp$), не затрагивают другие элементы структуры данных. Так, кортеж `('a', ⊥)` является совершенно легальной величиной в Haskell. Попробуйте:

```
---> fst ('a', undefined)
'a'
```


Использование величины `'a'` не затрагивает другие компоненты тьюпла.

Большинство языков программирования являются *строгими*, а не ленивыми: в них все компоненты структур данных вычисляются перед тем, как они помещаются в структуру.

Есть еще несколько особенностей, связанных с подобной упаковкой структур: все такие структуры требуют дополнительного времени для конструирования и вычисления; для их размещения в памяти используется куча (heap) и они позволяют применять сборку мусора, во время вычисления. Избежать таких накладных расходов позволяют, так называемые, *флаги строгости*, употребление которых в `data`-декларации приводит к немедленному вычислению отмеченных им полей конструктора, игнорируя ленивость.

Флаги строгости можно использовать в следующих ситуациях:

- для задания тех компонент структуры, которые обязательно будут вычисляться во время выполнения программы;
- для задания компонент структуры, которые легко вычисляются и никогда не приводят к ошибкам;
- при использовании типов, для которых употребление частично неопределенных величин не является необходимым.

Комплексные числа, например, можно объявить как тип `Complex` следующим образом:

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

В данном определении используется конструктор в форме инфиксного оператора `:+`. Это определение выделяет две компоненты — действительную и мнимую части комплексного числа — как строго определенные. Можно предложить более компактное представление комплексных чисел без использование флага строгости, но оно приводит к дополнительным расходам при создании комплексного числа с неопределенной компонентой, такого как `1 :+` $\perp$.

Поля, отмеченные знаком `!` *в декларации* `data`*, вычисляются в момент создания.*

Флаги строгости могут приводить к утечкам адресов памяти: такие структуры не позволяют использовать сборку мусора для тех частей, которые уже не потребуются для вычислений.

Флаг строгости `!` может использоваться только в декларациях `data`. Он не может употребляться в сигнатурах определений других типов деклараций. Следует предостеречь от бездумного использования флага строгости, так как ленивость является одним из фундаментальных свойств языка Haskell и употребление флагов строгости может значительно усложнить обработку бесконечных структур или привести к другим неожиданных последствиям.

▶ *Упражнение VII.3.9*

Определим тип данных, аналогичный спискам, но строгий по отношению к его первому элементу (голове):

λ
```
data HList a = Cons !a (HList a) | Nil deriving Show

hd (Cons x y) = x
tl (Cons x y) = y

e10 :: HList Bool
e10 = True `Cons` (error "e10" `Cons` Nil)
e11, e12 :: Bool
e11 = hd e10
e12 = hd (tl e10)
e13 :: HList Bool
e13 = tl (tl (e10))
```

Вычислите указанные значения и прокомментируйте полученные результаты:

H
```
---> e10
Cons True
Program error: e10
```

```
---> e11
True
---> e12
Program error: e10
---> e13
Program error: e10
```

Объясните, почему при удалении флага строгости (!) в объявлении типа исчезает ошибка при вычислении `e13`:

```
---> e13
Nil
```


3.7. Новые типы: декларация `newtype`

Общей программистской практикой является определение типа, который представляет в точности существующий тип, но представляется отдельно в системе типов. В Haskell декларация **`newtype`** создает новый тип из уже существующего. За исключением ключевого слова, **`newtype`**-декларация использует тот же самый синтаксис, что и **`data`** декларация с единственным конструктором, содержащим единственное поле.

Например, натуральные числа могут быть представлены типом `Integer` с помощью следующей декларации:

```
newtype Natural = MakeNatural Integer
      deriving (Eq, Ord, Show)
```

Такое объявление создает совершенно новый тип `Natural`, фактически переименовывающий `Integer`. Полученный таким образом тип отличается от типа-синонима тем, что это совершенно отдельный тип, который полностью соответствует исходному типу. Подобное преобразование не требует дополнительных временных затрат; **`newtype`** не изменяет лежащее в основе представление объекта. Допустимо определение типа, полученного с помощью декларации **`newtype`**, как экземпляра того или иного класса, что невозможно для типов, объявленных с помощью декларации **`type`**, т. е. синонимов.

В приведенном выше объявлении, конструктор `MakeNatural` преобразует величину типа `Integer` к типу `Natural`:

```
toNatural                  :: Integer -> Natural
toNatural x | x < 0         = error "Величина меньше 0!"
```

```
            | otherwise     = MakeNatural x

fromNatural                 :: Natural -> Integer
fromNatural (MakeNatural i) = i
```

Следующая декларация `instance` определяет `Natural` как экземпляр класса `Num`:

λ
```
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 "Разность < 0"
                             else toNatural r
  x * y       = toNatural (fromNatural x * fromNatural y)
```

Без подобной декларации `Natural` не может использовать методы класса `Num`. Декларация наследования для типа, используемого в качестве основы, не переносится автоматически на тип, определенной при помощи объявления `newtype`. Несмотря на то, что тип `Integer` уже является экземпляром класса `Num`, для типа `Natural` необходимо вновь переопределить методы родительского класса.

▶ *Упражнение VII.3.10*

Создайте скрипт, в который поместите объявление типа `Natural` и перегруженные методы класса `Num`, описанные выше. После этого создайте несколько натуральных чисел и выполните соответствующие арифметические операции:

H
```
---> toNatural 5
MakeNatural 5
---> MakeNatural 3 + MakeNatural 4
MakeNatural 7
---> MakeNatural 3 - MakeNatural 4
MakeNatural
Program error: Разность отрицательна
---> MakeNatural 3 * MakeNatural 4
MakeNatural 12
```

◀

Еще раз отметим, что так определенный тип не совпадает с `Num`, в отличии от использования типа-синонима для `Integer`, а является совершенно отдельным типом данных.

Подобного эффекта можно было бы добиться и с помощью декларации `data` вместо `newtype`. Тем не менее, манипуляция с величинами типа, объявленного с помощью декларации `data`, как уже отмечалось, приводит к повышенным тратам времени при работе с величинами такого типа. Применение декларации `newtype` позволяет избежать этих дополнительных расходов (связанных с ленивостью вычислений) неизбежно привносимых декларацией `data`. Другое отличие состоит в различных правилах применения образцов соответствия: *конструктор в декларации переименования (*`newtype`*) является строгой функцией.* Это означает, что если тип `N` объявлен с помощью декларации `newtype`, а $\perp$ — есть неопределенная величина, то `N` $\perp$ есть в точности $\perp$. В нашу задачу сейчас не входит описание всех тонкостей в различиях этих деклараций, рассмотрим лишь один пример, вносящий некоторую ясность в употребление той или иной декларации типа.

▶ *Упражнение VII.3.11*

Поместите в скрипт определения нескольких типов данных:

```
data D1   = D1 Int
data D2   = D2 !Int
type S    = Int
newtype N = N Int

d1 (D1 i) = 42
d2 (D2 i) = 42
s i       = 42
n (N i)   = 42
```

При таких определениях типов и функций, выражения `(d1` $\perp$`)`, `(d2` $\perp$`)`, `(d2 (D2` $\perp$`))` эквивалентны $\perp$, в то время как `(n` $\perp$`)`, `(n (N` $\perp$`))`, `(d1 (D1` $\perp$`))` и `(s` $\perp$`)` эквивалентны 42. В частности `(N` $\perp$`)` эквивалентно $\perp$, а `(D1` $\perp$`)` не эквивалентно $\perp$. Напомним, что неопределенное значение $\perp$ в языке Haskell задается с помощью функции `error` или `undefined`.

Выполните несколько вызовов определенных выше функций с различными аргументами, после чего объясните особенности их поведения:

```
---> d1 (undefined)

Program error: {undefined}
---> d1 (D1 undefined)
```

```
42
---> d2 (undefined)

Program error: {undefined}
---> d2 (D2 undefined)

Program error: {undefined}
---> s (undefined)
42
---> n (undefined)
42
---> n (N undefined)
42
```

◀

Вопросы и задания

VII.3.1

Предположим, мы хотим считать два расстояния равными, если они отличаются менее чем на 10 км. Можно ли определить тест на равенство для `Distance`, если `Distance` есть тип-синоним? Если да, то что можно назвать операцией `(==)`?

VII.3.2

Перепишите определение функции `alalyse`, определенной на стр. 180 так, чтобы случаи анализа не зависели от порядка, в котором они даются.

VII.3.3

Определите функцию `group`, объявленную как

λ
```
group   :: Int -> [a] ->[[a]]
```

Эта функция разделяет данный список на подсписки (помещая их в результирующий список), где все подсписки имеют заданную длину. Только последний подсписок может иметь длину меньшую заданной. Например,

H
```
---> group [3 [1 .. 11]
[ [1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11]]
```

VII.3.4

Предположим, что дата задается тройкой `(d, m, y)`, где `d`, `m` и `y` числа типа `Int`, задающие соответственно день, месяц и год. Дайте определение функции `showDate`, берущей дату и печатающую ее в виде аналогичном приведенному:

```
---> putStr(showDate (10, 11, 2002))
10 ноября 2002 года
```

VII.3.5

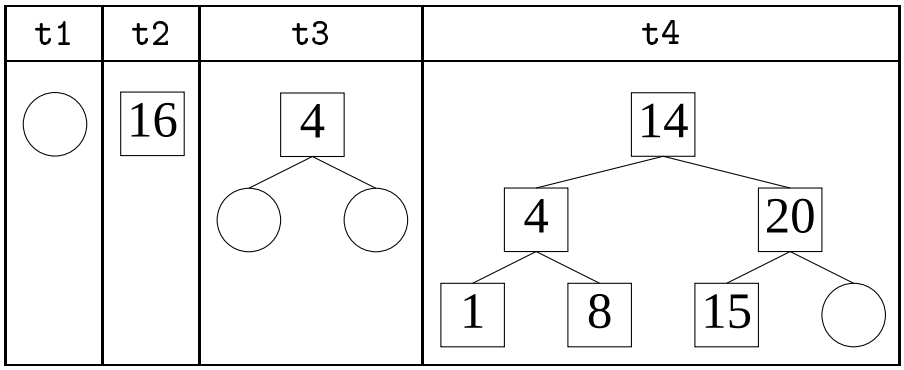
Напишите четыре арифметических функции для работы с комплексными числами. Комплексные числа — это числа, представимые в виде $a + bi$, где a и b есть действительные числа, а i — мнимая единица, удовлетворяющая уравнению $i^2 = -1$. Совет: найдите формулу для вычисления $1/(a + bi)$, решая уравнение $(a + bi) * (x + iy) = (1 + 0i)$ относительно x и y.

VII.3.6

Пусть дерево задается следующим объявлением типа:

```
data Tree'  = Empty | Leaf Int | Node Tree' Int Tree'
     deriving (Show)
```

В таблице приведено несколько примеров деревьев данного типа.



Здесь `t1` — пустое дерево, `t2` — дерево с единственным элементом (листом), имеющим тип `Int`, `t3` и `t4` — примеры многоуровневых деревьев, построенных из внутренних вершин, вершин-листьев и пустых деревьев. Каждая внутренняя вершина содержит число и два поддерева. Первые три дерева заданы определениями:

```
t1, t2, t3 :: Tree'
t1 = Empty
t2 = Leaf 16
t3 = Node Empty 4 Empty
```

Приведите выражение, описывающее дерево `t4`.

VII.3.7

Для дерева поиска из предыдущего задания напишите программу, содержащую функции вставки и удаления элементов в дерево.

VII.3.8

Перепишите функцию сортировки с помощью дерева таким образом, чтобы она использовала дерево типа `Tree'`.

VII.3.9

Дано дерево следующего типа:

```
data Tree'' a = Leaf a
              | Node'' (Tree'' a) (Tree'' a)
```

Напишите функции `mapTree` и `foldTree`, определенные на типе `Tree''`, и аналогичные функциям `map` и `fold`, заданным на списках. Обязательно укажите тип каждой функции.

VII.3.10

Рассмотрим объявление

λ
```
data  Jane   = MkJane Int
newtype Dick = MkDick Int
```

Дайте подходящие определения функций, демонстрирующих, что типы `Jane` и `Dick` отличаются друг от друга.

4. Примеры структурированных типов данных

4.1. Тип `Angle`

Ранее (стр. 156) мы ввели тип-синоним `Angle` для обозначения угла поворота на плоскости. Так как синоним не создает новых типов, он унаследует все свойства основного типа. Поэтому смысл оператора `==` для величин типа `Angle` тот же, что и для величин типа `Float`. На практике может потребоваться вложить другой смысл в тот или иной оператор или функцию, определенную на новом типе. Например, естественно объявить равенство на множестве углов на плоскости как равенство по модулю числа 2π (два угла равны между собой, если их разность кратна 2π). Конечно,

можно было бы ввести специальную функцию для проверки на равенство двух углов, учитывающую отмеченное свойство углов, но если нам хочется использовать именно оператор ==, то мы должны объявить Angle как новый тип данных, а не как тип-синоним:

```
data Angle = MkAngle Float
```

λ

```
instance Eq Angle where
   MkAngle x == MkAngle y = normalise x == normalise y
      normalise ::  Float -> Float
      normalise x
         | x < 0 = normalise (x +rot)
         | x >= rot = normalise (x-rot)
         | otherwise = x
            where rot = 2*pi
```

Вспомогательная функция normalise, используемая в этом определении, возвращает угол в диапазоне от 0 до 2π.

Платой за применение такого определения равенства углов, является то, что элементы Angle теперь имеют «суперобложку» в лице конструктора данных MkAngle.

```
---> MkAngle (-pi) ==  MkAngle pi
True
---> MkAngle 0 ==  MkAngle (2*pi)
True
---> MkAngle pi ==  MkAngle (2*pi)
False
```

H

У этого способа определения типа Angle есть два недостатка. Первым из них является то, что операции создания и использования углов, вследствие ленивости конструктора типов data, требуют постоянного «заворачивания» и «перезаворачивания» данных. Все это требует приводит к дополнительным затратам памяти и времени при работе с подобным образом определенными величинами. Во вторых, объявляя новый тип данных, мы также объявляем возможно нежелательные элементы, например, MkAngle $\perp$ также является величиной типа Angle. Другими словами, типы Angle и Float уже *не изоморфны*.

Как уже отмечалось, осознавая это, Haskell обеспечивает альтернативный механизм создания нового типа, полностью изоморфного уже существующему. Объявление

```
newtype Angle =  MkAngle Float
```

λ

вводит `Angle` как новый тип, чей образ *точно соответствует* типу `Float`. Отличие такого объявления от типа-синонима в том, что создается новый тип, и что можно объявлять выведенные из него классы. Отличие от `data`-декларации состоит, во первых, в том, что манипуляции с величинами, определенными с помощью такого конструктора данных `MkAngle`, не требует дополнительного времени. Хотя выражение `MkAngle` и присутствует в тексте программы, оно удаляется еще до начала вычислений: величины такого типа методично заменяются соответствующими значениями базового типа. Второе отличие состоит в том, что `MkAngle` — *строгий* конструктор и `MkAngle` $\perp$ и $\perp$ объявляют одно и то же значение. Следовательно, между величинами типов `Angle` и `Float` имеется взаимно однозначное соответствие.

4.2. Тип `Day`

Вернемся к рассмотрению типа данных `Day`, введенного на стр. 180 следующим определением:

λ
```
data Day =  Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

Возможно в программе потребуется сравнение элементов типа `Day`, поэтому следует объявить, что `Day` наследует свойства классов `Eq` и `Ord`. Если мы решим переопределять методы этих классов с помощью образцов соответствия, то придется задать очень большое число уравнений. Лучшей идеей будет кодирование элементов типа `Day` целыми числами и использование сравнения целых чисел. Так как подобная идея может быть применима и к другим перечисляемым типам, то заметим, что именно для таких ситуаций и введен класс `Enum` (см. стр. 175), который описывает типы, чьи элементы могут быть перечислены.

Тип, объявленный как экземпляр (instance) класса `Enum`, использует функции `toEnum` и `fromEnum` для преобразования между его элементами и типом `Int`. Функция `fromEnum` должна быть левообратимой (left-inverse) по отношению к `toEnum`, т. е. для всех `x` должно выполняться соотношение `fromEnum (toEnum x) = x`.

Определим `Day` как экземпляр класса `Enum`:

λ
```
instance Enum Day where
  toEnum Sun = 0
  toEnum Mon = 1
  toEnum Tue = 2
  toEnum Wed = 3
```

```
  toEnum Thu = 4
  toEnum Fri = 5
  toEnum Sat = 6
```

Функция `fromEnum` также задается семью уравнениями.

Подобная идея ассоциирования значений перечисляемого типа с целыми используется при описании типа `Char`. Фактически, можно объявить

```
instance Enum Char where
   toEnum   = ord
   fromEnum = chr
```

Имея функцию `toEnum` для `Day`, мы можем определить методы других классов:

```
instance Eq  Day where                                          λ
   x == y = toEnum x == toEnum y
instance Ord Day where
   x < y = toEnum x < toEnum y
```

Напомним, что Haskell позволяет использовать более удобную форму объявления типа наследующего свойства того или иного класса: вместо декларации `instance`, в которой перегружаются методы некоторого класса, можно объявить тип, указав при этом, что он *наследует* (deriving — наследование) свойства указанного класса. Так можно записать

```
data Day =  Sun | Mon | Tue | Wed | Thu | Fri | Sat              λ
      deriving (Eq, Ord, Enum, Show)
```

Пункт **deriving** служит для генерации подкласса перечисленных классов автоматически, но эти классы должны быть описаны ранее. Автоматическое наследование подкласса для `Enum` возможно только для типов данных, которые объявляются явным списком их величин, причем порядок следования их конструкторов в объявлении задает отношение порядка на элементах этого типа.

Теперь в функциях для работы с днями недели можно использовать тип `Day`:

```
workday :: Day -> Bool                                          λ
workday d = (Mon <= d) && (d <=  Fri)
restday :: Day -> Bool
restday d = (d == Sat) || (d == Sun)
```

Для определения функции `dayAfter`, выдающую день, следующий за текущим, используем функции `toEnum` и `fromEnum`:

λ
```
dayAfter  :: Day -> Day
dayAfter d = fromEnum (toEnum d + 1) `mod` 7
```

В частности, `dayAfter Sat = Sun`.

▶ *Упражнение VII.4.1*

Определите тип `Day`, используя автоматическую генерацию подклассов, и функции для работы с ним:

λ
```
data Day =  Sun | Mon | Tue | Wed | Thu | Fri | Sat
      deriving (Eq, Ord, Enum, Show)

workday :: Day -> Bool
workday d = (Mon <= d) && (d <=  Fri)

restday :: Day -> Bool
restday d = (d == Sat) || (d == Sun)

dayAfter  :: Day -> Day
dayAfter d = toEnum ((fromEnum d + 1) `mod` 7)
```

Вызовите определенные выше функции и проверьте, как выполняются на элементах данного типа отношения равенства и порядка:

H
```
---> dayAfter Sat
Sun
---> restday Sat
True
---> workday Sat
False
---> Sat < Mon
False
---> Wed == Wed
True
---> Wed == Fri
False
```

С другой стороны, унаследованное определение может быть не совсем таким, как нам хочется. Например, может возникнуть требование о специфическом виде печати тех или иных данных типа, выведенного из класса `Show`, т. е. потребуется другое видимое представление элементов, отличное от того, что следует из

наследуемого определения. В подобных ситуациях придется дать явно объявление типа как экземпляра того или иного класса — `instance`-декларацию. И, наконец, если потребуется разработать особый способ печати элементов какого-то типа данных, то всегда можно написать специальную функцию для этого случая. Мы не обязаны использовать функцию `show`. Проиллюстрируем эту возможность примером. Пусть дан тип

```
data  Data =  Pair Bool Int  deriving (Show)
```

λ

Если нужно напечатать величины типа `Data`, то наследуемое из `Show` определение дает следующее

```
---> Pair True (3+4)
Pair True 7
```

Н

Но ничто не мешает ввести функцию `showData`, предназначенную для печати величин типа `Data`:

```
showData            :: Data -> String
showData (Pair b n) =  if b then "+" ++ show n
                           else "-" ++ show n
putData = putStr . showData
```

λ

Например,

```
---> putData (Pair True (3+4))
+7
---> putData (Pair False  (3*4))
-12
```

Н

В определении `showData` функция `show` используется для `Int`, но не для `Data`.

4.3. Вектора и матрицы

В этом разделе мы напомним некоторые понятия векторной алгебры и увидим как можно смоделировать операции с матрицами с помощью типов языка Haskell.

Обобщением одномерной линии, двумерной плоскости и трехмерного пространства является n-мерное пространство, где каждая точка задается с помощью n координат. Аналогично определяется n-мерный вектор. Для задания подобных объектов можно использовать тип-синоним `Vector`:

```
type Vector = [Float]
```

λ

Количество элементов в таком списке определяет размерность пространства. Для того чтобы отличать подобный объект от других списков, содержащих числа типа `Float`, лучше определить новый тип данных при помощи декларации `data`:

λ
```
data  Vector = Vec [Float]
```

Здесь конструктор данных `Vec` преобразует список чисел в величину типа `Vector`. Аналогично, точку в n-мерном пространстве можно определить как вектор, соединяющий ее с началом координат — ведь все координаты точки и такого вектора равны. Одной из основных функций на множестве векторов является функция, определяющая длину вектора (или расстояние от точки с теми же координатами, что и у вектора, до начала координат):

λ
```
vecLength ::  Vector -> Float
```

Для двух векторов (одной размерности) можно выяснить, перпендикулярны ли они, или найти угол между ними:

λ
```
vecPerpendicular ::  Vector -> Vector -> Bool
vecAngle         ::  Vector -> Vector -> Float
```

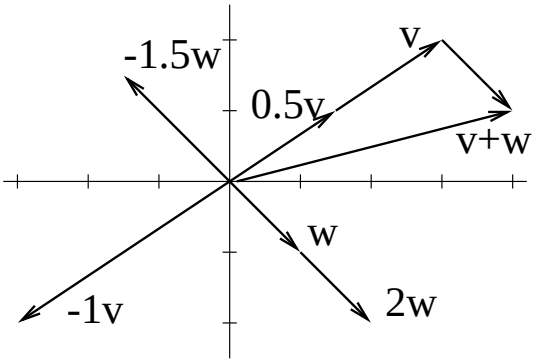
Вектор может быть умножен на число (что сводится к умножению на это число всех его координат), два вектора можно сложить:

λ
```
vecScale ::  Float -> Vector -> Vector
vecPlus  ::  Vector -> Vector -> Vector
```

Далее мы приведем определения этих функций. Сейчас лишь дадим их краткое описание:

- `vecScale`: получение коллинеарного вектора, лежащего на той же прямой, но деформированного в зависимости от аргумента следующим образом: если абсолютная величина параметра 1, то вектор сжимается, в противном случае — растягивается; если аргумент отрицательный, то вектор меняет свое направление на противоположное;
- `vecPlus`: второй вектор «прикладывается» к концу первого, после чего начало первого вектора соединяется с концом второго (в указанном порядке).

Следующий пример иллюстрирует эти операции для векторов в двумерном пространстве.




```
                    v  =  [3, 2]
                    w  =  [1, -1]
          vecPlus v w  =  [4, 1]
     vecScale 0.5 v    =  [1.5, 1]
     vecScale -1 v     =  [-3, -2]
      vecScale 2 w     =  [2, -2]
vecScale -1.5 w        =  [-1.5, 1.5]
```

Рассмотрим функции, преобразующие вектор в вектор. Особо интересны функции, преобразующие каждую из координат вектора в линейную комбинацию других координат. Так, для вектора на плоскости подобное преобразование задается формулой

$$f(x, y) = (a * x + b * y, c * x + d * y),$$

где a, b, c и d есть произвольные числа. Можно сказать, что любое линейное преобразование вектора в n-мерном пространстве однозначно определяется n^2 числами. Обычно они записываются в виде строк чисел, заключенных в скобки. Такой математический объект называется *матрица*. Например, следующая матрица описывает поворот вектора в трехмерном пространстве на 30° вокруг оси z:

$$\begin{pmatrix} \frac{1}{2}\sqrt{3} & -\frac{1}{2} & 0 \\ \frac{1}{2} & \frac{1}{2}\sqrt{3} & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Введем новый тип, описывающий матрицы как список списков:

```
data Matrix  = Mat [[Float]]
```

λ

Операция преобразования столбцов в строки и наоборот называется *транспонированием* матрицы и задается функцией `matTransp`, имеющей следующий тип:

```
matTransp  :: Matrix -> Matrix
```

λ

Например, `matTransp` $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$.

Наиболее важной из всех функций для работы с матрицами можно считать функцию, задающую линейное преобразование матрицы. Ее тип таков:

```
matApply ::  Matrix -> Vector -> Vector
```

λ

Композиция двух линейных преобразований также является линейным преобразованием. Две матрицы задают два линейных преобразования, композиция этих преобразований также задается матрицей. Такое преобразование часто называют *матричным произведением*. Тип этой функции таков:

λ
```
matProd ::  Matrix -> Matrix -> Matrix
```

Приведем пример умножения матриц:

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 2 & 2 \\ 2 & 3 & 4 & 1 \\ 1 & 3 & 4 & 2 \end{pmatrix} = \begin{pmatrix} 7 & 16 & 22 & 10 \\ 2 & 5 & 8 & 5 \end{pmatrix}$$

Так как операция композиции функций ассоциативна, то и функция `matProd` будет ассоциативна $(A \times (B \times C) = (A \times B) \times C$, однако операция умножения матриц *не коммутативна* (т. е. $A \times B$ в общем случае не равна $B \times A$). Матричное произведение $A \times B$ означает, что сначала применили преобразование B, а затем A.

Тождественное преобразование (ничего не меняющее) задается единичной матрицей. Его матрица содержит 1 на главной диагонали, а все остальные элементы равны 0. Можно определить функцию, которая выдает единичную матрицу указанной размерности:

λ
```
matId ::  Int -> Matrix
```

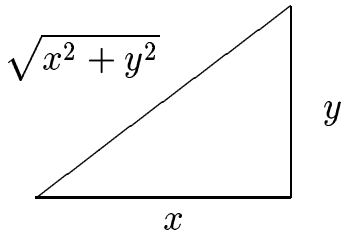
Некоторые линейные преобразования являются взаимно обратными. Обратное преобразование (если оно существует) также является линейным преобразованием и задается функцией

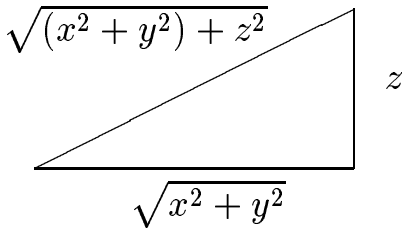
λ
```
matInv ::  Matrix -> Matrix
```

Матрица, полученная в результате обратного преобразования не обязательно имеет тот же размер, что и исходная. Условие совпадения количества строк с количеством столбцов (так называемая «квадратная» матрица) является необходимым, но не достаточным условием существования обратной матрицы, например, матрица $\begin{pmatrix} 1 & 2 \\ 3 & 6 \end{pmatrix}$ не имеет обратной.

Длина вектора

Длина вектора вычисляется с помощью теоремы Пифагора. Рисунок иллюстрирует ситуацию для 2-х и 3-х мерного пространства:





В общем случае длина вектора есть корень квадратный из суммы квадратов координат:

```
vecLength (Vec xs) = sqrt (sum (map square xs))
```
λ

Угол между двумя векторами

Рассмотрим два двумерных вектора единичной длины. Если начало векторов находится в точке с координатами $(0;0)$, то концы этих векторов находятся на окружности единичного радиуса с центром в начале координат. Пусть вектора образуют с осью Ox углы α и β соответственно, тогда их координаты задаются формулами $(\cos\alpha, \sin\alpha)$ и $(\cos\beta, \sin\beta)$. Угол между такими векторами равен $\beta - \alpha$. По формуле косинуса разности двух углов найдем

$$\cos(\beta - \alpha) = \cos\alpha\cos\beta + \sin\alpha\sin\beta$$

Выражение, стоящее в правой части формулы, называется *скалярным произведением векторов* (inner product). В общем случае скалярное произведение векторов можно найти с помощью функции

```
vecInprod (Vec xs)  (Vec ys)= sum (zipWith (*) xs ys)
```
λ

Для нахождения косинуса угла между векторами с длиной не равной единице, их предварительно следует нормировать, т. е. поделить все координаты на длину вектора. Теперь можно определить функцию

```
vecAngle v w = acos (vecInprod v w /
                     (vecLength v * vecLength w))
```
λ

Функция `acos` является обратной к косинусу (в русской литературе эта функция носит название arccos).

Косинус углов в $90°$ и $-90°$ равен нулю, что равнозначно равенству нулю скалярного произведения векторов, образующих эти углы, поэтому перпендикулярность определяется функцией

```
vecPerpendicular v w = vecInprod v w == 0
```
λ

Сложение и масштабирование векторов

Функции `vecScale` и `vecPlus` легко определяются через функции `map` и `zipWith`:

λ
```
vecScale k (Vec xs) = Vec (map (k*) xs)
vecPlus (Vec xs) (Vec ys) = Vec (zipWith (+) xs ys)
```

Подобные преобразования допустимы и при работе с матрицами. Разработаем две функции `mapl` и `ziplWith`, имена которых аналогичны функциям `map` и `zipWith`, но предназначенные для работы со списками списков.

λ
```
mapl  :: (a -> b) -> [[a]] -> [[b]]
mapl f = map (map f)
```

Альтернативное определение этой функции таково:

λ
```
mapl  = map . map
```

Функция `ziplWith` определяется аналогично:

λ
```
ziplWith   :: (a -> b -> c) -> [[a]] -> [[b]] -> [[c]]
ziplWith = zipWith . zipWith
```

С помощью этих функций можно определить `matScale` и `matPlus`:

λ
```
matScale   :: Float -> Matrix -> Matrix
matScale k (Mat xss)  = Mat (mapl (k*) xss)
matPlus  :: Matrix -> Matrix -> Matrix
matPlus (Mat xss) (Mat yss) = Mat (ziplWith (+) xss yss)
```

Транспонирование матриц

Транспонированная матрица — это матрица, у которой строки и столбцы поменялись местами. Эту операцию можно применять и к обычным спискам списков (не использующих конструктор `Mat`). Поэтому сначала определим функцию `transpose`:

λ
```
transpose   :: [[a]] -> [[a]]
```

После чего становится ясным определение функции `matTransp`:

λ
```
matTransp (Mat xss) = Mat (transpose xss)
```

Функция `transpose` есть обобщение функции `zip`. В то время как `zip` соединяет два списка в список пар, `transpose` собирает список списков в новый список списков. Определим эту функцию рекурсивно, но сначала посмотрим на примере как она работает:

H
```
---> transpose [[1,2,3], [4,5,6], [7,8,9], [10,11,12]]
[[1,4,7,10],[2,5,8,11],[3,6,9,12]]
```

Если список списков состоит только из одной строки (базовый случай рекурсии), то следует создать список списков, в котором каждый список состоит только из одного элемента:

```
transpose [row] = map singleton row
     where singleton x = [x]
```

λ

Для получения рекурсивного случая представим себе, что уже все транспонировали, за исключением первой строки. В рамках предыдущего примера это эквивалентно следующему:

```
---> transpose [[4,5,6], [7,8,9], [10,11,12]]
[[4,7,10],[5,8,11],[6,9,12]]
```

H

Как же скомбинировать первую строку с полученным частичным решением для получения общего случая? Ее элементы следует добавлять по одному в начало каждого списка из рекурсивного решения: 1 нужно добавить в начало `[4,7,10]`, 2 поместить в начало `[5,8,11]`, а 3 в начало `[6,9,12]`. Эту операцию легко осуществить с помощью `zipWith`:

```
transpose (xs :xss) = zipWith (:) xs (transpose xss)
```

λ

Единственный случай, упущенный нами из рассмотрения, — матрица с нулевым количеством строк, но транспонировать подобную матрицу и нет необходимости.

Нерекурсивная функция транспонирования матрицы

Можно предложить нерекурсивную версию `transpose`, основанную на вызове стандартной функции `foldr`. Отметим следующее свойство `transpose`

```
transpose (y:ys) = f y (transpose ys),
```

где `f` есть частично параметризованная функция `zipWith (:)`. В качестве начального элемента операции свертки естественно положить `transpose []`.

Пустую матрицу можно трактовать как матрицу из 0 строк и n-столбцов. После транспонирования подобная матрица превратится в матрицу из n пустых списков. Но, чему равно число n? В начале у нас была просто матрица без строк, поэтому нельзя сказать, что n — длина первой строки. Поэтому, чтобы не оказаться в ситуации, когда заданное нами в этом случае число n окажется недостаточно велико, возьмем ∞ в качестве n. Таким образом, транспонирую матрицу из 0 строк и ∞ элементов, мы получим матрицу из ∞ строк, содержащих пустые списки. Никаких проблем с

применением функции `zipWith` к такому списку не будет, так как второй аргумент этой функции является конечным списком.

Результатом наших рассуждений является очень изящная версия функции `transpose`:

λ
```
transpose  = foldr f e
       where f = zipWith (:)
             e = repeat []
```

Эту версию можно переписать и в виде

λ
```
transpose  = foldr (zipWith (:)) (repeat [])
```

Функциональное программирование есть программирование посредством функций!

Применение матрицы к вектору

Матрица задает линейное преобразование векторов. Функция `matApply` осуществляет это преобразование, например,

```
matApply (Mat [[1,2,3], [4,5,6]]) (Vec [x,y,z]) =
         Vec [1x+2y+3z, 4x+5y+6z]
```

Число столбцов в матрице равняется числу координат исходного вектора, а число строк — количеству координат получившегося вектора.

Для вычисления одной из координат результирующего вектора потребуются значения только из одной строки матрицы. Это дает возможность задать `matApply` с помощью функции `map`:

λ
```
matApply (Mat m) v = Vec (map f m)
```

Функция `f` берет одну строку из матрицы и возвращает одну из координат результирующего вектора, например, для второй строки имеем

```
f [4, 5, 6] = 4x + 5y + 6z
```

функция `f` вычисляет линейное произведение (в данном случае с вектором `v`, имеющим координаты `[x, y,z]`). Поэтому можно записать

λ
```
matApply ::  Matrix -> Vector -> Vector
matApply (Mat m) v = Vec (map f m)
          where f row = vecInprod (Vec row) v
```

Единичная матрица

Для любого пространства указанной размерности существует только одна единичная матрица: это квадратная матрица, у которой число строк и столбцов совпадает с размерностью пространства, на ее главной диагонали находятся единицы, а все остальные места заполнены нулями. Перед разработкой функции, которая возвращает единичную матрицу указанной размерности, полезно написать функцию, создающую матрицу бесконечного размера:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \dots \\ 0 & 0 & 1 & 0 & \\ 0 & 0 & 0 & 1 & \\ \vdots & & & & \ddots \end{pmatrix}$$

Первая строка такой матрицы есть бесконечный список из нулей, содержащий единицу в начале, т. е. `1 : repeat 0`. Все другие строки получаются добавлением еще одного нуля в начало списка. Используя функцию `iterate`, можно записать

```
matIdent :: Matrix
matIdent = Mat (iterate (0:) (1: repeat 0))
```

λ

Единичная матрица размерности n будет получена, если из бесконечной матрицы взять первые n столбцов из первых n строк:

```
matId ::  Int -> Matrix
matId n = Mat ( map (take n) (take n xss))
          where (Mat xss) = matIdent
```

λ

Определитель матрицы

Только взаимнооднозначное отображение задает матрицу, которая имеет обратную. Матрица, обратная к матрице A, обозначается как A^{-1} и удовлетворяет соотношению $A \times A^{-1} = E$, где E единичная матрица. Только квадратная матрица имеет обратную. Но даже не каждая квадратная матрица имеет обратную, так, например, матрица $\left(\begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}\right)$ которая преобразует любую точку к началу координат, не имеет обратной; матрица $\left(\begin{smallmatrix} 1 & 1 \\ 2 & 2 \end{smallmatrix}\right)$ каждую точку преобразует к точке лежащей на прямой $y = 2x$ и также не имеет обратной; матрица $\left(\begin{smallmatrix} 1 & 2 \\ 3 & 6 \end{smallmatrix}\right)$ преобразует все точки в точки, лежащие на одной прямой, также не имеет обратной. Только если

вторая строка двумерной матрицы не пропорциональна первой, можно вычислить обратную к ней матрицу.

Другими словами, матрица $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ имеет обратную, если $\frac{a}{c} \neq \frac{b}{d}$, что эквивалентно $ad - bc \neq 0$. Величина $ad - bc$ называется определителем (или детерминантом) матрицы 2×2. Если определитель равен нулю, то матрица не имеет обратной.

Определитель квадратной матрицы большей размерности также можно вычислить. Для матрицы размером 3×3 определитель вычисляется так:

$$\texttt{det} \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = a \times \texttt{det} \begin{pmatrix} e & f \\ h & i \end{pmatrix} - b \times \texttt{det} \begin{pmatrix} d & f \\ g & i \end{pmatrix} + c \times \texttt{det} \begin{pmatrix} d & e \\ g & h \end{pmatrix}$$

Определитель матрицы любого порядка может быть сведен к вычислению определителей матриц меньшего порядка. Для этого выбирают любую строку (первую в нашем примере), после чего складывают выражения, полученные путем умножения числа 1, либо -1 (для первого слагаемого 1, затем числа чередуются), на величину элемента, расположенного в очередном столбце этой строки, и на определитель матрицы, полученной после *удаления данной строки и текущего столбца из исходной матрицы.*

Проектирование функции `det` начнем с неформального описания определителя, который требуется вычислить. Мы можем использовать разделение матрицы на первую строку и оставшуюся часть матрицы:

λ
```
det (Mat (row:rows)) = ...
```

Нам нужно будет удалить соответствующий столбец из матрицы `rows`. Вместо удаления по одному элементу из каждой строки значительно проще транспонировать матрицу и уже из транспонированной матрицы удалить целиком строку. Для этого получим список списков, содержащий все возможные матрицы, полученные после удаления одного из столбцов. Данную операцию удобно выполнить с помощью функции `gaps`, образующей список списков, полученных всевозможными удаленьями одного элемента из списка, например,

H
```
---> gaps [1,2,3,4]
[[2,3,4],[1,3,4],[1,2,4],[1,2,3]]
```

Оставив в качестве самостоятельного задания разработку этой функции, вернемся к написанию функции, вычисляющей определитель. Полученные списки списков следует транспонировать обратно и с помощью конструктора `Mat` превратить в матрицы:

```
det (Mat (row:rows)) = ...                                        λ
    where cols = transpose rows
          mats = map (Mat . transpose) (gaps cols)
```

Для всех полученных матриц следует вычислить определители, для чего придется использовать рекурсию. затем полученные определители следует умножить последовательно на все элементы первой строки:

```
det (Mat (row:rows)) = ...                                        λ
    where cols  = transpose rows
          mats  = map (Mat . transpose) (gaps cols)
          prods = zipWith (*) row (map det mats)
```

Произведения `prods` должны быть поочередно умножены на $+1$ или -1 и затем сложены. Для выполнения этой операции воспользуемся функцией `altsum` (суммирование с альтернативами), использующей бесконечный список:

```
altsum xs = sum (zipWith (*) xs plusMinusOne)                     λ
    where plusMinusOne = 1 : -1 : plusMinusOne
```

Запишем итоговую функцию, вычисляющую определитель матрицы:

```
det (Mat (row:rows)) = altsum prods                               λ
    where cols  = transpose rows
          mats  = map (Mat . transpose) (gaps cols)
          prods = zipWith (*) row (map det mats)
```

Полученную функцию можно слегка упростить, заметив, что нет необходимости в обратном транспонировании матриц, так как определитель транспонированной матрицы совпадает с определителем исходной:

```
det (Mat (row:rows)) = altsum ( zipWith (*) row                   λ
    (map det (map (Mat (gaps (transpose rows))))))
```

Так как `det` — рекурсивная функция, необходимо учесть базовый случай. Определитель матрицы 1×1 совпадает со значением ее единственного элемента:

```
det (Mat [[x]]) = x                                               λ
```

И, наконец, окончательный результат:

λ
```
det                  :: Matrix -> Float
det (Mat [[x]])      = x
det (Mat (row:rows)) =
         (altsum . zipWith (*) row . map det .
         map Mat . gaps . transpose) rows
```

Вычислим определитель единичной матрицы четвертого порядка:

H
```
---> det (Mat [[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])
1.0
```

Вопросы и задания

VII.4.1

Напишите функцию `dayBefore`, возвращающую день, предшествующий данному.

VII.4.2

Определите тип `Direction`, чьи значения описывают четыре главных точки на компасе, и определите функцию `revCorner` для выдачи противоположного направления.

VII.4.3

В логике следствие (импликация, implication), обозначаемое как $\Rightarrow$, определяется условием, что $x \Rightarrow y$ ложно тогда и только тогда, когда x — истинно, а y — ложно. Дайте формальное определение импликации как операции на множестве `Bool`.

VII.4.4

Сколько уравнений потребуется вам написать, чтобы определить функцию, возвращающую значение типа `Triangle`, выведенного из класса `Ord`? (См. стр. 180.)

VII.4.5

Существуют ли числа, которые можно сравнить при помощи оператора `(==)`, но нельзя сравнить при помощи `(<)`?

VII.4.6

Это за величины получатся в результате вычисления следующих выражений?

```
show (show 42)
show 42 ++ show 42
show "\n"
```

VII.4.7

Напишите функцию `gaps`, которая возвращает список списков, полученных всевозможными удаленьями одного элемента из исходного списка, например,

```
---> gaps [1,2,3,4,5]
[[2,3,4,5],[1,3,4,5],[1,2,4,5],[1,2,3,5],[1,2,3,4]]
```

VII.4.8

Напишите функцию `matProd` для перемножения двух матриц.

VII.4.9

Напишите функцию `matInv` для определения обратной матрицы.

VII.4.10

Для отображения результатов функций, возвращающих вектора и матрицы, следует в декларации `data` этих типов отметить, что они наследуют свойства класса `Show`. Однако вывод, предлагаемый этим классом по умолчанию, не очень нагляден. Используя декларацию `instance` объявите типы `Vec` и `Matrix` как экземпляры класса `Show` и создайте свои функции `show`, перегружающие данный метод класса `Show`. Добейтесь того, чтобы вывод, осуществленный при помощи разработанных вами функций, соответствовал общепринятым математическим традициям.

5. Инкапсуляция данных в модулях

5.1. Необходимость сокрытия данных

Использование локальных переменных в конструкциях `let` и `where` является одним из способов сокрытия внутренних деталей реализации одной программной компоненты от других. Сущности, определяемые в `where`-фразах, доступны только внутри такого определения. Таким образом, `where`-предложения, являются некоторым способом скрыть от доступа извне (инкапсулировать, encapsulate) информацию. Величины, скрытые подобным образом, защищены от воздействия других определений. Тем не менее, все

же лучше сохранять `where`-предложения настолько короткими, насколько это возможно. Если они достаточно длины, то они содержат много переменных, что может привести к возникновению различных конфликтов.

Доступ к сущностям может также контролироваться определением их в программных единицах, известных как *модули*. Величины, определенные в модулях, могут быть как общедоступными (*public*), доступ к которым разрешен извне модуля, или локальными (*private*), доступные только внутри модуля. Такой подход дает возможность определять программные единицы, независимые друг от друга, взаимодействующие друг с другом неким заранее оговоренным способом, позволяющих добраться до общедоступных величин. Применение модулей делает возможными изменения внутреннего содержания модулей без затрагивания других частей программного обеспечения. О локальных величинах, определенных в модуле, говорят, что они «инкапсулированы в модуле».

5.2. Модули: экспорт и импорт

Программа на языке Haskell является коллекцией *модулей*. Основное назначение модулей состоит в инкапсулировании данных, что позволяет осуществлять контроль над пространством имен и создавать абстрактные типы данных.

Имена модулей состоят из алфавитных символов, начинаясь при этом с заглавной буквы. Нет никакой формальной связи между именами модулей и файловой системой. В частности нет связи между именем модуля и именем файла, в котором содержится описание модуля. Но обычно все же придерживаются *соглашения о взаимно-однозначном соответствии между именами модуля и файла*.

По договоренности, каждый модуль размещают в отдельном файле, причем имя файла совпадает с именем модуля с точностью до расширения (добавляется либо `.hs`, либо `.lhs`). Так, например, модуль `DecimalNumerals` должен размещаться в файле `DecimalNumerals.hs` (или `DecimalNumerals.lhs`). Есть единственное исключение: если это главный модуль программы (`Main`), то имя файла обычно отражает назначение данной программы, иначе имелось бы огромное число файлов с названием `Main.hs`.

Величины и типы не смешиваются в Haskell

В этом языке есть шесть родов имен:

— для переменных и конструкторов, обозначающих величины (конструкторов данных),
— для типовых переменных,
— для конструкторов типа,
— для классов типов и
— имена модулей.

При их использовании придерживаются следующих синтаксических правил.

- Имена переменных (величин) и типовых переменных начинаются с маленькой латинской буквы или подчеркивания; другие четыре вида имен начинаются с заглавной буквы.
- Имена *операторов*-конструкторов должны начинаться с символа : (двоеточие), имена других операторов не должны начинаться с двоеточия.

Модуль — это скрипт, который предназначен для определения некоторых сущностей, предназначенных для использования (*экспорта*) в других скриптах, и скрывающий при этом все определения внутри себя. Другой модуль, *импортирующий* подобные сущности, может их использовать, но не имеет доступа к их определениям.

Модуль начинается с ключевого слова `module`, за которым следует имя модуля. После имени модуля следует список экспорта, т. е. перечень сущностей (типов данных, функций), которые будут видны при импорте этого модуля, заключенный в круглые скобки. Все величины, используемые в модуле и не вошедшие в список экспорта, являются локальными и не доступны извне модуля. Ключевое слова `where`, следующее сразу за списком экспорта, активирует контекст, поэтому все нижерасположенные декларации *должны начинаться одной и той же позиции строки* (как правило, с первой).

Вот пример модуля, носящего название `Tree`. Этот модуль явно экспортирует конструктор типа `Tree` вместе с соответствующими конструкторами данных `Leaf`, `Branch` и функцию `fringe`:

```
module Tree ( Tree (Leaf, Branch), fringe ) where

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

λ

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

Описанный здесь тип данных `Tree` хранит информацию в листьях; функция `fringe` «разглаживает» дерево, превращая его в список. Заметим, что здесь мы использовали одно и тоже имя для обозначения модуля и типа данных. Подобные совпадения допустимы, так как они не приводят к конфликтам имен.

Модули можно *импортировать* в другие модули и скрипты. Декларация импорта должна быть *первым выполняемым оператором* в файле, т. е. размещаться сразу после декларации модуля (напомним, что комментарии допускаются в любом месте программы, где можно вставить пробел):

λ
```
module Main where
{-
Модуль Main импортирует модуль Tree
-}
import Tree (Tree( Leaf, Branch), fringe)

t1 :: Tree Int
t1 = Branch (Leaf 2) (Branch (Leaf 3)(Leaf 4))
```

С импортированным типом данных и функциями можно оперировать так, словно они определены в этом модуле:

H
```
---> fringe t1
[2,3,4]
```

Импортировать модули можно не только в модули, но и в скрипты, являющиеся просто набором функций:

λ
```
-- Скрипт импортирует модуль Tree

import Tree (Tree( Leaf, Branch), fringe)

t2 :: Tree Int
t2 = Branch (Leaf 2) (Leaf 4)
```

Вызов функций из такого скрипта ничем не отличается от предыдущего случая:

H
```
---> fringe t2
[2,4]
```

За исключением того факта, что *декларация* `import`*, размещается первой*, остальные декларации (`data`, `newtype` и `type` декларации, `class` и `instance` декларации, сигнатуры типов, определения функций и т. д.) могут размещаться в произвольном порядке. Можно сказать, что модуль в действительности является одной большой декларацией, которая начинается с ключевого слова `module`.

Если экспортируемый список, следующий за ключевым словом `module` отсутствует, то *все* имена в рамках верхнего уровня модуля будут экспортированы. При импорте модуля не обязательно импортировать все его функции, включенные в список экспорта: достаточно указать только те, которые будут использоваться.

▶ *Упражнение VII.5.1*

Пусть модуль `Ex1` содержит следующие определения:

```
module Ex1 (f, g) where
 f :: Int -> Int
 f x = (g . g) x
 g x =  x * x
```

Так как все функции, определенные в нем включены в список экспорта, то его можно не указывать:

```
module Ex1  where
 f :: Int -> Int
 f x = (g . g) x
 g x =  x * x
```

В модуль `MainEx1` включим декларацию импорта модуля `Ex1`, причем импортируем только одну из двух доступных нам функций:

```
module MainEx1 where
 import Ex1 (f)
```

Теперь функция `f` доступна, а функция `g` — нет:

```
---> f 3
81
---> g 3
ERROR - Undefined variable "g"
```

◀

Имена, включенные в список экспорта, не могут быть локальными определениями.

▶ *Упражнение VII.5.2*

Пусть в модуле `Ex2` определение функции `f` использует локальную переменную `g`. Отсутствие списка экспорта означает экспорт всех доступных определений:

λ
```
module Ex2 where

f x = (g . g) x
      where g x = x * x
```

В модуль `MainEx2` попробуем импортировать функции `f` и `g`:

λ
```
module MainEx2 where
 import Ex2 (f, g)
```

При попытке загрузить этот модуль будет выдано сообщение об ошибке, информирующее об отсутствии функции `g` среди всех сущностей, экспортированных из модуля `Ex2`:

H
```
ERROR "MainEx2.hs" - Unknown entity "g"
          imported from module "Ex2"
```

Появление этого сообщения связано с тем, что локально определенная функция `g` не видна при экспорте.

◀

Экспортируемый тип данных и его конструкторы должны быть сгруппированы вместе, например, `Tree(Leaf, Branch)`. В более короткой форме это может быть записано так: `Tree(..)`. Эспорт только некоторого подмножества (даже пустого) конструкторов также возможен. Сокрытие конструкторов данных и экспорт одного лишь конструктора типа используется при создании *абстрактного типа данных.*

Список импорта может содержать только те сущности, которые включены в список экспорта данного модуля. В список импорта желательно включать только те сущности, которые действительно потребуются в работе, не обязательно перечислять в нем весь список экспорта. Список импорта может быть даже пустым, в этом случае импортируется все содержимое списка экспорта.

Некоторые сущности могут быть явно *исключены* из списка импорта с помощью конструкции `hiding (name1, name2, ...)`, где `name1, name2` и так далее, есть имена «скрываемых» сущностей.

Модуль с названием `Prelude` по умолчанию импортируется во все программы. Мы уже использовали ранее в наших программах строку импорта файла `Prelude` с указанием ключевого слова `hiding` (hide — прятать) для того, чтобы скрыть автоматически загруженное определение той или иной функции.

5.3. Контроль пространства имен

Есть некоторая проблема, связанная с импортом имен одного модуля в пространство имен другого модуля. Что произойдет, если два импортируемых модуля содержат отличные друг от друга определения, но названные одним и тем же именем? Haskell решает эту проблему с помощью квалификатора имен `qualified`.

Объявление импорта может содержать ключевое слово `qualified`, позволяющее указать имя импортированного модуля перед именем метода, разделяя их символом `.` (точка). Обратите внимание на отсутствие пробелов между именами модуля, метода и точкой. Следующие две синтаксические конструкции имеют совершенно разный смысл: `A.x` есть ограничение имени, в то время как `A . x` — применение инфиксной функции `.` (композиция).

Посмотрим на пример использования квалификатора в декларации импорта. Пусть в модуле `Fringe`, также, как и в модуле `Tree`, определена функция `fringe`:

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

fringe :: Tree a -> [a]    -- другое определение fringe
fringe (Leaf x) = [x]
fringe (Branch x y) = fringe x
```

Модуль `Main` *импортирует* функцию `fringe` из двух модулей:

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

l1 = fringe (Branch (Leaf 1) (Leaf 2))
l2 = Fringe.fringe (Branch (Leaf 1) (Leaf 2))
```

Список `l1` получен с помощью функции `fringe` из модуля `Tree`, а список `l2` — с помощью функции `fringe` из модуля `Fringe`:

H
```
---> :l Qualified.hs
Reading file "Qualified.hs":
Reading file "Tree.hs":
Reading file "Fringe.hs":
Reading file "Qualified.hs":

Hugs session for:
/usr/share/hugs/lib/Prelude.hs
Tree.hs
Fringe.hs
Qualified.hs
---> l1
[1,2]
---> l2
[1]
```

Как видим, несмотря на одинаковые имена функций, выполняемые ими действия совершенно различны.

Некоторые программисты предпочитают использовать спецификаторы имен во всех декларациях импорта, что позволяет полностью избежать неясностей в вызове методов. Другие предпочитают короткие имена и используют спецификаторы только в случае необходимости.

Возможность использования квалифицированных имен позволяет полностью контролировать доступ к требуемому объекту, что дает возможность определять в модуле функции и другие величины с теми же именами, что и импортируемые сущности.

▶ *Упражнение VII.5.3*

Импортируем модуль `Prelude`, указав при этом ключевое слово `qualified`. После подобной декларации, имена всех функций, определенных в прелюдии, следует указывать с помощью префикса, но зато появилась возможность определять новые функции (или операторы) с теми же именами:

λ
```
import qualified Prelude
import List

xs + ys = xs ++ ys
succ = (Prelude.+ 1)
```

Теперь для операции соединения («сложения») двух списков можно использовать привычный символ `+`. Но при этом осталась возможность использования стандартного оператора `+`, для чего достаточно указать квалификатор перед его именем.

```
---> [1..3] + [3..6]
[1,2,3,3,4,5,6]
---> succ 5
6
```

При импорте можно дать новое (например, более короткое) имя импортируемому модулю, добавив к декларации импорта ключевое слово `as`, после которого указывается псевдоним (alias) модуля, например:

```
import qualified Complex as C
```

После такого объявления можно употреблять более короткий префикс `C.` вместо `Complex.` — стандартного префикса. Допустимо использование псевдонимов и в неквалифицированных декларациях:

```
import Foo(f) as A
```

Теперь к импортируемой функции можно обращаться как `f`, так и `A.f`.

Декларации экземпляра (`instance`) не могут быть включены в список экспорта или импорта, так как все такие декларации, содержащиеся в модуле, *всегда экспортируются* и все они будут видны после импорта такого модуля. Наряду с экземплярами классов, не могут быть скрыты при импорте и классы, определенные в прелюдии (стр. 170).

Квалификатор имени только идентифицирует модуль, из которого производится импорт; это имя может отличаться от имени того модуля, где действительно определена экспортируемая сущность. Например, если модуль `A` экспортирует `B.c`, то для доступа к этой величине следует указывать `A.c`, а не `A.B.c`.

5.4. Модуль как способ реализации АТД

Кроме контролирования пространства имен, модули предоставляют способ построения абстрактных типов данных, сокращенно АТД (abstract data types, ADT). Характерной чертой абстрактного типа данных может считаться сокрытие представления типа, все

операции с АТД осуществляются на абстрактном уровне, скрывающим реальное представление данных. Так, например, хотя тип `Tree` достаточно прост, но мы его еще не можем считать абстрактным, потому что при работе с величинами этого типа нам явно приходится указывать какие конструкторы данных требуются для создания и использования величины данного типа.

«Хороший» абстрактный тип отражает важнейшие свойства объектов, не привлекая внимания к несущественным деталям реализации. Другими словами, хорошая абстракция явно отражает потребности пользователя и скрывает особенности реализации. Подходящий АТД, предназначенный для работы с древовидной структурой, может включать следующие операции (из перечня которых и формируется список экспорта):

```
data Tree a              -- только 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
```

Модуль, позволяющий работать с этим АДТ, может выглядеть так:

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

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

leaf = Leaf
branch = Branch
cell (Leaf a) = a
left (Branch l r) = l
right (Branch l r) = r
isLeaf (Leaf _) = True
isLeaf _        = False
```

Обратите внимание, что список экспортируемых функций содержит только имя типа `Tree`, т. е. без конструкторов данных. Так как `Leaf` и `Branch` не экспортированы, то их нельзя уже использовать для построения или выделения частей деревьев, для этих целей следует использовать определенные в модуле функции, обеспечивающие доступ к конструкторам данных типа `Tree`.

Попробуем определить функции для работы с типом `Tree` так, как мы это делали раньше — с помощью образцов, содержащих конструкторы данных:

```
module MainADT1 () where
```

```
import TreeADT

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

Но конструкторы типа `Tree` скрыты, поэтому сопоставление с образцами невозможно:

```
Dependency analysis
ERROR "MainADT1.hs":6 - Undefined
         constructor function "Leaf"
```

Для того чтобы задать величину типа `Tree`, можно использовать только те функции, которые включены в список экспорта:

```
module MainADT () where
import TreeADT

fringe :: Tree a -> [a]
fringe x = if isLeaf x then [cell x]
           else fringe (left x) ++ fringe (right x)
```

Теперь можно работать с данным типом, как с абстрактным:

```
---> fringe (branch (branch (leaf 3) (leaf 2)) (leaf 1))
[3,2,1]
```

Несомненным преимуществом такого представления данных является то, что в дальнейшем можно легко изменить внутреннее представление типа `Tree` без переписывания всего программного обеспечения, которое его использует.

▶ *Упражнение VII.5.4*

Одной из широко известных абстрактных структур данных является *стек*. Элементы помещаются в стек по одному. Только последний элемент стека доступен в каждый момент. Иногда стек сравнивают с коробкой, содержащей листы бумаги. Новый лист кладется в стопку поверх остальных. Только верхний лист может быть прочитан или извлечен из коробки. Для того чтобы извлечь

некоторый лист из коробки, необходимо сначала извлечь все лежащие над ним листы.

Стек функционирует аналогично. Новый элемент помещается в вершину стека с помощью операции `push` (втолкнуть). Виден в стеке только его верхний элемент, значение которого может быть получено с помощью операции `top` (вершина). Наконец, верхний элемент может быть извлечен из стека операцией `pop` (вытолкнуть). Иногда операции `top` и `pop` объединяют при реализации. О стеке также говорят как об очереди типа LIFO (Last In First Out, последний пришел — первый вышел).

Определение стека как АТД начинается с описания его операций на абстрактном уровне:

```
emptyStack    :: Stack a
push          :: a -> Stack a -> Stack a
pop           :: Stack a -> Stack a
top           :: Stack a -> a
isEmpty       :: Stack a -> Bool
showStack     :: (a -> [Char]) ->  Stack a -> [Char]
```

К трем основным операциям обычно добавляют еще несколько функций, которые делают работу со стеком более удобной, например, `emptyStack`, создающую пустой стек, `isEmpty`, проверяющую, пуст ли стек, и `showStack`, печатающую содержимое стека.

Предположим, что АТД, описывающий стек уже реализован, и разработаем модуль для работы с этим типом. Так как приведенный выше интерфейс стека полиморфен, создадим стеки, предназначенные для хранения информации различных видов: символов, целых чисел и записей, содержащих информацию об имени и возрасте детей:

λ
```
module Main where
import Stack (Stack,
              emptyStack,
              push,
              pop,
              top,
              isEmpty,
              showStack )

-- стек символов --
stack1 :: Stack Char
stack1 = push 'd' (push 'f' (push 'k' emptyStack))
```

```
s1 = putStr(showStack show stack1)

-- стек целых чисел --
stack2 :: Stack Int
stack2 = emptyStack
stack3 = push 5 (push 5 (push 2  stack2))
stack4 = pop stack3
s4 = putStr(showStack show stack4)

-- стек пар (Имя, Возраст) --
stack5 :: Stack Record

type Age = Int
type Name = [Char]
type Record = (Name,Age)

showRec (name,age) = "(" ++ name ++ "," ++ show age ++ ")"

stack5 = push  ("Kate",6) (push ("Jan",5)
                (push ("Ann",10) emptyStack))
s5 = putStr(showStack showRec stack5)
```

Приступим к разработке модуля Stack. Наверное, одной из самых простых его реализаций является реализация на базе списка:

```
module Stack (Stack,                                          λ
              emptyStack,
              push,
              pop,
              top,
              isEmpty,
              showStack ) where

-- реализация на базе списка --
type Stack a =  [a]
-----------------------
isEmpty     :: Stack a -> Bool
emptyStack  :: Stack a
push        :: a      -> Stack a -> Stack a
top         :: Stack a -> a
pop         :: Stack a -> Stack a
showStack   :: (a -> [Char]) ->  Stack a -> [Char]
```

```
----------------------

isEmpty [] = True
isEmpty _  = False

emptyStack = []

push item oldlist = item:oldlist

top (first:_) = first

pop (_:rest) = rest
pop []       = error "pop:empty stack"

showStack f [] ="Bottom"
showStack f (x:xs) = f x ++ " , " ++ showStack f xs
```

Загрузив модуль `Main` можно приступать к работе с абстрактным типом `Stack`:

H
```
---> s1
'd' , 'f' , 'k' , Bottom
---> s4
5 , 2 , Bottom
---> s5
(Kate,6) , (Jan,5) , (Ann,10) , Bottom
```

Теперь представим, что данные, помещаемые в стек, поступают сериями из повторяющихся величин, например, на ввод поступают нули и единицы в следующем порядке: 1,1,1,0,1,0,0,0,1,1,1,1,1,0,0,0. Эту же последовательность можно записать в виде пар `(величина, количество)`: (1,3), (0,1), (1,1), (0,3), (1,5), (0,3). Такой метод хранения значительно эффективнее, если средняя длина повторяющихся групп достаточно велика.

Ниже представлена реализация стека, использующая для хранения данных список пар:

λ
```
module Stack' (Stack,
               emptyStack,
               push,
               pop,
               top,
               isEmpty,
```

```
              showStack ) where

-- реализация на базе списка пар --
type Stack a =  [(a, Int)]
-----------------------
isEmpty     :: Stack a -> Bool
emptyStack  :: Stack a
push        :: Eq a => a -> Stack a -> Stack a
top         :: Stack a -> a
pop         :: Stack a -> Stack a
showStack   :: (a -> [Char]) ->  Stack a -> [Char]
-----------------------

isEmpty [] = True
isEmpty _  = False

emptyStack = []

push item  [] = [(item,1)]
push item oldlist
   | top_item == item =((item,top_cnt+1):rest)
   | otherwise        = ((item,1):oldlist)
   where
      (top_item,top_cnt):rest = oldlist

top (( top_item,_):_) = top_item
top _                 = error "top:empty stack"

pop  ((top_item,1):rest) = rest
pop  ((top_item,n):rest) = ((top_item,n-1):rest)
pop _                    = error "pop:empty stack"


showStack f [] ="Bottom"
showStack f ((x,1):xs) = f x ++ " , " ++
                         showStack f xs
showStack f ((x,n):xs) = f x ++ " , " ++
                         showStack f ((x,n-1):xs)
```

◀

5.5. Библиотеки программ

Многолетний опыт многих тысяч программистов показал, что написание большой программной системы значительно отличается от разработки отдельных небольших программ. В этих случаях требуются применение дополнительных методов и инструментальных средств, выходящих за рамки простого программирования. Термин *проектирование программного обеспечения* (software ingineering) используется для обозначения методов и инструментальных средств, предназначенных для проектирования, конструирования и управления при создании программных систем.

Любой инженер знает о преимуществах, связанных с использованием готовых компонент при разработке того или иного проекта. Подобное применение заимствованных компонент является распространенной разновидностью инженерной деятельности. В инженерии программ также весьма полезны заимствованные компоненты, называемые в различных языках программирования модулями (Haskell, Modula), пакетами (Ada) или классами (в C++, Java). Использование программных модулей дает ряд преимуществ. Эти средства структурирования программ дают возможность инкапсуляции данных и методов в более крупные единицы. Еще одним преимуществом модулей является то, что их согласование можно проверить при компиляции, чтобы предотвратить ошибки и недоразумения.

Первые языки программирования не позволяли разбивать программы на отдельные части, что приводимо к сложностям их восприятия и отладки. Опыт показал, что, возможно, 10000 строк является верхним пределом для монолитной программы. В стандарте языка Pascal, например, не определено никакого метода для раздельной компиляции или декомпозиции программ. Первый Pascal-компилятор был единой программой, содержащей свыше 8000 строк кода на языке Pascal. Вместо того, чтобы изменять Pascal, его создатель Никлаус Вирт разработал новый (хотя и похожий) язык, названный Modula, так как центральным понятием в нем, является модуль.[1] Его идеи нашли отражение во многих языках программирования, в том числе и в Haskell.

[1]Компиляторы Pascal, используемые на практике, поддерживают декомпозицию на модули, но никаких стандартных методов для этого не существует, так что большие программы на Packal не переносимы с одной платформы на другую.

Повторное использование программного кода во вновь создаваемых приложениях имеет ряд преимуществ.

- Программисту не требуется разрабатывать уже существующий алгоритм.
- Организация-разработчик экономит ресурсы.
- Программы, как правило, становятся надежнее.
- Программы, как правило, становятся эффективнее.

Большинство современных языков программирования поставляются с набором «стандартных» библиотек программ, и Haskell не является исключением. Обычно в той же директории, где располагается файл `Prelude.hs` (во многих Linux-системах это директория `/usr/share/hugs/lib/`) находятся скрипты, содержащие программные модули, реализующие многие широко известные алгоритмы и структуры данных.

▶ *Упражнение VII.5.5*

Среди библиотечных модулей находится модуль `Complex`, предназначенный для работы с комплексными числами. Любое комплексное число представимо в виде $z = x + i \cdot y = r \cdot e^{i\varphi}$, где i есть мнимая единица ($i^2 = -1$). Величины x и y называются действительной и мнимой частью числа и задают координаты точки на плоскости, соответствующей данному числу; r равно расстоянию от этой точки до начала координат, а φ — углу между осью $0x$ и вектором с координатами $(x; y)$.

Просматривая список экспорта данного модуля, мы встречаем все стандартные функции для работы с комплексными числами. Таблица поясняет их назначение.

Метод	**Назначение**
`(:+)`	Оператор создания комплексного числа по заданным x и y
`realPart`	Выделение действительной части числа
`imagPart`	Выделение комплексной части числа
`conjugate`	Вычисление сопряженного комплексного числа (если $z = x + iy$, то сопряженное число равно $x - iy$)
`mkPolar`	Вычисление комплексного числа, заданного полярными координатами r и φ

`cis`	Получение «нормированного» комплексного числа $(x/r + i \cdot y/r = \cos\varphi + i \sin\varphi)$
`polar`	Вычисление полярных координат точки, соответствующей заданному комплексному числу
`magnitude`	Вычисление $r = \sqrt{x^2 + y^2}$ — модуля комплексного числа
`phase`	Вычисление φ — аргумента комплексного числа

Не зная, как реализован абстрактный тип данных `Complex`, и пользуясь лишь предоставленным этим модулем интерфейсом, мы легко можем разработать программу, использующую комплексные числа. Например,

λ
```
module Main where
import Complex

c1 = 2 :+ 3
x1 = realPart c1
y1 =  imagPart c1

c2 = conjugate c1
c3 = mkPolar (sqrt 2) (pi/4)

p1 = polar ((-1) :+ 0)
```

Вычислим значения величин определенных в данном модуле:

H
```
---> c1
2.0 :+ 3.0
---> x1
2.0
---> y1
3.0
---> c2
2.0 :+ (-3.0)
---> c3
1.0 :+ 1.0
---> p1
(1.0,3.14159)
```

◀

Вопросы и задания

VII.5.1

Множество является базовой структурой, которая лежит в основании всей математики. При разработке алгоритмов множества используются как основа для многих важных абстрактных типов данных (например, такого как словарь).

Множеством называется совокупность элементов, каждый элемент которого или сам является множеством или является примитивным элементов, называемым атомом. Все элементы любого множества *различны*, так как в множестве не может содержаться двух копий одного и того же элемента.

Предложите несколько реализаций этого АТД. В таблице перечислены операторы, выполняемые над множествами, которые часто включают в реализацию.

Метод	Назначение
`union`	Объединение двух множеств
`intersection`	Пересечение двух множеств
`difference`	Разность множеств
`merge`	Объединение непересекающихся множеств (слияние)
`member`	Возвращает `True`, если элемент принадлежит множеству
`makeNull`	Создает пустое множество
`insert`	Добавляет элемент в множество (если элемент уже присутствует в множестве, то множество не изменяется)
`delete`	Удаляет элемент из множества (если элемента нет в множестве, то множество не меняется)
`equal`	Возвращает `True`, если два множества состоят из одних и тех же элементов

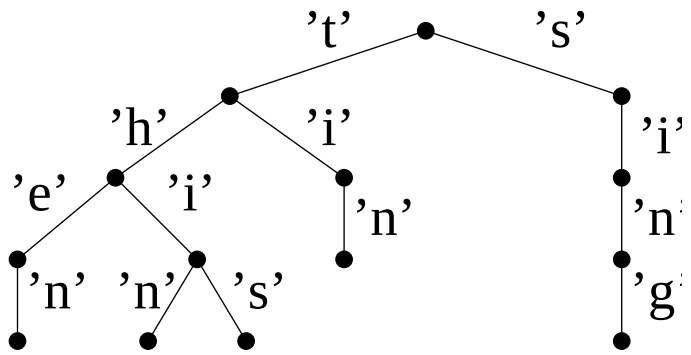
VII.5.2

Еще одним широко распространенным абстрактным типом данных является *очередь* `Queue`. Она реализует структуру вида FIFO (First In, First Out — первым пришел, первым вышел). Примеры очередей встречаются очень часто как в окружающей нас жизни, так и в программировании. Очередь можно рассматривать как последовательность объектов. Объекты становятся только в конец

очереди, а обслуживаются и удаляются только из ее начала. Обычно над очередью выполняется три операции. Операция постановки в очередь, часто называемая «ввести» (enqueue). Операция извлечения из очереди, именуемая «вывести» (dequeue). Значение первого объекта получается с помощью операции «первый» (front). Предложите несколько реализаций данного АТД и напишите модуль, использующий очередь. Добавьте к интерфейсу также функции `makeNull`, очищающую очередь, делая ее пустой, и `isEmpty` — возвращающую значение `True` тогда и только тогда, когда очередь пуста.

VII.5.3

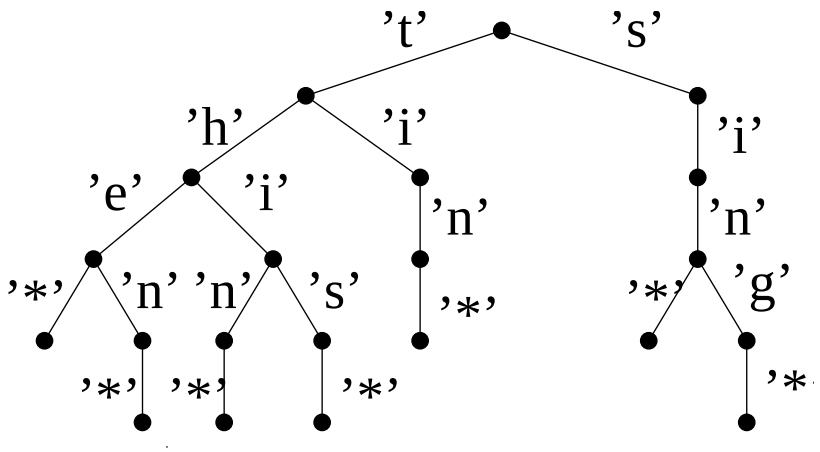
Рассмотрим структуру, предназначенную для представления множеств, состоящих из символьных строк, и называемую «нагруженное дерево» (trie). [2] Структура нагруженного дерева поддерживает операторы множеств, у которых элементы являются словами, т. е. символьными строками.



В нагруженном дереве каждый путь от корня к листу соответствует одному слову из множества. При таком подходе узлы дерева соответствуют префиксам слов. На рисунке представлено нагруженное дерево для слов the, then, thin, this, tin, sin, sing:

Чтобы избежать проблем, связанных с совпадением префиксов и слов, подобных наличию слова "the"и префикса "the"у слова "then", введем специальный символ `*` — маркер конца, указывающий окончание любого слова. Только тогда слова будут словами, а не префиксами. Наше дерево теперь выглядит так:

[2] Англоязычное название этого типа trie (произносится как “try”) образовано средними буквами слова re*trie*val (поиск, выборка, возврат). Устоявшегося термина для этой структуры в русской литературе пока нет.



Реализуйте АТД Trie и набор методов для работы с ней: г

Метод	**Назначение**
`add`	Добавляет новый элемент в дерево
`isMember`	Возвращает значение `True`, если элемент содержится в структуре
`remove`	Удаляет элемент из дерева
`isEmpty`	Проверяет пусто ли дерево
`emptyTrie`	Создает пустое дерево
`showTrie`	Выдает список всех слов, содержащихся в дереве

VII.5.4

Предложите другую реализацию нагруженного дерева, хранящую в узлах кортеж типа `(Bool, Char)`. Первый элемент кортежа принимает значение `True`, если узел является окончанием слова, и `False` в противном случае.

VII.5.5

Напишите программу, использующую АТД Trie из предыдущих упражнений, предназначенную для проверки синтаксиса строк текста. Строка состоит из слов, разделенных пробелами. Слово считается написанным правильно, если оно есть в словаре, реализованном с помощью нагруженного дерева.

Глава VIII
Ввод и вывод информации

1. Интерактивный ввод и вывод

Ввод и вывод контролируются операционной системой. Haskell общается с операционной системой для выполнения операций ввода-вывода, используя набор внутренних функций, которые оперируют величинами `IO` типа. Haskell интерпретирует величины этого типа, обращаясь с запросами о вводе и выводе к операционной системе.

Так, следующий скрипт использует встроенную функцию `putStr` для вывода строки "Привет, мир!" на экран.

λ
```
main = putStr "Привет, мир!\n"
```

По соглашению, скрипт на языке Haskell, который осуществляет операции ввода-вывода, должен содержать определение переменной `main` в своем главном модуле. Во время ее вычисления Haskell использует операционную систему для проведения операций ввода-вывода.

H
```
---> main
Привет, мир!
```

Именно эта величина и является результатом выполнения откомпилированной программы на Haskell. До сих пор мы пользовались только интерпретатором Hugs для получения результатов программ. Для запуска компилятора и получения выполняемого файла, содержащего программу на языке Haskell, следует сначала откомпилировать модуль с помощью, например, компилятора `ghc`, а затем выполнить полученный откомпилированный файл, называемый по умолчанию `a.out`:

```
$ ghc  Hello.hs
$ ./a.out
Привет, мир!
```

Здесь символ `$` есть приглашение к работе, выводимое окном shell.

При компиляции можно указать имя выходного файла с помощью ключа `-o`:

```
$ ghc  Hello.hs -o hello
$ ./hello
Привет, мир!
```

Команды ввода данных дают возможность ассоциировать данные, введенные с клавиатуры, с переменной в программе на Haskell, после чего эта переменная может быть использована в командах вывода. Скрипт может содержать несколько входных и выходных директив. В Haskell любая последовательность команд ввода-вывода определяется `do`-выражением.

Рассмотрим скрипт, запрашивающий с клавиатуры имя пользователя, после чего оно используется в команде вывода.

```
main =                                                    λ
  do
    putStr "Введите Ваше имя, пожалуйста:\n"
    name <- getLine
    putStr ("Здравствуйте, " ++ name ++ ".\n" ++
            "Сегодня замечательный день! (:-)\n")
```

Подобный скрипт можно откомпилировать и выполнить полученный файл:

```
$ ghc enterName.hs -o enterName
$ ./enterName
Введите Ваше имя, пожалуйста:
Иван
Здравствуйте, Иван.
Сегодня замечательный день! (:-)
```

Конечно, можно воспользоваться и интерпретатором:

```
---> main                                                 λ
Введите Ваше имя, пожалуйста:
Иван
Здравствуйте, Иван.
Сегодня замечательный день! (:-)
```

Первая строка, напечатанная в процессе выполнения функции `main`, является результатом выполнения операции вывода, вторая — отражением операционной системой ввода, произведенного с клавиатуры, третья и четвертая снова являются результатом выполнения операций вывода.

За ключевым словом `do` должна следовать последовательность операций ввода-вывода. В этом примере их было три. Первая из них вывела строку "Введите Ваше имя, пожалуйста:\n"на экран. Так как в конце строки добавлен символ "\n", то после окончания вывода курсор переместился в начало следующей строки. Вторая команда (`name <- getLine`) читает строку, введенную с клавиатуры и ассоциирует последовательность полученных символов с переменной, указанной перед стрелкой (`<-`). В нашем примере переменная носит имя `name`.

Любая *следующая* директива `do`-выражения может оперировать этой переменной, но она *не видна вне* `do`-выражения.

И, наконец, третья команда посылает на экран последовательность символов, включающую два символа перехода на новую строку (`'\n'`).

При использовании конструкции `do` следует помнить о двумерном синтаксисе языка: нужно внимательно следить за глубиной отступа в строках, являющихся продолжением `do`-выражения. Здесь применяются те же правила, что и в конструкциях `let` и `where`.

Последовательность директив ввода-вывода может, конечно, содержать несколько шагов. Так в следующем скрипте с клавиатуры запрашиваются две величины, затем осуществляется вывод, после чего запрашивается сигнал подтверждающий окончание ввода:

λ
```
main =
  do
    putStr "Введите Ваше имя, пожалуйста:\n"
    name <- getLine
    putStr "Введите Ваш e-mail адрес::\n"
    adress <- getLine
    putStr (unlines [
              "Здравствуйте, " ++ name ++ ".",
              "Мы вышлем вам сообщение на адрес "
              ++ adress ++ ".",
              "Нажмите Enter для завершения работы"])

    signOff <- getLine
    return()
```

Обмен сообщениями во время диалога с функцией `main` может выглядеть так:

H
```
---> main
```

```
Введите Ваше имя, пожалуйста:
Иван
Введите Ваш e-mail адрес::
pypkin@qqq.com
Здравствуйте, Иван.
Мы вышлем вам сообщение на адрес pypkin@qqq.com.
Нажмите Enter для подтверждения согласия
```

Пустая строка, появившаяся после последней напечатанной программой фразы, является отображением нажатия клавиши `Enter`.

Используемая в скрипте функция `unlines` определена в прелюдии и формирует одну строку из списка строк, вставляя между ними символ перевода на новую строку:

```
---> unlines ["line1", "line2", "line3"]                    Н
"line1\nline2\nline3\n"
```

При выводе удобно использовать функцию `putStrLn`, которая после печати строки добавляет символ `'\n'` для перевода курсора на новую строку:

```
main = do putStr "Введите что-нибудь: "                     λ
          str <- getLine
          putStrLn ("Вы ввели: " ++ str)
```

```
---> main                                                   Н
Введите что-нибудь: 2+3
Вы ввели: 2+3
```

Haskell дает возможность ввода или вывода одного единственного символа, для чего используются функции `getChar` и `putChar`:

```
e1 = do c <- getChar                                        λ
        putChar c
```

При вызове функции `e1` введенный символ отобразится дважды: сначала как эхо клавиатурного ввода, затем как результат работы функции `putChar`.

```
---> e1                                                     Н
ww
```

Рассмотренная выше функция `putStrLn` определена в прелюдии с помощью функции `putChar`:

```
putStrLn  :: String -> IO ()
putStrLn s = do putStr s
                putChar '\n'
```

Итак, мы можем выполнять некоторые действия и использовать полученный результат, но как нам вернуть величину, полученную из последовательности действий? К примеру, пусть функция `ready`, считывает символ и возвращает `True`, если введенный символ равен `'y'`:

λ

```
ready = do c <- getChar
           c == 'y'
```

Определенная таким образом функция не будет работать — интерпретатор выдаст ошибку несоответствия типа, так как выражение `c == 'y'` имеет тип `Bool`, в то время как наличие конструкции `do` требует, чтобы результат имел тип `IO`, т. е. являлся бы действием. В подобной ситуации требуется создать величину типа `IO`, берущую логическую величину, и ничего не делающую, кроме возвращения результата логического типа.

Для решения проблемы воспользуемся функцией `return`, включенной в прелюдию и имеющей тип `a -> IO a`. Отметим, что функция `getLine`, используемая нами в предыдущих примерах, определена в прелюдии также с помощью функции `return`:

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

Обратите внимание на второе утверждение `do`, входящее в ветвь `else`. Каждое `do`-утверждение является единственной цепочкой утверждений. Любая внедряющаяся в нее конструкция, такая как `if`, должна использовать новое `do`-утверждение для инициации дальнейшей последовательности действий.

Функция `return` берет некую величину и превращает ее в действие, связанное с вводом-выводом данных. Каким же образом осуществить обратное преобразование? Можно ли использовать величины типа `IO` в обычных выражениях? Ответ на последний вопрос — отрицательный. Нельзя смешивать действия и величины в одном выражении.

▶ *Упражнение VIII.1.1*

Определите функцию `ready` с помощью `return`. Для печати результата примените функцию `print` (также содержащуюся в файле `Prelude.hs`), которая преобразует свой аргумент в строку и печатает ее, добавляя символ `'\n'` в конце.

```
ready = do c <- getChar
           return (c == 'y')

e2 = do r <- ready
        print r
```

λ

Теперь мы можем увидеть результаты работы функции `ready` (`True`, если введен символ `'y'` и `False` в противном случае):

```
---> e2
dFalse

---> e2
yTrue
```

H

Отметим, что функцию `ready` нельзя использовать аналогично обычным логическим величинам (например, включить в логическое выражение), так как ее тип — `IO Bool`:

```
---> ready && True
ERROR - Type error in application
*** Expression     : ready && True
*** Term           : ready
*** Type           : IO Bool
*** Does not match : Bool
---> print (ready)
<<IO action>>
---> :type ready
ready :: IO Bool
```

H

◀

Функция `print` выводит на печать свой аргумент, если он отличен от типа `IO t` для любого `t`.

```
---> print True
True
---> print"Hello"
"Hello"
---> print 3
3
```

H

```
---> print(putChar 'p')
<<IO action>>
```

Величины типа `IO` разделяют мир программ, написанных на языке Haskell, на величины и действия. В предыдущих разделах книги мы использовали только величины, но теперь у нас появилась возможность выполнять те или иные действия. Hugs дает нам информацию о типе выражения печатая его. Определим список, содержащий три величины типа `IO`, после чего выведем его:

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

H
```
---> todoList
[<<IO action>>,<<IO action>>,<<IO action>>]
```

Каким же образом можно выполнить действия, заданные в списке `todoList`? Включенная в прелюдию функция `sequence_` позволяет выполнять действия, заданные величинами типа `IO ()`. Если ее применить величинам, аналогичным списку `todoList`, то действия будут выполнены:

H
```
---> sequence_ todoList
abc!!
```

Здесь сначала были напечатаны символы 'a', 'b', 'c', после чего ожидался ввод еще одного символа. В данном примере был введен символ '!', «эхо» которого было напечатано при вводе символа, а затем было выполнено действие `putChar c` — печать введенного символа.

2. Хранение информации в файлах

Мы научились писать программы, осуществляющие интерактивное взаимодействие с пользователем: информация от пользователя передается посредством ввода данных с помощью клавиатуры, а информация для пользователя выводится на экран. Однако информация, отображаемая на экране, слишком изменчива и не может хранится на нем очень долго — она скоро «затирается» другой информацией. Компьютерные системы предлагают

способ сохранения информации в течении продолжительного времени, основанный на использовании файловой системы. Взаимодействуя с файлами, программа может получить информацию из файла, как только она потребуется. Интерактивно взаимодействуя с файловой системой, программа может воспользоваться информацией из файла, созданного ранее либо ею самой, либо другим программным средством.

Предположим, что требуется написать программу на Haskell, которая будет записывать в файл строку текста, введенную с клавиатуры. Программа может выглядеть так:

```
main =                                                    λ
  do
     putStr (unlines["Введите одну строку:"])
     lineFromKeyboard <- getLine
     writeFile filename lineFromKeyboard
     putStr ("Введенная строка записана в файл\""++
             filename ++ "\"")
  where
     filename = "oneLiner.txt"
```

Запись в файл осуществляется посредством команды вывода, называемой `writeFile`. Первый аргумент этой функции — строка, содержащая имя создаваемого файла, а второй — строка, записываемая в него. В данном случае выводимая строка содержала всего один символ окончания строки, но в общем случае их может быть несколько. Следующий скрипт создает файл, содержимое которого разделено на три строки текста:

```
main = writeFile "books.dat" (unlines books)              λ
  where
  books = [
   "А.С. Пушкин,  \t \"Сказка о рыбаке и рыбке\"",
   "Л.Н. Толстой, \t \"Война и мир\", ч.1",
   "В.А. Зорич,   \t \"Математический анализ\", т.2"
    ]
```

Ниже показана работа этой программы. Для получения выполняемого кода использовался компилятор `ghc`. Команда `ls -l` выдает информацию о размере файла и правах доступа к нему. Мы видим, что файл `writeBooks`, полученный в результате компиляции, является исполняемым (в правах доступа содержится символ `x`). Как уже отмечалось, откомпилированная программа при ее вызове вычисляет функцию `main`, которая в данной программе создает

файл с количеством строк, равным длине списка `books`. Команда `cat` выводит содержимое файла построчно на экран.

```
$ ghc writeBooks.hs -o writeBooks
$ ls -l writeBooks
-rwxrwxr-x  1 test test 370558 Ноя 18 17:40 writeBooks
$ ./writeBooks
$ cat books.dat
А.С. Пушкин,     "Сказка о рыбаке и рыбке"
Л.Н. Толстой,    "Война и мир", ч.1
В.А. Зорич,      "Математический анализ", т.2
$
```

Итак, команда `writeFile` создает текстовый файл. Команда `readFile` осуществляет обратную операцию — считывает данные из существующего текстового файла.

▶ *Упражнение VIII.2.1*

Напишем программу, запрашивающую имена входного и выходного файлов и копирующую тест, содержащийся в первом, во второй.

λ
```
main
    = do
          putStr "Укажите имя входного файла: "
          ifile <- getLine
          putStr "Укажите имя выходного файла: "
          ofile <- getLine
          s <- readFile ifile
          writeFile ofile s
          putStr "Файл скопирован\n"
```

Интерактивный процесс взаимодействия пользователя и программы может выглядеть так:

H
```
---> main
Укажите имя входного файла: test1.txt
Укажите имя выходного файла: test2.txt
Файл скопирован

--->
```

Полученный выходной файл идентичен входному (воспользуйтесь, например, командой `diff` для проверки этого утверждения).

Изменим программу, добавив в нее фильтр, копирующий из входного файла только буквы и цифры:

λ
```
main
    = do
          putStr "Укажите имя входного файла: "
          ifile <- getLine
          putStr "Укажите имя выходного файла: "
          ofile <- getLine
          s <- readFile ifile
          writeFile ofile  (filter isAlphaNum s)
          putStr "Файл отфильтрован\n"
```

Теперь при копировании, например, файла, содержащего следующие три строки

```
Строка 1.
Строка 2.
Строка 3.
```

мы получим однострочный файл

```
Строка1Строка2Строка3
```

◀

Вопросы и задания

VIII.2.1

Каков будет результат выполнения команды `main`, определенной следующим образом

λ
```
main =
  do  putStr ("Введите ваши фамилию и имя " ++
              "(например, Петров Иван):")
      firstLast <- getLine
      putStr (reverse firstLast)
```

а) Сначала будет выведено имя, а затем фамилия.
б) Будет выведено только имя, а фамилия проигнорирована.
в) Будет выведено имя, но в обратном порядке следования букв.
г) Будет выведено все, что ввели, но в обратном порядке следования букв.

VIII.2.2

Как должна выглядеть команда ввода-вывода в предыдущем примере, чтобы была напечатана одна фамилия (без имени):

а) `putStr (take 1 firstLast)`
б) `putStr (drop 1 firstLast)`
в) `putStr (takeWhile (/=' ') firstLast)`
г) `putStr (dropWhile (/=' ') firstLast)`

3. Графический интерфейс (GUI)

Работа в современных операционных системах немыслима без использования различных оконных систем, предоставляющих пользователю удобные средства взаимодействия с программами и доступа к ресурсам системы (например, файлам). Организация такого диалога с пользователем осуществляется с помощью **графического интерфейса пользователя** (Graphical User Interface, GUI), позволяющего управлять приложением с помощью мыши или путем ввода информации в те или иные поля окон. Подобные графические интерфейсы могут содержать кнопки, переключатели, различные полосы прокрутки, выпадающие меню и другие элементы управления.

В современном программировании также ярко выражена тенденция к написанию программ, интерактивно взаимодействующих с пользователем через графический интерфейс. Такие программы в общем случае являются более гибкими, чем программы с текстовым интерфейсом. Традиционно, GUI-программирование доминировало в императивных языках программирования.

Учитывая важность графического интерфейса для языков функционального программирования сразу несколько групп программистов разрабатывают для них средства создания GUI.

Некоторые из этих сред разработки графического интерфейса основываются на общих абстракциях, заложенных в императивные среды разработки GUI, и терпят неудачу при попытке найти свое будущее в функциональном программировании. Другие же разработаны в функциональном стиле, что позволяет использовать их, оставаясь в рамках функциональной парадигмы программирования.

Перечислим программные средства, позволяющие разрабатывать графический интерфейс для программ, написанных на функциональных языках:

- Fudgets
- FranTk
- TclHaskell
- FRUIT
- Gtk2HS
- Haggis
- Clean I/O
- Gtk+HS

Особо отметим Fudgets, FranTk, Haggis, Clean I/O, которые являются представителями функциональных GUI. В нашу задачу не входить обучение принципам разработки графического интерфейса, мы лишь продемонстрируем их возможности на базе одного из них, а именно `FranTk`.

FranTk (произносится как «франтик») — одно из средств разработки GUI для Haskell. Оно использует концепцию поведения и событий, что позволяет моделировать работу системы в течении длительного времени. К дополнительным преимуществам FranTk можно отнести его независимость от платформы: с его помощью можно создавать программы с оконным интерфейсом, функционирующие как в Linux, так и в Winows.

События (events) используются для описания величин, которые происходят дискретно, например, нажатие кнопки. *Поведение* (behaviors) — есть непрерывная величина в течение продолжительного времени. Величины такого рода используются для представления состояния приложения. События и поведения могут быть интерактивными. Например, нам может потребоваться поведение, которое изменяется в зависимости от некоторого события.

По сути, FranTk есть применение широко известных средств Tcl/Tk. Ниже приведен код программы на языке Haskell, которая создает окно, содержащее две кнопки и полосу прокрутки (scrollbar). При нажатии на соответствующую кнопку значение счетчика увеличивается, либо уменьшается на единицу:

```
module Main where
import FranTk

main :: IO ()
main =  display $ withRootWindow [title "Счетчик"]
```

```
                    $ scaleAndButton

scaleAndButton  :: Component
scaleAndButton  = do { m <- mkBVar 0; composite m}

composite, scale, counterB, lbl, incr, decr :: BVar Int
    -> Component

composite m = above (counterB m) (scale m)

scale m = mkHScale [scaleValB (bvarBehavior m)]
    (bvarInput m)

counterB m = above (lbl m) (beside (incr m) (decr m))

lbl m = mkLabel [textB (lift1 show (bvarBehavior m))]

incr m = mkButton [text " +1 "] (tellL (bvarUpdInput m)
    (+1))
decr m = mkButton [text " -1 "] (tellL (bvarUpdInput m)
    (subtract 1))
```

Для компицяции выполним команду

```
ghc  -O -package FranTk -o exampleFranTk exampleFranTk.hs
```

В результате компиляции получается выполняемый файл `exampleFranTk`. После запуска программы появляется окно, позволяющее либо нажатием на кнопку, либо передвижением бегунка на полосе прокрутки менять значение счетчика.

Более подробную информацию о возможностях `FranTk` и других средств, предназначенных для создания графического интерфейса, можно найти в сети интернет по следующим адресам:

```
http://www.cs.chalmers.se/Fudgets/Manual/
http://www.dcs.gla.ac.uk/~meurig
http://www.dcs.gla.ac.uk/fp/software/haggis/
       haggis-doc/users_toc.html
```

Литература

[1] Бадд Т. *Объектно-ориентированное программирование в действии* — СПб: Питер, 1997.

[2] Бен-Ари М. *Языки программирования. Практический сравнительный анализ* — М.: Мир, 2000.

[3] Брукшир, Дж., Глен. *Введение в компьютерные науки. Общий обзор, 6-е издание* — М.: Издательский дом «Вильямс», 2001.

[4] Кнут, Дональд, Эрвин *Искусство программирования, том 2. Получисленные алгоритмы, 3-е изд.: Уч. пос.* — М.: Издательский дом «Вильямс», 2000.

[5] Филд А., Харрисон П. *Функциональное программирование* — М.: Мир, 1993.

[6] `http://haskell.org` — Официальный сайт языка Haskell.

Предметный указатель