Душкин Р. В.

Справочник по языку Haskell

ДМК Пресс
издательство
Москва, 2008

УДК 004.4
ББК 32.973.26-018.2
Д86

Душкин Р. В.
Д86 Справочник по языку Haskell. М.: ДМК Пресс, 2008. 544 с., ил.
ISBN 5-94074-410-9

Данная книга является первой книгой на русском языке, описывающей набор стандартных библиотек функционального языка программирования Haskell. В первой хасти книги кратко рассматривается синтаксис языка и способы его применения для решения задач. Во второй части описываются стандартные библиотеки языка, входящие в поставки всех современных трансляторов Haskell (GHC, HUGS и др.).

Книга станет прекрасным подспорьем для программистов, занимающихся прикладным программированием на языке Haskell, а также для студентов, изучающих функциональное программирование.

УДК 004.4
ББК 32.973.26-018.2

ISBN 5-94074-410-9

Оглавление

Введение **9**

I. Синтаксис и идиомы языка 11

1. Функции **12**

1.1. Общий вид определения функций 12

1.1.1. Детальный разбор нескольких примеров определения функций 13

1.1.2. Ветвление 16

1.1.3. Замыкания 17

1.1.4. Бинарные операции 20

1.2. Технология сопоставления с образцами 23

1.2.1. Образцы вида $(\mathbf{n} + \mathbf{k})$ 25

1.2.2. Именованные образцы 26

1.2.3. Ленивые образцы 27

1.3. Ввод и вывод 28

1.3.1. Действия ввода/вывода 28

1.3.2. Обработка исключений 32

1.4. Приёмы программирования 34

1.4.1. Двумерный синтаксис 34

1.4.2. Рекурсия и корекурсия 35

1.4.3. Накапливающий параметр и хвостовая рекурсия 39

1.4.4. Бесточечная нотация 41

1.4.5. Анонимные функции 42

1.4.6. Охрана 44
1.4.7. Определители списков 46

2. Типы данных **48**
2.1. Базовые типы 48
2.1.1. Кортежи 49
2.1.2. Списки 51
2.2. Кратко об алгебраических типах данных 53
2.2.1. Перечисления 54
2.2.2. Простые структуры 56
2.2.3. Именованные поля 59
2.3. Синонимы типов 61
2.4. Параметрический полиморфизм 63
2.5. Типы функций 64
2.5.1. Функции как программные сущности с типом 64
2.5.2. Каррирование и частичное применение 66
2.5.3. Функции высшего порядка 68

3. Классы типов и экземпляры классов **71**
3.1. Класс как интерфейс 71
3.2. Контекст и прикладные функции 76
3.3. Экземпляр — связь между типом и классом 78
3.3.1. Экземпляры класса `Logic` 80
3.4. Изоморфные типы 83
3.4.1. Определение нескольких экземпляров для уникальной пары (класс, тип) 85
3.5. Автоматическое построение экземпляров 86
3.6. Окончательные замечания о системе типов в языке Haskell 88

4. Модули **91**
4.1. Система модулей 91
4.1.1. Экспорт программных сущностей 92
4.1.2. Импорт сторонних модулей 93
4.2. Абстракция данных при помощи модулей 97
4.3. Кое-что ещё о модулях 98

5. Сводная информация **100**

II. Стандартные библиотеки 105

6. Стандартный модуль `Prelude` **108**
6.1. `Prelude`: Алгебраические типы данных 108
6.2. `Prelude`: Классы и их экземпляры 115
6.3. `Prelude`: Функции 125
6.4. `Prelude`: Операторы 170

7. Пакет модулей `Control` **172**
7.1. Модуль `Applicative` 172
7.2. Модуль `Arrow` 176
7.3. Модуль `Concurrent` 181
7.3.1. Модуль `Chan` 186
7.3.2. Модуль `MVar` 188
7.3.3. Модуль `QSem` 192
7.3.4. Модуль `QSemN` 193
7.3.5. Модуль `SampleVar` 194
7.4. Модуль `Exception` 197
7.5. Модуль `Monad` 211
7.5.1. Модуль `Fix` 220
7.5.2. Модуль `Instances` 222
7.5.3. Модуль `ST` 222
7.6. Модуль `Parallel` 224

8. Пакет модулей `Data` **226**
8.1. Модуль `Array` 226
8.1.1. Модуль `Base` 231
8.1.2. Модуль `Diff` 231
8.1.3. Модуль `IArray` 233
8.1.4. Модуль `IO` 234
8.1.5. Модуль `MArray` 237
8.1.6. Модуль `ST` 241
8.1.7. Модуль `Storable` 243

8.1.8. Модуль `Unboxed` . . . 245
8.2. Модуль `Bits` . . . 245
8.3. Модуль `Bool` . . . 247
8.4. Модуль `ByteString` . . . 248
8.4.1. Модуль `Base` . . . 277
8.4.2. Модуль `Char8` . . . 286
8.4.3. Модуль `Lazy` . . . 287
8.5. Модуль `Char` . . . 288
8.6. Модуль `Complex` . . . 298
8.7. Модуль `Dynamic` . . . 300
8.8. Модуль `Either` . . . 302
8.9. Модуль `Eq` . . . 303
8.10. Модуль `Fixed` . . . 304
8.11. Модуль `Foldable` . . . 305
8.12. Модуль `Graph` . . . 313
8.13. Модуль `HashTable` . . . 320
8.14. Модуль `Int` . . . 323
8.15. Модуль `IntMap` . . . 324
8.16. Модуль `IntSet` . . . 348
8.17. Модуль `IORef` . . . 360
8.18. Модуль `Ix` . . . 361
8.19. Модуль `List` . . . 362
8.20. Модуль `Map` . . . 374
8.21. Модуль `Maybe` . . . 383
8.22. Модуль `Monoid` . . . 385
8.23. Модуль `Ord` . . . 388
8.24. Модуль `Ratio` . . . 390
8.25. Модуль `Sequence` . . . 390
8.26. Модуль `Set` . . . 396
8.27. Модуль `STRef` . . . 401
8.27.1. Модуль `Lazy` . . . 402
8.27.2. Модуль `Strict` . . . 402
8.28. Модуль `Traversable` . . . 402
8.29. Модуль `Tree` . . . 404
8.30. Модуль `Tuple` . . . 408

8.31. Модуль Typeable 408
8.32. Модуль Unique 414
8.33. Модуль Version 415
8.34. Модуль Word 416

9. Пакет модулей Debug **419**
9.1. Модуль Trace 419

10. Пакет модулей Foreign **421**
10.1. Модуль C 422
10.1.1. Модуль Error 423
10.1.2. Модуль String 429
10.1.3. Модуль Types 436
10.2. Модуль ForeignPtr 439
10.3. Модуль Marshal 444
10.3.1. Модуль Alloc 445
10.3.2. Модуль Array 447
10.3.3. Модуль Error 453
10.3.4. Модуль Pool 455
10.3.5. Модуль Utils 459
10.4. Модуль Ptr 462
10.5. Модуль StablePtr 466
10.6. Модуль Storable 468

11. Пакет модулей System **471**
11.1. Модуль Cmd 471
11.2. Модуль CPUTime 472
11.3. Модуль Directory 473
11.3.1. Модуль Internals 482
11.4. Модуль Environment 482
11.5. Модуль Exit 484
11.6. Модуль Info 485
11.7. Модуль IO 486
11.7.1. Модуль Error 497
11.7.2. Модуль Unsafe 507
11.8. Модуль Locale 507

11.9. Модуль `Mem` 510
11.9.1. Модуль `StableName` 510
11.9.2. Модуль `Weak` 512
11.10Модуль `Random` 515
11.11Модуль `Time` 519

12.Пакет модулей `Text` **528**
12.1. Модуль `Printf` 528
12.2. Модуль `Read` 531
12.2.1. Модуль `Lex` 533
12.3. Модуль `Show` 535
12.3.1. Модуль `Functions` 536

Заключение **537**

Литература **538**

Введение

Язык Haskell является динамично развивающимся функциональным языком программирования, который получает всё больше и больше сторонников во всём мире, в том числе и в России. Этот язык вырвался из рамок научных лабораторий и стал языком программирования общего назначения. Вместе с тем хорошей литературы об этом прекрасном языке программирования категорически мало, тем более на русском языке.

В конце 2006 года из печати вышла первая и на текущий момент (2007 год) единственная книга на русском языке, рассматривающая функциональное программирование на языке Haskell [1]. Несмотря на то что в этой книге тема языка Haskell раскрыта практически полностью, его описание в ней страдает неполнотой и некоторой «поверхностностью». С другой стороны, достаточно серьёзная математика в книге немного отпугивает неподготовленного читателя. Поэтому книга явилась своеобразным «первым блином», который необходим для первоначального ввода в проблематику. Однако в связи с ростом популярности как языка Haskell, так и парадигмы функционального программирования, необходимо больше всевозможных материалов, охватывающих различные аспекты и предназначенных для разной целевой аудитории.

Данная книга является кратким справочником по функциональному языку программирования Haskell стандарта Haskell-98 (без описания многочисленных расширений языка). В книге собрано описание знаний по успешному применению языка Haskell на практике. Она предназначена для тех, кто уже знает принципы функциональной парадигмы и сам язык Haskell. Это связано с тем, что, несмотря на то что практически всю информацию можно почерпнуть из интернета, очень часто необходимо иметь под рукой полноценный справочник, в котором мож-

но быстро найти ответы на специализированные вопросы. И эта книга как раз и предназначена для подобных целей.

Поскольку книга названа «кратким справочником», одним из принципов, которым руководствовался автор при её написании, является минимизация информации и предоставление компактно выраженных знаний, с достаточной степенью полноты раскрывающих смысл конструкций языка Haskell, идиом, существующих функций и других программных объектов, определённых в стандартных библиотеках. Поэтому стиль этого справочника является более или менее сухим и выдержанным, а описание программных сущностей наиболее формализованным.

Справочник разбит на две части. В первой части представлено краткое описание синтаксиса языка Haskell, а также наиболее часто и успешно используемые техники программирования на нём (ведь не секрет, что в каждом языке имеются свои особые методы «правильного» программирования). Во второй части описываются наиболее часто использующиеся стандартные модули, входящие в поставку двух наиболее известных трансляторов языка — HUGS 98 и GHC. Первая часть разбита на главы, каждая из которых описывает одну из пяти существующих в языке программных сущностей (и дополнительная шестая глава со сводной информацией). Главы второй части соответствуют стандартным библиотекам языка Haskell.

В целях единообразия представления информации в книге используется специальное форматирование текста, выделяющее определённые структурные элементы. Так, наименования программных сущностей выделяются моноширинным шрифтом обычного начертания: `head`, `True`, `Enum` и т. д. В отличие от идентификаторов ключевые слова записываются моноширинным шрифтом с подчёркиванием: **if**, **do**, **instance** и т. д. Знаки операций и специальные символы при записи ограничиваются круглыми скобками, чтобы выделить и отделить знаки от основного текста: (`+`), (`>=`), (‘) и т. д., в то время как сами скобки в случае необходимости записываются в кавычках: «(», «)». Кроме того, наименования модулей, библиотек и специальных утилит также записываются моноширинным шрифтом: `Prelude`, `Data.List` и пр.

Краткость — сестра таланта, как говаривал русский классик А. П. Чехов. Поэтому осталось только упомянуть, что автор чрезвычайно благодарен Роганову В. А. за помощь в создании книги, и то, что автор будет рад получить комментарии и замечания по адресу электронной почты `darkus.14@gmail.com`.

Часть I.

Синтаксис и идиомы языка

Глава 1.

Функции

Функции являются базовым программным элементом в языке Haskell, при помощи которого строятся программы. Сложно (но можно) построить программу на этом языке, в которой не было бы определений функций. При помощи функций определяются вычислительные процессы, которые являются сутью создаваемой программы. Поэтому изучение способов определения функций является важнейшим делом в постижении языка Haskell.

1.1. Общий вид определения функций

С точки зрения функционального программирования и языка Haskell функция является программной сущностью верхнего уровня, при этом программа может состоять только из набора этих сущностей. Остальные программные сущности языка Haskell (типы данных, классы и их экземпляры, модули) могут присутствовать, а могут и отсутствовать в программах, но функции присутствуют всегда. Поэтому определение функций является главным при программировании на языке Haskell.

Как уже сказано, каждое определение функции является декларацией верхнего уровня. В определение может входить необязательная сигнатура, то есть описание типа функции, а также собственно описание того, что функция возвращает на основании входных параметров. Тем самым функция в языке Haskell является отражением математического понятия «функция», которое определе-

но как некоторая взаимосвязь между входными параметрами (аргументами) и выходным значением (результатом).

Любые вычислительные процессы в языке Haskell описываются при помощи функций. Такое описание происходит в виде вызовов других функций или самой себя, ветвления в зависимости от какого-либо значения, либо определения локальных функций, чья зона видимости ограничивается описываемым вычислительным процессом. В качестве примера определения функции можно привести следующую запись:

```
repeat :: a -> [a]
repeat x = xs
  where
    xs = x:xs
```

Это определение функции `repeat` из стандартного модуля `Prelude`, которая строит бесконечный список из переданного ей на вход значения (подробно эта функция описана на стр. 158). Список состоит из элементов, каждый из которых равен входному значению. В связи с тем, что язык Haskell является ленивым, в нём вполне возможны такие объекты, как бесконечные списки.

Данное определение читается достаточно просто. Первая строка как раз является сигнатурой, которая описывает тип функции (подробное описание типов функций приводится в разделе 2.5.). Запись в этом примере обозначает, что функция `repeat` получает на вход ровно один аргумент некоторого типа `a`, а возвращает значение типа `[a]`, то есть список значений типа `a`. Вторая строка определяет способ вычисления результата, возвращаемого функцией. Здесь используется локальное определение («замыкание») `xs`, которое описывается ниже после ключевого слова **where**. Само замыкание `xs` равно входному параметру `x`, который при помощи конструктора списков `(:)` добавляется к тому же `xs`. То есть здесь используется рекурсия, которая никогда не остановится, чем и достигается бесконечность получаемого результата.

1.1.1. Детальный разбор нескольких примеров определения функций

Каждое определение функции (не сигнатура) состоит из набора так называемых клозов, то есть отдельных вариантов определения функции, которые зависят от вида входных параметров, которые называются образцами и разделены

пробелами. Более детально образцы и клозы описываются чуть ниже (см. раздел 1.2.). Здесь же детально описывается несколько примеров определения различных функций, которые взяты всё из того же стандартного модуля `Prelude`.

Определение любой функции может состоять из одного клоза. Следующим образом, к примеру, определены функции для работы с парами, являющими собой кортежи из двух элементов (подробно кортежи описываются в главе 2.):

```
fst (x, _) = x


snd (_, y) = y
```

Как видно, обе функции принимают на вход один параметр, который является парой: оба значения пары заключены в скобки и разделены запятой. Первая функция возвращает первый элемент пары, вторая — второй соответственно. Формальный входной параметр функций записан в виде образца, в котором используется маска подстановки (`_`) вместо тех параметров, которые не используются в теле функции. Так, функция `fst` оперирует только первым элементом пары, поэтому второй можно заменить маской (`_`). Впрочем, ничто не мешает пользоваться и полноценными образцами:

```
fst (x, y) = x
```

Такое определение полностью тождественно тому, что было приведено ранее. Однако хорошим тоном является замена неиспользующихся образцов именно символом (`_`). В разделе 1.2. маска подстановки описана более подробно. Сами приведённые функции подробно описываются на стр. 136 и стр. 163 соответственно.

Различных образцов в клозе функции может быть несколько. Их количество может соответствовать числу формальных параметров функции или быть меньше, но не больше (большее количество образцов просто обозначает, что функция принимает на вход большее число параметров, при этом в сигнатуре функции, если она приводится, должен быть описан тип каждого входного параметра). Вот примеры функций с двумя и тремя образцами (эти функции детально рассматриваются на стр. 129 и стр. 133 соответственно):

```
const k _ = k
```

```
flip f x y = f y x
```

Все рассмотренные примеры имеют в определении один клоз, то есть одно выражение, определяющее значение функции. В функциональном программировании принято, что функция может быть определена несколькими клозами, каждый из которых характеризуется определённым набором образцов. Например, вот функции для оперирования со списками — их определения состоят из двух клозов:

```
last [x]    = x
last (_:xs) = last xs


init [x]    = []
init (x:xs) = x : init xs
```

Первая функция возвращает последний элемент заданного списка (детально рассматривается на стр. 251), вторая — все начальные элементы списка, кроме последнего (детально рассматривается на стр. 137).

Использование нескольких клозов в определении функции является естественным способом ветвления алгоритма в функциональном программировании. При этом в разных языках функционального программирования используется различный способ обработки клозов. В языке Haskell весьма важен порядок клозов, так как транслятор просматривает набор клозов сверху вниз и выбирает среди них первый, чьи образцы подходят под фактические параметры функции, переданные ей на вход при вызове.

Так, в указанных выше функциях `last` и `init` на вход приходит некий список, в котором должен быть по крайней мере один элемент. В случае если список состоит из одного элемента, «срабатывает» первый клоз. Если список состоит из более чем одного элемента, транслятор пропускает первый клоз и выбирает второй. Если бы клозы в этих функциях стояли наоборот, то первый клоз срабатывал бы на любом непустом списке, в том числе и на таком, в котором содержится один элемент (поскольку на самом деле в нём содержится пара этого элемента и пустого списка). Поэтому программист всегда должен внимательно следить за порядком расположения клозов в языке Haskell.

1.1.2. Ветвление

Несколько клозов — не единственный способ организации ветвления вычислительного процесса в языке Haskell. В нём присутствуют и «традиционные» способы ветвления, а именно оператор **if** и оператор **case**.

Оператор if

Оператор **if** предназначен для ветвления вычислительного процесса в зависимости от условия булевского типа. Обычно этот оператор представляется в виде **if-then-else**, где после ключевого слова **if** идёт условие ветвления, после слова **then** следует выражение, которое выполняется в случае истинности условия; а после ключевого слова **else** находится выражение, которое выполняется в случае ложности условия. В языке Haskell обе части **then** и **else** обязательны при использовании оператора **if**, так как они определяют не действия в порядке некоторого вычисления, а функции, которые возвращают результат.

Выражения в обеих частях условного оператора `then` и `else` должны иметь один и тот же тип, который равен типу, возвращаемому функцией. Это очень важное условие использования этого ключевого слова. В качестве примера использования оператора **if** в языке Haskell можно привести функцию `until` из стандартного модуля `Prelude` (описывается на стр. 167):

```
until p f x = if p x
              then x
              else until p f (f x)
```

Эта функция предназначена для организации циклического применения функции `f` к аргументу, начальным значением которого является `x`. Цикл останавливается, когда предикат `p` становится равным `True` на очередном значении, которое возвратила функция `f`.

Оператор case

Оператор **case** предназначен для множественного ветвления, когда вычислительный процесс разбивается на несколько ветвей в зависимости от значения выражения произвольного типа. Оператор **if** является частным случаем оператора **case**, и, в принципе, любое ветвление можно было бы организовывать при помощи

оператора **case**. Оператор **if** вводится исключительно ради удобства и для поддержки традиционных идиом программирования.

В свою очередь оператор **case** сравнивает значение заданного выражения и выбирает из предложенных альтернатив такую, которая соответствует рассматриваемому значению. В языке Haskell синтаксис оператора **case** прост. Его можно рассмотреть на примере функции `scanl` из стандартного модуля `Prelude` (описывается на стр. 257):

```
scanl f q xs = q:(case xs of
                    []   -> []
                    x:xs -> scanl f (f q x) xs)
```

Как видно, после ключевого слова **case** идёт выражение, на основании значения которого производится ветвление. После выражения записывается ключевое слово **of**, вслед за которым идёт набор образцов, с которыми сопоставляется выражение оператора. Первый сверху образец, с которым успешно сопоставилось значение выражения, определяет ветвь ветвления, которая выбирается для вычисления. Это значит, что образцы в принципе могут определять и пересекающиеся множества значений, здесь технология выбора вычислительной альтернативы такая же, как и для клозов.

В процессе создания функций способы организации ветвления можно комбинировать друг с другом. Кроме того, все операторы ветвления являются полноценными выражениями, которые могут участвовать в вычислениях. Этим они отличаются от таких же операторов в императивном программировании. Каждый из операторов ветвления в языке Haskell возвращает значение определённого типа. Это надо помнить при программировании, поскольку значения, которые возвращены операторами ветвления, могут участвовать в вычислительных процессах наравне с прочими значениями. Это как раз и можно увидеть на примере функции `scanl`.

1.1.3. Замыкания

Замыкания или локальные определения — один из механизмов функционального программирования, который предназначен для оптимизации определений функций, и, насколько это возможно, выполнения некоторых последовательных действий (правда, это больше побочный эффект использования локальных определений, нежели запланированный разработчиками функциональных языков).

Замыкания позволяют вычислять некоторые значения внутри функций и перед вычислением самой функции, что обуславливает их использование исключительно внутри функций. Снаружи такие определения не видны.

Локальные определения являются функциями, чья область видимости ограничена верхней функцией, при этом в таких локальных определениях можно использовать всё то, что определено в функции, — образцы и даже прочие локальные определения (а их может быть, естественно, несколько). Из-за детерминизма, свойственного функциональному программированию, значение локальных определений вычисляется один раз, и оно не может быть изменено в рамках текущего вычислительного процесса. Это свойство и используется для оптимизации, поскольку локальным определением можно обозвать нечто в теле функции, что вычисляется несколько раз. Так как в любом случае при вычислениях будут получены одинаковые результаты, локальное определение позволяет выполнить вычисления единожды.

Локальные определения бывают двух видов: префиксные (они находятся перед самим вычислительных процессом) и постфиксные (они находятся после вычислительного процесса). Особой разницы между ними нет, за исключением того, что префиксные локальные определения являются выражениями.

Префиксное локальное определение — let

Ключевое слово **let** в совокупности со словом **in** используется для определения замыканий перед самим вычислительным процессом. При этом само определение локальных функций в данном случае является выражением, которое можно использовать в прочих выражениях. Пояснить это можно при помощи следующего примера (он выполняется в интерпретаторе языка Haskell, на что показывает символ приглашения интерпретатора (`>`) в начале строки):

```
> (let x = y * y; y = 5 in x / 5) + 5
```

В результате вычислений будет получено значение `10.0`, что и ожидалось. Если обратить внимание, то можно увидеть, что определение замыканий `x` и `y` находится внутри выражения, ограниченного скобками, которое далее участвует в выражении более высокого уровня. Проделать то же самое с постфиксным локальным определением не удастся, интерпретатор выведет сообщение о синтаксической ошибке.

Приведённый выше пример уже показал синтаксис префиксных локальных определений. Между ключевыми словами **let** и **in** располагается набор локальных определений в полном в соответствии с правилами определения функций. Внутри локальных определений можно пользоваться образцами главной функции, другими локальными определениями, собственными образцами. Все локальные определения должны быть отделены друг от друга символом (`;`) (если, конечно, не используется двумерный синтаксис). После ключевого слова **in** описывается основное определение, в котором можно пользоваться локальными.

В качестве полноценного примера функции с префиксным локальным определением можно привести функцию `lines` из стандартного модуля `Prelude`, которая разбивает заданный текст на строки по символу перевода строки. Её определение выглядит следующим образом (а подробное описание приведено на стр. 144):

```
lines "" = []
lines s  = let (l, s') = break ('\n' ==) s
           in  l : case s' of
                     []      -> []
                     (_:s'') -> lines s''
```

Стандартная функция `break` (детально описывается на стр. 262) принимает на вход предикат и строку, а возвращает пару из двух строк, являющихся подстроками входной строки. Их конкатенация как раз и равна входной строке, а точкой разделения является символ входной строки, на котором первым вернул истинное значение предикат. Этот символ относится ко второй подстроке в паре.

Как видно, в представленном определении функции `lines` локальное определение используется для разбиения входной строки на подстроки по символу перевода строки (`\n`). Такое разбиение выполняется первым, поскольку далее замыкания `l` и `s'` используются в вычислительном процессе. Локальное определение `s'` проверяется на пустоту, и если оно не равно пустому списку, то от него «отрывается» первый символ, который равен (`\n`), после чего процесс повторяется.

Постфиксное локальное определение — where

Можно определить замыкания после вычислительных процессов. Это делается при помощи ключевого слова **where**, после которого и перечисляются замы-

кания. Такой тип локальных определений ничем не отличается от префиксного, за исключением того, что не является выражением. Однако с эстетической точки зрения он более интересен, так как обычно локальным определениям дают осмысленные наименования, исходя из которых можно сразу понять их предназначение, а потому при использовании постфиксных определений можно сразу перейти к чтению кода основной функции, лишь позже обратившись к локальным определениям в случае надобности.

В качестве примера использования постфиксных локальных определений можно привести определение функции `gcd` из стандартного модуля `Prelude`, которая вычисляет наибольший общий делитель методом Евклида (детально описывается на стр. 136):

```
gcd 0 0 = error "gcd 0 0 is not defined."
gcd x y = gcd' (abs x) (abs y)
  where
    gcd' x 0 = x
    gcd' x y = gcd' y (x `rem` y)
```

Постфиксные локальные определения следуют всё тем же правилам определения функций, которые используются и для функций верхнего уровня.

В целом же можно отметить, что использование того или иного способа определения замыканий является вопросом предпочтения программиста. Можно даже использовать оба типа определений в одной функции, но при этом надо помнить, что из-за того, что префиксные объявления являются выражениями, их приоритет более высок перед постфиксными, поэтому если среди префиксных и постфиксных замыканий имеются одинаковые идентификаторы, то использоваться будет префиксный.

1.1.4. Бинарные операции

В языке Haskell для удобства программирования имеется возможность определять бинарные операции, назначая им имена в виде значков или их последовательностей. Собственно, все арифметические операции: (`*`), (`/`) и т. д. определены в стандартном модуле `Prelude` (хотя это и сделано через примитивные функции для базовых типов). Эта техника позволяет создавать функции, которые записываются между своими аргументами и имеют более традиционный внешний вид (с точки зрения математики).

В качестве имён бинарных операций можно пользоваться любыми последовательностями неалфавитных символов. Нельзя лишь называть операции так, как уже названы некоторые операции из стандартного модуля `Prelude` (хотя в случае необходимости можно отменить импорт соответствующих операций из этого модуля, что позволит их переопределить), ну и невозможно дать бинарным операциям имена, которые представляют собой зарезервированные для нужд языка последовательности символов (например, `(::)`, `(=>)` или `(->)`, которые используются в сигнатурах функций).

При определении бинарных операций используются круглые скобки `()`, в которых записывается наименование операции. В случае, если такая операция находится между своими операндами, то скобки необходимо опускать. Вот так, к примеру, определена операция конкатенации списков в стандартном модуле `Prelude` (операция `(++)` подробно описывается на стр. 171):

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

Как видно из этого определения, в сигнатуре бинарной операции используются круглые скобки, а в её определении — нет.

Определение приоритета и ассоциативности

Для бинарных операций возможно определение приоритета исполнения и ассоциативности. Для этих целей используется следующий набор ключевых слов: **infix**, **infixl** и **infixr**. Эти ключевые слова являются декларациями верхнего уровня, которые видны повсюду в модуле, где определяются бинарные операции.

Синтаксис определения приоритета и ассоциативности прост. На отдельной строке вначале идёт одно из перечисленных ключевых слов. Следующим термом после пробела записывается значение приоритета — целое число от 0 до 9. Чем выше число, тем выше приоритет операции. После числа через запятую перечисляются бинарные операции (просто наименования, без круглых скобок), которые имеют заданный приоритет и ассоциативность. Вот так, к примеру, определены приоритеты и ассоциативность бинарных операций в стандартном модуле `Prelude`:

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

Сами ключевые слова, как должно быть понятно, определяют ассоциативность. Слово **infix** регламентирует тот факт, что последовательное применение операций невозможно, ассоциативности нет в принципе. Действительно, для вышеуказанных операций с приоритетом 4 ((==), (/=) и т. д.), которые являются логическими операциями сравнения, записи вроде «`1 == 2 == 3`» бессмысленны. Для прочих же бинарных операций подобные записи вполне осмысленны, а потому ассоциативность определять необходимо.

Ключевые слова **infixl** и **infixr** определяет левую и правую ассоциативность соответственно. Например, для арифметических операций (+) и (-) определена левая ассоциативность, которая предполагает, что в записях вида «`1 + 2 + 3`» и «`3 - 2 - 1`» скобки будут восстанавливаться по ассоциативности влево: «`((1 + 2) + 3)`» и «`((3 - 2) - 1)`». Если для операции сложения (+) это не столь важно, то результат операции вычитания (-) очень зависит от типа ассоциативности.

Как уже сказано, значение приоритета может быть из интервала от 0 до 9. однако имеется ещё одно значение — 10, наивысший приоритет. Этот приоритет нельзя назначить ни одной операции, но его имеет операция применения функции к своим аргументам. Поэтому в выражениях с операциями можно использовать функции без заключения их в скобки.

Инфиксная нотация

В принципе, любая функция с двумя аргументами может быть записана между ними (так называемая «инфиксная нотация»). Для этого достаточно её имя

заключить в обратные апострофы: (‘). Обычно таким образом записываются функции целочисленного деления и получения остатка, поскольку для них нет устоявшихся математических знаков, а они соответствуют бинарным математическим операциям:

```
even n = (n ‘mod‘ 2) == 0
```

Эта функция является предикатом, который возвращает значение `True` на чётных величинах и `False` на нечётных (подробно описывается на стр. 119).

Функции с двумя аргументами, которые используются в качестве бинарных операций, могут также получить указание относительно их приоритета и типа ассоциативности. Для этого их имена в обратных апострофах (‘) необходимо перечислить в соответствующих декларациях **infix**, **infixl** или **infixr**.

1.2. Технология сопоставления с образцами

Образцы в функциональном программировании заменяют то, что в традиционном (императивном) программировании называется переменной. В языке Haskell переменных нет. Однако слово «образец» не является синонимом слова «переменная». Это даже не синоним словосочетания «формальный параметр», как это могло показаться после изучения приводимых до этого момента определений функций. Образцы являются частью технологии сопоставления с образцами, которая используется для выполнения вычислительных процессов внутри функций в части означивания формальных параметров и сравнения фактических величин (значений) с определёнными шаблонами.

Образцами могут быть не произвольные значения или наименования объектов (идентификаторы). Для того чтобы быть образцом, нечто должно подходить под определённые ограничения. Образцы — это выражения в произвольной форме, которые построены при помощи конструкторов данных. Переменные в образцах (то есть те части образцов, которые могут принимать разные значения) обозначаются строчными буквами, константы вводятся непосредственно. Главное ограничение — сопоставление с образцами во всех случаях должно происходить единственным образом. Другими словами, при сопоставлении некоторого фактического параметра с образцом все переменные образца должны означиваться однозначно, без двусмысленностей.

Особо следует рассмотреть образец (_). Этот образец облегчает работу транслятора языка Haskell, указывая ему, что фактическое значение использоваться не будет, а потому это значение просто не вычисляется в силу ленивости вычислительной модели языка. Более того, таких образцов в одном клозе может быть несколько, чего нельзя сказать о прочих переменных образцов. Любое имя переменной, которое используется в образцах, может быть задействовано только один раз во всём наборе образцов одного клоза. Нельзя написать что-то вроде:

```
someFunction x x = x + x
```

Такая запись вызовет сообщение о синтаксической ошибке: «повторяющийся идентификатор переменной в образцах». Но маску подстановки (_) можно использовать неограниченное количество раз.

Но ещё раз необходимо подчеркнуть, что образцы не являются наименованием фактических параметров функции (более того, они используются не только в определениях функций, но и во многих других местах). В двух соседних клозах одной функции для обозначения одного и того же формального параметра можно использовать совершенно разные образцы. Здесь играет роль способ сопоставления, а не наименование формального параметра. И именно на это необходимо обращать внимание программисту.

Где же используются образцы? В языке Haskell имеется несколько мест их использования, которые резонно перечислить:

1) В клозах определений функций. Это обычный способ использования образцов, в котором механизм сопоставления с образцами работает наиболее ярким образом, если так можно выразиться.

2) В определениях альтернатив оператора **case**. Каждая альтернатива этого оператора оформляется образцом. Обычно это константные образцы, но могут быть и вполне себе переменные.

3) В определениях анонимных функций. Здесь способ использования образцов абсолютно такой же, как и при определении обычных функций.

4) В последовательностях действий, оформленных оператором **do**. В этом случае образец стоит слева от операции (<-), и этот образец просто заменяет соответствующий образец в анонимной функции, которая скрывается при помощи удобного синтаксиса.

Остаётся рассмотреть некоторые специальные виды образцов, которым посвящены подразделы ниже.

1.2.1. Образцы вида $(\mathbf{n + k})$

В целях обеспечения совместимости с математической нотацией в языке Haskell имеется возможность использования так называемых образцов вида $(n + k)$. Это значит, что в образцах можно использовать символ (+), который обозначает арифметическое сложение чисел. Другими словами, для числовых значений можно использовать выражение последующих вычисляемых элементов последовательности через уже имеющиеся. Такой способ представления формул принят в математике, а потому в языке Haskell было решено внедрить эту технику.

Однако такие образцы не приветствуются большинством программистов на языке Haskell, а потому их невозможно найти в стандартном модуле `Prelude`, чтобы привести в качестве примера. Поэтому пример будет достаточно простой:

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

Это функция для вычисления факториала заданного числа. Как видно, она соответствует математической рекуррентной формуле:

$$(n + 1)! = (n + 1) * n!$$

Конечно, этот пример несколько надуман, но именно рекуррентные формулы в математике послужили прототипом образцов вида $(n + k)$ в языке Haskell. Надо отметить, что такой способ записи существует не во всех функциональных языках. И в сообществе любителей языка Haskell по этому поводу даже был раскол, и часть ведущих программистов и специалистов подписала меморандум о запрете образцов вида $(n + k)$. Но такие образцы всё равно были включены в стандарт языка. Тем не менее использование этого вида образцов лежит на совести разработчика программного обеспечения. Некоторым они нравятся за дополнительную выразительность и подобие математическим формулам.

Но необходимо помнить, что в образцах можно использовать только операции конструирования данных (конструкторы). Арифметический знак (+) — это единственный символ операции, который можно использовать в образцах.

1.2.2. Именованные образцы

Другими образцами специального вида являются так называемые «именованные образцы». Конечно, это достаточно условное название, потому как любой образец, который не является константой или маской подстановки (_), имеет какой-то идентификатор, по которому к нему можно обратиться. Но здесь речь идёт о несколько иной вещи.

Иногда в вычислительном процессе, описываемом некоторой функцией, необходимо обратиться к фактическому значению входного параметра как к целому, но при этом имеется надобность и в разложении этого фактического значения на составные части (если это значение сложного типа). Так, к примеру, в стандартном модуле `Prelude` определена функция `dropWhile`, которая «выкидывает» элементы входного списка, пока значение входного предиката на них равно `True` (подробно эта функция описана на стр. 261):

```
dropWhile p []                      = []
dropWhile p xs@(x:xs') | p x        = dropWhile p xs'
                       | otherwise = xs
```

Именованный образец записывается при помощи символа (`@`), который разделяет наименование образца как целого и запись внутренней структуры образца. В приведённом примере образец `xs@(x:xs')` является именованным. Это список, который обрабатывает функция `dropWhile`. Если в теле функции имеется необходимость обратиться к этому списку в целом, то используется идентификатор `xs`, но если надо получить голову и хвост этого списка, то можно воспользоваться идентификаторами `x` и `xs'` соответственно.

Эта техника позволяет обращаться к значениям сложных типов двояко: во-первых, как к целому, а во-вторых, по частям типов. Этого можно добиться и другими способами (по крайней мере, двумя), но именованные образцы дают наиболее естественный и понятный путь.

Какие же способы позволяют сделать то же самое? Первый способ — повторная сборка объекта сложной природы. В этом случае второй клоз функции `dropWhile` выглядел бы так:

```
dropWhile p (x:xs') | p x       = dropWhile p xs'
                    | otherwise = x:xs'
```

Как видно, вариант `otherwise` повторно собирает объект (список) из элемента `x` и списка `xs'` при помощи конструктора `(:)`. Здесь именно происходит создание нового объекта, а не использование входного параметра, как в предыдущем варианте функции. Хотя это приведёт к абсолютно такому же результату, ресурсоёмкость такого решения выше.

Второй вариант — использование селекторов для доступа к элементам значения сложного типа. В этом случае второй клоз функции `dropWhile` будет выглядеть уже так:

```
dropWhile p xs | p (head xs) = dropWhile p (tail xs)
               | otherwise   = xs
```

Здесь видна другая крайность, а именно — явное использование функций доступа в случае, когда этого можно было бы избежать при помощи использования образцов. Именно поэтому именованные образцы являют собой достаточно изящный механизм, который следует использовать в тех случаях, когда в одной и той же функции необходимо обращаться к фактическому значению входного параметра как к целому, так и к его частям.

1.2.3. Ленивые образцы

Остаётся рассмотреть так называемые «ленивые образцы», сопоставление с которыми происходит только в случае, если сам образец или какая-либо из его частей используется в теле функции. Определение таких образцов иногда позволяет выгодно уменьшить ресурсоёмкость вычислительных процессов.

Для определения ленивых образцов используется символ (`~`). В стандартном модуле `Prelude` имеется всего пара функций, в которых определены ленивые образцы. Например, функция `unzip`, которая получает на вход список пар, а возвращает пару списков (подробно описывается на стр. 270):

```
unzip = foldr (\(a,b) ~(as,bs) -> (a:as, b:bs)) ([], [])
```

При рассмотрении подобных конструкций с ленивыми образцами необходимо помнить, что транслятор языка Haskell просто раскрывает ленивые образцы следующим образом: `f ~x = e` трансформируется в `f p = let x = p in e`. Это и обозначает, что только при использовании идентификатора `x` внутри выражения `e` ему будет сопоставлено формальное значение.

1.3. Ввод и вывод

Ни один язык программирования общего назначения не может обойтись без работы с внешними устройствами. Однако должно быть вполне понятно, что ввод/вывод — это область программирования, где очень серьёзно встаёт вопрос о недетерминированности функций и наличии у них побочных эффектов. Встаёт очень сложная проблема, поскольку в чистых функциональных языках, каким является язык Haskell, такие функции запрещены. Более того, они просто запрещены теорией функционального программирования. Но отказ от реализации ввода/вывода не позволит языку стать языком общего назначения.

Решение было найдено при помощи выделения операций ввода/вывода в отдельный «подъязык», в рамках которого функции с определённым типом могли выполнять действия (вызывать побочный эффект изменения внешнего окружения — устройств вывода) и быть недетерминированными. Однако вне этого подъязыка язык Haskell остаётся чистым.

1.3.1. Действия ввода/вывода

Честно говоря, нельзя думать о таких вещах, как вывод строки на экран или чтение строки с клавиатуры, как о функциях. Поэтому в языке Haskell используется понятие «действие» для описания таких специальных функций. Более того, эти функции должны иметь и специальный тип. Например, функция `putStrLn`, определённая в стандартном модуле `Prelude` и необходимая для того, чтобы вывести на экран строку, заканчивающуюся символом перевода строки, имеет следующий тип (подробно описана на стр. 273):

```
putStrLn :: String -> IO ()
```

Таким же образом и тип функции `getChar`, которая считывает с клавиатуры один символ, выглядит так (подробно описана на стр. 136):

```
getChar :: IO Char
```

Как специальная функция, определённая в языке Haskell, каждое действие ввода/вывода должно возвращать какое-то значение. Для того чтобы различать эти значения от базовых, типы этих значений как бы обёрнуты контейнерным

типом `IO`. Поэтому любое действие ввода/вывода будет иметь в сигнатуре своего типа символы `IO`, которые предваряют собой другие типы.

Необходимо отметить, что действия в отличие от обычных функций выполняются, а не вычисляются. Как это сделано и чем выполнение действия отличается от вычисления значений функции, полностью зависит от транслятора. Однако запустить действие на выполнение просто так нельзя, для этого необходимо использовать ключевое слово **do** либо специальные методы, определённые в классе `Monad`. Но существует одно действие, которое выполняется само. Это функция `main`, которая является, как и в языках программирования типа C, точкой входа в откомпилированные программы. Именно поэтому тип функции `main` должен быть `IO ()` — это действие, которое автоматически выполняется первым при запуске программы.

Тип `IO ()` — это тип действия, которое ничего не возвращает в результате своей работы. Иные действия, имеющие некоторый результат, который можно получить в программе, должны возвращать другой тип. Так, к примеру, действие функции `getChar` заключается в чтении символа с клавиатуры, причём далее этот считанный символ возвращается в качестве результата. Именно поэтому тип этого действия — `IO Char`.

Любые действия связываются в последовательности при помощи ключевого слова **do**. При помощи него можно связывать вызовы функций (в том числе и использовать операторы ветвления **if** и **case**), получение значений в образцы (при помощи символа (`<-`)) и множество определений локальных переменных (ключевое слово **let**).

Например, так можно определить программу, которая читает символ с клавиатуры и тут же выводит его на экран:

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

Если такую программу откомпилировать, то функция `main` будет являться точкой входа в программу. Для интерпретаторов это не важно — в режиме интерпретации можно вызывать любые функции, которые требуются разработчику или пользователю. Однако для компилируемых программ имеется еще один нюанс — для того чтобы такие программы можно было успешно откомпилировать

и запустить, необходимо, чтобы функция `main` находилась в одноименном модуле `Main`. Хотя во многих трансляторах при отсутствии явного указания имени модуля по умолчанию используется имя `Main`, и поэтому если не указывать имя модуля, всё откомпилируется в таких трансляторах и запустится замечательно, необходимо помнить о такой особенности.

Ещё один небольшой пример. Пусть имеется функция `isReady`, которая должна возвращать значение `True`, если нажата клавиша «y», и значение `False` в остальных случаях. Нельзя просто написать:

```
isReady :: IO Bool
isReady = do
  c <- getChar
  c == 'y'
```

В этом случае результатом выполнения операции сравнения будет значение типа `Bool`, а не `IO Bool`, так как результат и соответственно его тип в списке **do** определяются по последнему действию. В этом случае необходимо воспользоваться методом `return`, который из простого типа данных делает контейнерный, в котором хранится значение исходного типа. То есть в предыдущем примере последняя строка определения функции `isReady` должна была выглядеть как `return (c == 'y')`.

В следующем примере показана более сложная функция, которая считывает строку символов с клавиатуры:

```
getString :: IO String
getString = do
  c <- getChar
  if (c == '\n')
    then return ""
    else do
      cs <- getString
      return (c:cs)
```

Необходимо отметить, что операторы множественного ветвления алгоритма можно вполне использовать внутри последовательности действий, которые определяются ключевым словом **do**. Однако имеется очень важный момент — в частях **then** и **else** условного выражения и выражениях для альтернатив в операторе **case** необходимо также использовать последовательность действий, оформленную в виде списка **do**. Только в случае если в этих местах используется толь-

ко одно действие, ключевое слово **do** можно не использовать (это видно в части **then** в предыдущем примере).

Такое положение дел связано с тем, что в таких местах необходимо указывать выражения, имеющие тот же тип, который возвращает само условное выражение. А раз такое условное выражение участвует в последовательности действий **do**, то оно должно иметь соответствующий тип `IO`.

Действия ввода/вывода являются обычными значениями в терминах языка Haskell. То есть действия можно передавать в функции в качестве параметров, заключать в контейнерные структуры данных и вообще использовать там, где можно использовать данные языка Haskell. В этом смысле система операций ввода/вывода является полностью функциональной. Таким образом, к примеру, можно предположить возможность определения списка действий:

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

Сам по себе этот список не возбуждает никаких действий, его определение не приводит к выполнению записанной в нём последовательности операций ввода/вывода. Этот список просто содержит действия как описания операций ввода/вывода. Для того чтобы выполнить эту последовательность, то есть возбудить все её действия, необходима некоторая функция, на вход которой подаётся подобный список. Её определение может выглядеть следующим образом:

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

Эта функция может использоваться для определения функции `putString`, которая выводит заданную строку на экран (её действие в чём-то противоположно действию функции `getString`, которая была определена чуть ранее):

```
putString :: String -> IO ()
putString s = sequence (map putChar s)
```

На этом примере видно очень явное отличие системы ввода/вывода языка Haskell от таких же систем императивных языков. Если бы в каком-нибудь императивном языке была бы определена функция, аналогичная функции `map`, то она бы в данном примере выполнила кучу действий. Однако вместо этого в языке Haskell просто создается список действий (одно для каждого символа строки), который потом обрабатывается функцией `sequence` для выполнения.

1.3.2. Обработка исключений

Во время работы с операциями ввода/вывода очень часто происходят ситуации, когда происходят ошибочные ситуации. Неправильный ввод с клавиатуры, невозможность открытия файла в силу его отсутствия, невозможность записи в файл в силу отсутствия прав на эту операцию — это далеко не полный список ситуаций, которые могут привести к ошибкам, которые, в свою очередь, могут остановить программу. В обычных языках программирования в целях обработки подобных ошибочных ситуаций был разработан механизм возбуждения и дальнейшего отлова так называемых исключений. Так, функция, в которой произошла ошибка, обрамляет её в некоторый объект, называемый исключением, а потом передает его в обслуживающий модуль. В этом случае программист может самостоятельно отловить исключение и обработать его, а может положиться на операционную систему, у которой имеются стандартные средства для обработки исключений (но в этом варианте придётся смириться с тем, что во многих случаях программа будет остановлена).

Такие же ошибочные ситуации могут возникнуть и внутри действий `IO` в языке Haskell. Для этих целей используется абсолютно такой же механизм, как и в высокоуровневых императивных языках программирования. Однако если в таких языках имеется специальный синтаксис для описания блоков программного текста, которые предназначены для отлова исключений, то в языке Haskell используется обычный механизм — использование функций.

Дело в том, что любая ошибка ввода/вывода, которая может возникнуть внутри системы ввода/вывода, имеет тип `IOError`, а обработчик исключительной ситуации обязан иметь тип `IOError -> IO a`. Для связывания обработчика с кодом, в котором возможно возникновение ошибочной ситуации, имеется специальная функция `catch`, имеющая тип

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

По сигнатуре этой функции видно, что её аргументами являются некоторое действие (первый аргумент, который может представлять собой и список действий, оформленный при помощи ключевого слова **do**) и обработчик исключений (второй аргумент). Возвращает функция результат выполнения некоторых действий, причём эти действия выбираются по простой логике. Если действие, описанное в первом аргументе, выполнено успешно без возбуждения исключения, то просто возвращается результат этого действия. Если же в процессе выполнения действия возникла ошибка, то она передается обработчику исключений в качестве операнда типа `IOError`, после чего выполняется сам обработчик. Этот обработчик также выполняет некоторое действие, результат которого возвращается в качестве окончательного результата функции `catch`.

Таким образом, можно написать более сложные функции, которые будут грамотно вести себя в случае выпадения ошибочных ситуаций:

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


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

На примере определения этих функций видно, что можно использовать вложенные друг в друга обработчики ошибок. В функции `getChar'` отлавливается ошибка, которая возникает при обнаружении символа конца файла. Если ошибка другая, то при помощи функции `ioError` она отправляется дальше и ловится обработчиком, который «сидит» в функции `getString'`. Для определённо-

сти в языке Haskell предусмотрен обработчик исключений по умолчанию, который находится на самом верхнем уровне вложенности. Если ошибка не поймана ни одним обработчиком, который написан в программе, то её ловит обработчик по умолчанию, который выводит на экран сообщение об ошибке и останавливает программу.

Более подробно система ввода/вывода языка Haskell описывается в части II. этого справочника при детальном описании модуля `IO` и некоторых других.

1.4. Приёмы программирования

Язык Haskell, как и любой иной высокоразвитый язык программирования, имеет уже устоявшиеся приёмы, которые позволяют писать программное обеспечение более быстро и эффективно. Такие приёмы часто называются «идиомами». Конечно, все они обычно постигаются на практике путём долгого самостоятельного обучения на своём опыте или опыте коллег. А потому этот раздел поможет читателю не тратить времени на самостоятельный поиск и постижение того, что уже давным-давно сделано.

1.4.1. Двумерный синтаксис

Определения функций, содержащие в себе такие ключевые слова, как **case**, **let**, **where** и **do**, используют так называемый двумерный синтаксис в случаях, когда после этих служебных слов идёт несколько выражений. Этот двумерный синтаксис позволяет структурировать исходный код, а также не перегружать его лишними символами, необходимыми для разделения выражений. На самом деле правильная запись выражений, находящихся после перечисленных служебных слов, подразумевает заключение их в фигурные скобки — `{}`, а также разделение выражений символом (`;`), как это принято в большинстве языков программирования. Так, к примеру, постфиксные локальные определения можно записывать следующим образом:

```
abcValue a b c = x * y / z where {x = a + b; y = b + c; z = c + a}
```

Однако создатели языка Haskell решили в дополнение к такой записи разрешить программисту использовать двумерный синтаксис для записи подобных определений следующим образом:

```
abcValue a b c = x * y / z
  where
    x = a + b
    y = b + c
    z = c + a
```

Смысл использования двумерного синтаксиса состоит в следующем. Каждое выражение, следующее после ключевого слова **where** (или другого ключевого слова, поддерживающего двумерный синтаксис), должно находиться на новой строке и при этом начинаться с одного и того же знакоместа в самой строке, то есть все выражения должны находиться как бы в столбик друг под другом, начинаясь на одной и той же позиции.

1.4.2. Рекурсия и корекурсия

В языке Haskell нет таких операторов, как **for**, **while** или **goto**. Это связано с тем, что эти операторы явно императивны, то есть они определяют пошаговый порядок исполнения некоторых инструкций. Как уже было неоднократно упомянуто, язык Haskell, как чистый функциональный язык, не имеет (и не должен иметь) подобных средств. Для организации цикла здесь используется другой механизм — рекурсия. А такая конструкция, как безусловный переход, просто невыразима в терминах функционального программирования.

Общие понятия

Рекурсия известна большинству программистов, использующих в своей работе императивную парадигму. Под этим термином обычно подразумевается вызов функцией самой себя, прямо или косвенно. Прямой вызов обозначает то, что функция вызывает саму себя непосредственно в собственном теле. Соответственно, косвенный рекурсивный вызов — это вызов функцией другой функции, которая уже в свою очередь вызывает изначальную прямо или косвенно.

В функциональном программировании рекурсия является широко используемым механизмом. Однако в силу декларативности функционального подхода разработчик программного обеспечения не думает о рекурсии как о вызовах функции (что являлось бы определением порядка исполнения программы, то есть отвечало бы на вопрос «как?»). Рекурсия — это средство, с помощью которого

функция может быть определена через себя, то есть использует саму себя в качестве элемента своего определения. Это полностью аналогично рекурсивным определениям в обычной речи или математике. Например, в качестве определения понятия «предок» можно использовать следующее: «Предок некоего человека — это его родитель или предок его родителя». Этот пример полностью декларативен, он сжато и ясно объясняет, кто такой предок. По всей видимости, рекурсивные механизмы являются более близкими психологии человека, чем инструкции, в которых объясняются пошаговые алгоритмы.

В качестве примера рекурсивной функции можно рассмотреть тривиальный поиск факториала заданного числа. Эту функцию можно определить следующим образом:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

В определении функции для вычисления факториала имеется два момента, которые стоит отметить особо. Во-первых, это организация цикла. В императивных языках возможно записать вычисление факториала при помощи итерации, определяемой через ключевое слово **for**. В языке Haskell подобное определение невозможно просто из-за отсутствия подобного ключевого слова и, собственно, идиомы итеративного процесса. Поэтому для организации циклических вычислений используется только рекурсивные функции. Во-вторых, свойства функции `factorial` могут быть доказаны при помощи математической индукции:

1) Факториал для 0 действительно равен единице (по математическому определению).

2) Допустим, что доказано, что вызов `factorial n` возвращает значение $n!$. Необходимо доказать, что вызов `factorial (n + 1)` возвратит $(n+1)!$.

3) Факториал $(n+1)$ есть произведение $(n+1)$ на факториал n (по определению). Поскольку доказано, что `factorial n` возвращает значение $n!$, вызов `factorial (n + 1)`, равный по определению функции произведению `(n + 1)` на значение `factorial n`, вернёт значение $(n+1)!$ Что и требовалось доказать.

Необходимо обратить внимание на то, что индукция может использоваться и для создания определений рекурсивных функций. В функциональном про-

граммировании это часто используемый приём. Допустим, имеется необходимость подсчитать произведение всех чисел из некоторого списка. Список является индуктивным типом, определение которого ограничивается пустым списком. Для пустого списка произведение равно единице. Это естественно, так как `n * product []` должно быть равно `n`. Произведение же чисел непустого списка будет равно произведению его головы на произведение чисел его хвоста:

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

Корекурсия

Теперь можно кратко рассмотреть другой схожий механизм — корекурсию. Корекурсия является дуальным понятием к рекурсии. В то время как рекурсивные определения работают с данными, осуществляя «спуск» от начальных значений входных параметров к таким, на которых происходит остановка рекурсивного процесса, корекурсия работает с коданными, производя «накрутку» значений на начальные величины входных параметров, никогда не останавливаясь.

Понятие коданных в явном виде не определено в языке Haskell. Для общего понимания имеет смысл привести неформальное определение этого понятия (формально оно определено в рамках теории категорий):

1) Коданные, в отличие от данных, скрывают свою структуру.

2) В коданных нет атрибутов, имеющих смысл «значение».

3) Для коданных существуют только методы доступа. Некоторые из них возвращают данные, некоторые — коданные.

4) Обычно коданные представляют из себя бесконечные структуры.

В языке Haskell ближайшим отображением коданных могут являться рекурсивные абстрактные типы данных, например список. Ниже корекурсия как раз и будет рассмотрена на примере работы со списками.

Обычно корекурсия используется для построения бесконечных списков, основанных на представлении разнообразных последовательностей. Вот пример корекурсивной функции, возвращающей бесконечный список чисел Фибоначчи:

```
fibonaccies = 0:1:zipWith (+) fibonaccies (tail fibonaccies)
```

Как работает эта функция? Её определение не так тривиально, хотя является классическим при рассмотрении понятия корекурсии в языке Haskell. Для того чтобы понять, как данная функция вычисляет бесконечный список чисел Фибоначчи, можно рассмотреть несколько шагов. Перед этим имеет смысл напомнить, что функция `zipWith` из стандартного модуля `Prelude` «сшивает» два списка при помощи заданной операции.

Итак, на вход функции `zipWith` подаётся операция (+) и два списка: первый — результат вычисления функции `fibonacci`, второй — хвост этого результата. Как быть, если оба списка бесконечны и ещё не вычислены (имеются только первые два элемента: 0 и 1)? Корекурсивная функция вкупе с использованием механизма ленивых вычислений довольствуется малым — использует то, что есть, то есть первые два элемента списка. Их можно подать на вход функции `zipWith`, которая применит к ним операцию (+) и получит третий элемент. Далее всё повторяется, ведь уже имеется три элемента, и можно двигаться далее. Этот процесс можно пояснить следующим рассмотрением вычислений:

```
fibonaccies:      0 1 1 2 3  5
tail fibonaccies: 1 1 2 3 5  8
zipWith (+):      1 2 3 5 8 13
```

Другим хорошим примером, который помогает понять корекурсию, является функция для вычисления бесконечного списка простых чисел, основанная на алгоритме Эратосфена, известном под наименованием «решето Эратосфена»:

```
primes = sieve [2..]
  where
    sieve (x:xs) = x:sieve (filter ((/= 0).(`mod` x)) xs)
```

Как видно, эта функция также использует корекурсию для получения очередной итерации списка чисел, из которого «отсеяны» делители найденного на этой итерации простого числа. Собственно, корекурсия — замечательный механизм, который, однако, также необходимо использовать с осторожностью, как и рекурсию.

1.4.3. Накапливающий параметр и хвостовая рекурсия

Рекурсия и корекурсия — достаточно ресурсоёмкие технологии организации вычислительных процессов, которые требуют бóльших затрат памяти, нежели простые итерации, и поэтому в рамках парадигмы функционального программирования очень часто исключительно серьёзно встаёт проблема расхода памяти. Эту проблему можно пояснить на примере функции, вычисляющей длину заданного списка:

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

Если, к примеру, рассмотреть вычисления этой функции с аргументом `['a', 'b', 'c']`, то можно будет увидеть следующую последовательность (здесь символом (`==>`) обозначается очередной шаг вычислений):

```
    length ['a', 'b', 'c']
==> 1 + length ['b', 'c']
==> 1 + (1 + length ['c'])
==> 1 + (1 + (1 + length []))
==> 1 + (1 + (1 + 0))
==> 1 + (1 + 1)
==> 1 + 2
==> 3
```

На примере этого достаточно простого вычисления наглядно видно, что при рекурсивных вызовах функций довольно серьёзно используется память — и на хранение промежуточных результатов вычисления, и на хранение адресов возврата из вложенных рекурсивных вызовов. В данном случае количество памяти пропорционально длине списка, но вычисления могут быть намного сложнее, да и параметров может быть больше. Возникает резонный вопрос: можно ли так написать функцию вычисления длины списка (и ей подобные), чтобы память использовалась минимально?

Чтобы ответить на данный вопрос положительно, необходимо изучить понятие аккумулятора (накопителя, накапливающего параметра, сумочки или кармашка). Для этого можно рассмотреть следующий пример построения функции для вычисления длины заданного списка, но уже с аккумулятором.

```
length_a :: [a] -> Int
length_a l = lngt l 0


lngt :: [a] -> Int -> Int
lngt [] n     = n
lngt (x:xs) n = lngt xs (n + 1)
```

В этом примере второй параметр функции `lngt` выполняет роль аккумулирующего параметра, именно в нём содержится результат, который возвращается после окончания рекурсии. Сама же рекурсия в этом случае принимает вид «хвостовой», память при этом расходуется только на хранение адресов возврата функции.

Хвостовая рекурсия представляет собой специальный вид рекурсии, в которой имеется единственный вызов рекурсивной функции, и при этом этот вызов выполняется после всех вычислений.

При реализации в трансляторах функциональных языков вычисления хвостовой рекурсии могут выполняться при помощи итераций в постоянном объёме памяти. На практике это обозначает, что такие трансляторы должны уметь распознавать хвостовую рекурсию и реализовывать её в виде цикла. В свою очередь метод накапливающего параметра не всегда приводит к хвостовой рекурсии, однако он в любом случае помогает уменьшить общий объём расходуемой памяти.

Для того чтобы понять, какие функции можно определить при помощи использования накапливающего параметра, необходимо рассмотреть принципы построения таких функций. Таких принципов несколько:

1) Вводится новая функция с дополнительным аргументом (аккумулятором), в котором накапливаются результаты вычислений.

2) Начальное значение аккумулирующего параметра задаётся в равенстве, связывающем старую и новую функции.

3) Те равенства исходной функции, которые соответствуют выходу из рекурсии, заменяются возвращением аккумулятора в новой функции.

4) Равенства, соответствующие рекурсивному определению, выглядят как обращения к новой функции, в которых аккумулятор получает то значение, которое возвращается исходной функцией.

Возникает вопрос: любую ли функцию можно преобразовать так, чтобы вычисления производились при помощи аккумулятора? Очевидно, что ответ на этот вопрос отрицателен. Построение функций с накапливающим параметром — приём не универсальный, и он не гарантирует получения хвостовой рекурсии. С другой стороны построение определений с накапливающим параметром является делом творческим.

1.4.4. Бесточечная нотация

В функциональном программировании очень часто функции определяются при помощи композиции других функций без обращения внимания на то, сколько аргументов в действительности функция принимает. Например, уже приводился пример функции `sum` из стандартного модуля `Prelude`, которая определена в нём следующим образом (подробно эта функция описывается на стр. 311):

```
sum :: Num a => [a] -> a
sum = foldl (+) 0
```

Как видно, в определении нет формальных параметров, но тип функции ясно указывает, что единственным параметром функции должен быть список. Транслятор языка Haskell самостоятельно определит количество необходимых параметров функции исходя из того, сколько данных требуется в её теле для выполнения вычислений. Поэтому на языке Haskell очень любят записывать определения функций примерно следующим образом:

```
f = g . h . k
```

вместо:

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

Первая запись представляет собой так называемую бесточечную нотацию. Этот стиль определения функций достаточно полезен при определении эффективных вычислительных процессов. Но в любом случае это вообще очень хорошая практика. Бесточечная нотация позволяет разработчикам программного обеспечения думать на более высоком уровне абстракции в терминах композиции функций, а не на низком в терминах передачи данных из одного вычислительного процесса в другой.

Внимательный и дотошный читатель может возразить, что в бесточечной нотации используется больше точек. Речь, конечно, идёт об операции (`.`), которая определена в стандартном модуле `Prelude` и осуществляет композицию функций. Однако в наименовании техники корень «точка» не относится к этой операции. Этот термин взят из топологии, которая оперирует пространствами, состоящими из точек, а также функциями между пространствами. Так что бесточечное определение функции — это такое определение, в котором не участвуют «точки» пространства, над которым действует функция. В терминах языка Haskell «точками» называются элементы данных, то есть формальные параметры. Другими словами, пространства в языке Haskell — это типы, а точки — их значения.

Однако с бесточечной нотацией имеется одна проблема. При неблагоразумном использовании эта методика может легко привести к эффекту «обфускации» исходного кода. При композиции цепочек функций высшего порядка бывает очень сложно мысленно вывести тип таких композиций, так как нет зацепок в виде аргументов, тип которых весьма помогает в процессе получения типа функции.

Также функции, которые определены в бесточечном стиле, обладают свойством тяжёлой модификации в случаях, когда необходимо внести в функциональность малейшие изменения. Иной раз для этого приходится переписывать всю функцию. Это происходит потому, что композиция нескольких функций является более сложным элементом программирования, нежели вызов функций в обычном стиле.

1.4.5. Анонимные функции

В языке Haskell имеется возможность использовать в определениях функций вызовы так называемых анонимных функций, то есть таких функций, которые определены исключительно в одном месте, а именно — месте своего вызова. Эта возможность может быть использована, к примеру, для быстрого прототипирования или для вызова простой функции, определение которой на верхнем уровне бессмысленно.

Эта возможность основана на методах λ-исчисления, которое является ядром языка Haskell. Через λ-нотацию можно определять любую функцию, в том числе и анонимную, поскольку любая λ-абстракция является безымянной функцией. Необходимо отметить, что здесь не будет описываться теория λ-исчисления, поскольку это выходит далеко за рамки справочника. В любом случае тот читатель,

который не знаком с этим формализмом, может найти его описание в специализированной литературе.

Использовать λ-исчисление в языке Haskell несложно. Ниже показан пример, где определены функции `multiply` и `double` именно при помощи λ-исчисления.

```
multiply = \x y -> x * y


double = \x -> x * 2
```

Эти определения функций записаны в полном соответствии с такими же определениями, которые могут быть сделаны при помощи λ-выражений:

$$\lambda xy.(x * y),$$

$$\lambda x.(x * 2).$$

То есть видно, что λ-абстракции кодируются на языке Haskell просто. Символ (λ) заменяется на символ (\), а символ (.) заменяется на стрелку (->). Остальное даётся без изменения (естественно, принимая во внимание синтаксис языка Haskell).

```
squares = map (\x -> x^2) [0..]
```

Этот пример показывает вызов анонимной функции, возводящей в квадрат переданный параметр (пример большей частью надуман, так как для этих целей нет особой надобности использовать анонимную функцию). Результатом выполнения этой инструкции будет бесконечный список квадратов целых чисел, начиная с нуля.

Необходимо отметить, что в языке Haskell используется упрощённый способ записи λ-выражений, так как в точной нотации функцию `multiply` из примера выше правильнее было бы написать как

```
multiply = \x -> \y -> x * y
```

что в большей мере соответствует точной математической записи:

$$\lambda x.\lambda y.(x * y).$$

Такой способ определения анонимных функций и λ-выражений также возможен, а использование того или иного способа предоставляется на выбор разработчику программного обеспечения.

1.4.6. Охрана

Охрана, или охраняющее выражение, — это логическое выражение (то есть возвращающее значение типа `Bool`), которое накладывает некоторые ограничения на используемые образцы. Охрана может применяться в языке Haskell в качестве дополнительной возможности к технологии сопоставления с образцом. При определении функций охраняющие выражения записываются после образцов, но перед выражениями, являющими собой описание вычислительного процесса. Для разграничения охраняющих выражений и образцов используется символ (`|`). Например, следующим образом можно определить функцию для получения знака числа:

```
sign :: Integer -> Int
sign x | x < 0  = -1
       | x == 0 = 0
       | x > 0  = 1
```

Как и в случае с процессом сопоставления с образцами, просмотр охраняющих выражений производится сверху вниз до первого выражения, значение которого равно `True`. В этом случае значением функции будет то, что записано после найденной охраны. После этого вычислительный процесс останавливается, поиск среди оставшихся охраняющих выражений не производится. Именно поэтому обычно в самом конце списка охраны и записывается слово `otherwise` — оно «поймает» вычисления в любом случае, даже если никакое иное охраняющее выражение не было истинно. Это слово не является ключевым, оно определено в качестве константной функции в стандартном модуле `Prelude` и является синонимом значения `True`. Сделано это для удобства записи и чтения исходных кодов программ (слово «otherwise» по-английски обозначает «в противном случае»).

Вполне понятно, что в определениях функций можно использовать обе технологии — и охрану, и образцы (разбивающие определения на несколько клозов). Например, предыдущую функцию можно было бы записать и так:

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

В книге [1] была допущена ошибка относительно этой записи. Было сказано, что такая запись содержит одну логическую ошибку — если подать на вход функции `sign'` число 0, то произойдет ошибка времени выполнения, так как транслятор не сможет найти подходящее выражение для вычисления. Это связано с тем, что при сопоставлении с образцами число 0 на входе подойдёт к первому клозу, после чего ни одно из охраняющих выражений не позволит вычислить значение функции. А раз сопоставление с образцами произошло успешно, клоз был выбран, то сам процесс сопоставления уже остановлен.

Однако это не так. Современные трансляторы языка Haskell распознают такие определения, а потому если подать на вход функции `sign'` значение 0, результат будет успешно вычислен — процесс сопоставления с образцами войдёт во второй клоз и позволит определить результат выполнения функции. Тем не менее крайне не рекомендуется использовать подобные определения.

Охраняющие выражения могут также использоваться и в сочетании с ключевым словом **case**, когда после выбора альтернативы по условию дополнительно производится выбор по охраняющему выражению. Например:

```
sign'' x = case x of
             0         -> 0
             _ | x < 0 -> -1
               | x > 0 -> 1
```

В этом случае также необходимо повышенное внимание уделять логической согласованности условий охраны и альтернатив выполнения, так как в полном соответствии с тем, что происходит при совместном использовании охраны и нескольких клозов, в этом случае может произойти то же самое.

Остаётся отметить, что механизм охраны в языке Haskell используется тогда, когда сложно использовать технологию сопоставления с образцами. Так, к примеру, это сложно сделать в прикладных функциях, которые обслуживают классы определённых типов значений, в которых доподлинно неизвестен тип обрабатываемых значений, поэтому использование конструкторов данных в образцах попросту невозможно (подробно см. главу 3.).

1.4.7. Определители списков

В языке Haskell имеется возможность просто и быстро конструировать списки, основанные на какой-нибудь простой математической формуле. Для этих целей используется технология, которая известна как определители списков. При помощи неё можно определять самые незаурядные последовательности, выраженные посредством списков. К таким последовательностям относятся как числовые ряды, основанные на какой-либо математической формуле, так и списки значений произвольных типов.

Наиболее общий вид определителей списков выглядит следующим образом:

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

Эта запись может быть прочитана как «список из всех таких `x`, взятых из списка `xs`». Терм «`x <- xs`» называется генератором. После такого генератора (он должен быть один для каждого образца, определяемого им) может стоять некоторое число выражений охраны, разделённых запятыми. В этом случае выбираются все такие значения `x`, для которых все выражения охраны истинны. То есть запись

```
[x | x <- xs,
     x > m,
     x < n]
```

можно прочитать как «список из всех таких `x`, взятых из списка `xs`, что (`x` строго больше `m`) *и* (`x` строго меньше `n`)».

Определители списков могут использоваться для создания списков из более сложных структур данных, нежели простые значения. Для этого можно использовать столько образцов, сколько требуется, и таких типов, которые требуются для решения задачи. Например:

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

Эта запись читается так: «список пар из всех элементов списков `xs` и `ys`, при этом первый элемент пары должен быть не больше второго элемента». Как видно, дело это несложное.

Можно рассмотреть несколько несложных примеров использования определителей списков. Например, бесконечный список простых чисел может быть построен при помощи следующей простейшей функции:

```
primes = [x | x <- [2..],
              isPrime x]
  where
    isPrime x = (dividers x == [1, x])


dividers x = [y | y <- [1..x],
                  x `mod` y == 0]
```

Функция `primes` возвращает список всех таких натуральных `x`, то есть взятых из множества натуральных чисел `[2..]` (единица выкинута из этого множества, так как она не является простым числом по определению), при этом на каждом таком значении `x` предикат `isPrime` должен давать истинное значение. Этот предикат возвращает значение «ИСТИНА» только для простых чисел, поскольку сравнивает список делителей заданного числа со списком, состоящим из значения «1» и самого числа, и если в полном соответствии с определением он равен такому списку, то предикат возвращает значение «ИСТИНА». В противном случае он возвращает значение «ЛОЖЬ».

Список делителей заданного числа вычисляется опять же при помощи технологии создания определителя списка. В список делителей заданного числа входят все такие числа, меньшие либо равные самому числу, остаток от деления исходного числа на которые равен нулю (остаток от целочисленного деления вычисляется при помощи функции `mod` из стандартного модуля `Prelude`, которая детально описывается на стр. 147).

Как видно из представленных примеров, язык Haskell предоставляет удивительную возможность для генерации списков довольно сложной природы. Причём дело не ограничивается только натуральными числами — в качестве значений, которые могут создаваться при помощи технологии определителей списков, как уже сказано могут использоваться любые структуры данных.

Глава 2.

Типы данных

Язык Haskell обладает достаточно развитой системой типов, в которую включены не только сами типы данных, но и некоторые другие механизмы, позволяющие работать с типами. Более того, из-за принятой в языке модели типизации (статическая модель Хиндли-Милнера) в трансляторах имеется мощнейший механизм вывода типов, который позволяет самостоятельно вычислять типы выражений, функций и других объектов. Этот механизм в дополнение нагружен системой классов, которые могут рассматриваться в качестве ограничений и интерфейсов. Всем этим аспектам языка посвящена эта и следующая главы.

2.1. Базовые типы

Как и любой другой развитый язык программирования, язык Haskell имеет некоторое множество так называемых «базовых типов», или встроенных типов данных, которые не определены где-либо в загрузочных модулях, но внедрены глубоко внутрь трансляторов. Обычно такие встроенные типы данных используются для представления фундаментальных величин: чисел, символов и прочих сходных объектов. Такие объекты обычно сложно описать во внешних модулях, тем более, что обычно они имеют прямое отображение в систему хранения данных на аппаратном уровне, — именно этим и объясняется подобное положение вещей.

Однако если во многих языках программирования встроенные типы охватывают довольно большое количество описанных объектов (булевские значения; символы, целые числа простой, двойной, четверной точности; также числа с плавающей точкой нескольких типов точности и даже строки), то язык Haskell отличается в этом вопросе некоторой аскетичностью. Дело в том, что в нём определено всего пять базовых типов, а именно (в алфавитном порядке):

1) `Char` — тип для представления литералов, занимающих в памяти один байт (восемь бит), символьный тип.

2) `Double` — тип для представления десятичных чисел с плавающей точкой двойной точности (значения типа `Double` занимают в два раза больше места в памяти, чем значения типа `Float`).

3) `Float` — тип для представления десятичных чисел с плавающей точкой.

4) `Int` — тип для представления целых чисел, входящих в интервал `[-2147483648, 2147483647]` ($[-2^{31}, 2^{31} - 1]$).

5) `Integer` — тип для представления целых чисел любой величины (вплоть до бесконечности).

Остальные типы данных можно получить из перечисленных при помощи операций конструирования типов (при этом создаются так называемые алгебраические типы данных), применения определённых разработчиком конструкторов типов и создания изоморфных типов данных.

Однако в целях оптимизации вычислительных процессов и способов хранения данных в языке Haskell на достаточно глубоком уровне «зашиты» два типа, которые базовыми не являются, но имеют некоторые преимущества (и ограничения) по сравнению с произвольными алгебраическими типами данных, которые могут создаваться программистом. Эти типы — список и кортеж. Их и необходимо рассмотреть в первую очередь.

2.1.1. Кортежи

В математике кортежом называется упорядоченный набор из n элементов (где n — любое натуральное число), называемое его компонентами. Различные (то есть стоящие на разных местах в одном и том же кортеже) компоненты могут

между собой и совпадать. Часто синонимом термина «кортеж» является термин «вектор», что связано с наиболее естественной интерпретацией кортежа как точек n-мерного пространства или упорядоченных совокупностей их координат. Посредством кортежей удобно характеризовать объекты, описываемые при помощи n независимых друг от друга признаков.

Это понятие довольно широко используется в математике, поэтому в языке Haskell понятие кортежа нашло непосредственную реализацию в системе типов и структур данных. К сожалению, как будет показано далее, в языке Haskell нет возможности сделать список, состоящий из элементов разных типов, как это, к примеру, можно сделать в языке Lisp. Это довольно существенное ограничение, которое не позволяет использовать такие значение, как, например, `[1, 'a']`. Однако возможность объединения разнотиповых элементов данных в одну структуру дают именно кортежи, которые в языке Haskell имеют тождественное с математическим определение. Кортеж выглядит как некоторый набор значений, заключённых в круглые скобки и разделённых запятыми. Например:

```
(14, 'r', [9, 6, 5])
```

Представленный выше кортеж состоит из трёх значений — целого числа, символа и списка целых чисел. Такие кортежи сами являются самостоятельными элементами данных, которые передаются в вычислительные процессы как одно целое. Хотя, в принципе, их использование в языке Haskell не так необходимо, как это может показаться на первый взгляд. Вполне возможно написать программу и без использования кортежей вовсе. Одно из главных назначений кортежей — написание некаррированных функций.

В стандартном модуле `Prelude` имеются некоторые функции для работы с кортежами. В первую очередь к этим функциям относятся такие функции, как `fst` и `snd` (подробно описаны на стр. 136 и стр. 163 соответственно), которые принимают на вход кортеж, состоящий из двух элементов любого типа. Первая функция возвращает первый элемент кортежа, вторая — второй.

Что же касается типов кортежей, то они определяются очень просто. Для любого кортежа, состоящего из n членов, типом будет считаться кортеж из n членов, на каждой позиции которого стоит тип элемента данных на соответствующей позиции исходного кортежа. В качестве примеров можно привести следующие кортежи:

1) `(1, 1) :: (Num a, Num b) => (a, b)`

2) `(14::Int, 'n') :: (Int, Char)`

3) `("Hello", "World") :: (String, String)`

4) `([1, 1, 2, 3, 5], ["Fibonacci"], True) :: Num a => ([a], [String], Bool)`

5) `(fst, snd) :: ((a, b) -> a, (c, d) -> d)`

С математической точки зрения общая формула для вычисления типа кортежа записывается следующим образом:

$$\#(a_1, a_2, \ldots, a_n) : (\#a_1, \#a_2, \ldots, \#a_n)$$

Естественно, для вычисления типов кортежей в языке Haskell используется тот же механизм, что и для вычисления типов прочих структур данных и функций (механизм, основанный на модели типизации Хиндли-Милнера), а потому выводимый тип будет наиболее общим. Это видно на первом примере, где имеется контекст `(Num a, Num b) =>`, который обозначает, что типы `a` и `b` принадлежат классу числовых величин `Num`.

2.1.2. Списки

Традиционно списки являются основной структурой данных, которая обрабатывается при помощи функциональных языков программирования. Эта ситуация сложилась из-за того, что в первом таком языке — Lisp — списки были той структурой данных, при помощи которой описывалось вообще всё, вплоть до самих программ на этом языке.

В языке Haskell используется реализация списков, максимально приближенная математическому определению. Определение типа «список над элементами типа `a`» на языке Haskell выглядит следующим образом (синтаксис для определения структур данных описывается в разделе 2.2.):

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

Данное определение в целях оптимизации обычно встроено в глубины трансляторов языка Haskell (поэтому оно некорректно с точки зрения синтаксиса

языка, а приведено исключительно в целях понимания). Тем не менее основные функции и экземпляры классов для типа «список» всегда определяются вне транслятора во внешних библиотеках или модулях. Например, в стандартном модуле `Prelude`, который автоматически импортируется во все проекты и модули, определены функции для доступа к элементам списка и для проверки списков на пустоту (данные функции подробно описываются на стр. 251 и стр. 393):

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


tail :: [a] -> [a]
tail (_:xs) = xs


null :: [a] -> Bool
null []    = True
null (_:_) = False
```

На самом деле в связи с тем, что в русском языке имеется некоторая двусмысленность слова «список», далее рассматриваются аспекты применения списков именно как типа, а не как элемента данных. Это значение подразумевает лишь тот факт, что структура данных «список» имеет тип «список над элементами типа `a`». Поэтому в языке Haskell нотация с квадратными скобками используется как в записи типов выражений, так и в записи структур данных.

Определение типа данных, приведённое выше, предполагает, что в языке Haskell в списке могут находиться только значения одного типа — `a`. Это может быть совершенно любой тип, даже другой список. Но факт остаётся фактом, такой гибкости и даже волюнтаризма в определении списка, которые наблюдаются в языке Lisp, в языке Haskell нет. Для группировки значений разных типов в одну структуру данных необходимо использовать рассмотренные ранее кортежи.

Математическая формула для определения типа структуры данных, являющейся списком, выглядит следующим образом:

$$\#[a_1, a_2, \ldots, a_n] : [A]$$

где $A = \#a_1 = \#a_2 = \ldots = \#a_n$.

Остаётся отметить, что кортежи и списки в языке Haskell могли бы стать универсальным средством для организации любых структур данных, как это сделано в языке Lisp, в котором списки по своей сути являются некоторым слиянием

возможностей кортежей и списков в единый формализм. Однако этого не произошло, и в языке Haskell имеются достаточно серьёзные инструменты для создания сложных типов вроде структур, перечислений и объектов.

2.2. Кратко об алгебраических типах данных

В теории программирования алгебраическим типом данных называется любой тип, значения которого являются значениями некоторых иных типов, «обёрнутыми» конструкторами алгебраического типа. Другими словами, алгебраический тип данных имеет набор конструкторов данных, каждый из которых принимает на вход значения определённых типов и возвращает значение конструируемого типа. Важное отличие конструктора данных от функции заключается в том, что конструктор не исполняется, но единственная его задача — создание значения своего типа на основе входных значений. Для работы с такими значениями используется механизм сопоставления с образцами как наиболее эффективный способ разбора значений (но это не означает, что иные механизмы работы с значениями не применимы к алгебраическим типам данных).

В языке Haskell любой тип данных, который не является примитивным, является алгебраическим. Все возможные виды значений (перечисления, объекты, структуры и т. д.) строятся при помощи конструкторов алгебраических типов данных.

Самым простым алгебраическим типом данных является список, который был рассмотрен в предыдущем разделе. Действительно, список имеет два конструктора — конструктор пустого списка и конструктор пары, первым элементом которой является значение определённого типа, а вторым — список. Так что видно, что алгебраические типы данных являются контейнерными типами — они содержат внутри себя значения других типов (или того же самого типа). То, что у списка первый конструктор не принимает на вход каких-либо параметров, не должно вводить в сомнение. Такая форма конструктора является необходимой для создания значений, которые внутри себя не содержат ничего, но являются «единичными» элементами алгебраических типов данных.

Специальными разновидностями алгебраических типов данных являются декартовы типы (они имеют только один конструктор) и перечисления (у них все конструкторы аргументов не имеют вовсе, хотя самих конструкторов может быть несколько). Также алгебраический тип данных может быть абстрактным, если

такой тип определён в некотором модуле, из которого не экспортируются конструкторы соответствующего типа, а доступ к значениям внутри алгебраического типа данных осуществляется при помощи специальных методов — селекторов. Абстракция данных подробно описана в главе 4..

Остаётся отметить, что с точки зрения синтаксически-ориентированного конструирования данных алгебраическим типом данных является размеченное объединение декартовых произведений типов. Каждое слагаемое в размеченном объединении соответствует одному конструктору, а каждый конструктор в свою очередь определяет декартово произведение типов, соответствующих параметрам конструктора. Конструкторы без параметров являются пустыми произведениями. Если алгебраический тип данных является рекурсивным, всё размеченное объединение обёртывается рекурсивным типом, и каждый конструктор типа возвращает рекурсивный тип.

Все алгебраические типы данных в языке Haskell определяются при помощи ключевого слова **data**, независимо от назначения типа.

2.2.1. Перечисления

Перечисления — это специальный вид алгебраических типов данных, который характеризуется тем, что любой его конструктор не принимает на вход ни одного аргумента. Это позволяет в рамках языка Haskell кодировать то, чем исходно являются перечисления, а именно ограниченным набором именованных констант, которые используются для улучшения внешнего вида исходных кодов, так как использование мнемонических имён для некоторых значений всегда лучше, чем использование для тех же целей числовых или иных констант.

Соответственно, обычный способ определения перечислений выглядит просто — необходимо перечислить конструкторы типа, разделив их вертикальной чертой (|). Например, так определяется булевский тип в стандартном модуле `Prelude` (детально рассматривается на стр. 108):

```
data Bool = False  -- Ложь
          | True   -- Истина
```

Понятно, что обычно перечисления используются для создания и описания множеств с ограниченным количеством элементов в них, при этом разработчику программного обеспечения требуется иметь наименования для каждого элемента такого множества. В этом случае именем множества является наименование ал-

гебраического типа данных, а названиями элементов множества — конструкторы. Вот несколько примеров определений перечислений в языке Haskell.

```
-- Определение типа для представления шахматных фигур.
data ChessFigure
  = King   -- Король
  | Queen  -- Ферзь
  | Rook   -- Ладья
  | Bishop -- Слон
  | Knight -- Конь
  | Pawn   -- Пешка


-- Определение типа для представления планет Солнечной системы.
data Planet
  = Mercury  -- Меркурий
  | Venus    -- Венера
  | Earth    -- Земля
  | Mars     -- Марс
  | Jupiter  -- Юпитер
  | Saturn   -- Сатурн
  | Uranus   -- Уран
  | Neptune  -- Нептун
  | Pluto    -- Плутон
```

Использование таких типов ничем не отличается от использования любых прочих типов данных в языке Haskell. В первую очередь все конструкторы перечислений могут использоваться в сопоставлении с образцами. Без создания дополнительных определений использование перечислений ограничено этим механизмом. Значения перечислений невозможно сравнивать друг с другом, их нельзя вывести на экран, над ними невозможно проводить какие-либо операции. Для того чтобы более полноценно использовать перечисления в программах, необходимо для соответствующего перечисления определить экземпляр некоторого класса. Этот вопрос будет самым тщательным образом раскрыт в главе 3..

Уже упомянутое перечисление `Bool` используется для представления булевских значений в программах. Однако это перечисление достаточно сильно связано с трансляторами языка, так как оно используется в конструкции `if-then-else`, а эта конструкция выполнена в языке Haskell не в виде функции, а внедрена в глубины транслятора. Поэтому перечисление `Bool` стоит немного отдельно от других перечислений, которые определяются программистом.

Другой особенностью перечислений в языке Haskell является отсутствие автоматического преобразования в целые числа, как это, к примеру, сделано в языках C или C++. Для того чтобы такая возможность была, необходимо определить соответствующее перечисление экземпляром класса `Enum`. После такого определения для перечисления становятся доступными некоторые интересные методы, к примеру методы для получения списков, состоящих из значений соответствующего перечисления, стоящих в определённом порядке.

Наконец, необходимо особо подчеркнуть, что алгебраический тип данных называется перечислением постольку, поскольку ни у одного из его конструкторов нет аргументов. Но это абсолютно не означает, что если программист задумал организовать перечисление, то он не может в алгебраический тип данных вставлять конструкторы с аргументами после того, как перечисление организовано. В наборе конструкторов в любом случае могут быть как конструкторы без аргументов, так и конструкторы с аргументами. Например, следующий алгебраический тип данных, который может определять наименования цветов, до создания конструктора `RGB` мог считаться перечислением, описывающим цвета радуги:

```
data Color = Red              -- Красный
           | Orange           -- Оранжевый
           | Yellow           -- Жёлтый
           | Green            -- Зелёный
           | Blue             -- Голубой
           | Indigo           -- Синий
           | Violet           -- Фиолетовый
           | RGB Int Int Int  -- Представление цвета в системе RGB
```

2.2.2. Простые структуры

Та же технология определения алгебраических структур данных используется и для создания типов, которые в других языках программирования носят названия «структуры» или «записи». Такие типы данных отличаются тем, что содержат некоторый набор полей, каждое из которых может быть какого-либо типа, но при этом весь такой набор рассматривается как единое и целостное значение.

Структуры предназначаются для описания некоторых сложных сущностей предметной области, у которых имеется несколько свойств, важных с точки зрения алгоритма решения задачи. Структура может быть обработана в качестве

цельного объекта, а может предоставлять доступ к своим внутренним свойствам для работы с ними. Это делает структуры достаточно серьёзным инструментом для решения самого широкого ряда задач и проблем.

В языке Haskell объединение значений различных типов в единый объект производится при помощи перечисления по порядку типов значений, которые хранятся в соответствующем алгебраическом типе данных. Само собой разумеется, что порядок перечисления типов внутренних значений важен.

Например, в каком-либо приложении, работающим с геометрическими фигурами в трёхмерном пространстве, имеется необходимость определить тип для представления точки пространства. Это можно сделать следующим образом:

```
data Point3D
  = Point3D Float Float Float
```

Эта запись означает то, что алгебраический тип данных `Point3D` имеет один конструктор `Point3D` (совершенно неважно, что эти программные сущности называются одинаково — их наименования находятся в разных пространствах имён), который «прячет» внутри себя три значения типа `Float`, каждое из которых представляет собой координату точки по одной из осей трёхмерного пространства.

Естественно, не стоит думать, что значения в наборе одного конструктора должны быть только одного типа, как показано в примере выше. Типы могут быть произвольными. Например, точка на экране в каком-нибудь графическом приложении может быть описана при помощи такого типа:

```
data Point2D
  = Point2D Int Int Color
```

Необходимо отметить, что доступ к элементам таких структур в языке Haskell достаточно сильно отличается от традиционного способа. Как видно, никаких особенных идентификаторов, дифференцирующих внутренние элементы структуры, не предусмотрено. Поэтому обращение к ним осуществляется при помощи механизма сопоставления по образцу. К примеру, в функции, которая двигает точку в трёхмерном пространстве, получить доступ к координатам точки можно следующим образом:

```
shift (Point3D x y z) dx dy dz = Point3D (x + dx) (y + dy) (z + dz)
```

Как видно, первый параметр функции является точкой в трёхмерном пространстве. Для доступа к её внутренней структуре в заголовке функции она непосредственно расписана в виде указания на конструктор данных и соответствующие поля структуры, при этом сами поля получают определённые имена, которые действительны только внутри функции. Если же необходимости в доступе нет, то всю структуру можно назвать одним именем (именем параметра функции). Например, для сдвига точки в трёхмерном пространстве только по одной из осей можно воспользоваться следующим набором функций:

```
shiftX p dx = shift p dx 0 0

shiftY p dy = shift p 0 dy 0

shiftZ p dz = shift p 0 0 dz
```

Более того, если в функции есть необходимость доступа только к некоторым элементам структуры, то можно называть именно их, оставляя остальные анонимными. Для этого используется маска подстановки «любое значение», которая в языке Haskell записывается символом подчёркивания (_). Так, к примеру, функции для получения координат заданной точки в трёхмерном пространстве выглядят так:

```
getX (Point3D x _ _) = x

getY (Point3D _ y _) = y

getZ (Point3D _ _ z) = z
```

Остаётся отметить, что при объявлении элементов структур можно ограничивать применение методов ленивых вычислений, возводя флажок строгости для требуемых элементов. Другими словами, если элемент структуры данных определён с флажком строгости, то при использовании конструктора значение переданного параметра вычисляется всегда, даже если оно и не требуется.

Флажок строгости имеет вид (**!**) и ставится непосредственно перед типом элемента структуры. Например:

```
data Point3D
  = Point3D !Float !Float !Float
```

2.2.3. Именованные поля

При определении структур можно всем их элементам давать определённые наименования, и, более того, это автоматически сгенерирует методы доступа на чтение и на запись к соответствующим элементам. Для этого в языке Haskell введено такое понятие, как именованное поле.

Пусть, к примеру, имеется тип данных, который описывает положение грузового контейнера на контейнерной площадке (для единообразия в этом разделе рассматриваются только геометрические задачи). Этот тип можно описать следующим образом:

```
data ContainerPlace
  = CP Int Int Int Int
```

Грамотный разработчик немного модифицировал бы взаимное расположение элементов данного определения, чтобы работать с ним было бы удобнее. В результате он мог бы получить что-нибудь вроде:

```
data ContainerPlace
  = CP Int -- Ряд
       Int -- Секция
       Int -- Место
       Int -- Ярус
```

Но такая запись не сильно поможет при дальнейшей разработке прикладных функций для оперирования элементами такого типа. Ведь, к примеру, для доступа на чтение и на запись для такого типа пришлось писать бы ещё восемь примерно одинаковых функций:

```
getCPRow (CP row _ _ _) = row


getCPSection (CP _ section _ _) = section
```

и т. д.

Видно, что поля данных имеют в данном примере одинаковые типы (а в общем случае многие поля данных имеют одинаковые типы), что может привести к серьёзным логическим ошибкам, так как можно спутать два поля друг с другом, особенно если полей много. Кроме того, как и для доступа, так и для записи значения в каждое из полей необходимо создавать специальные функции (так называемые `get-` и `set`-функции). Этот путь достаточно трудоёмкий (необходимо написать по две функции для каждого элемента), но в то же время легко формализуемый. Поэтому в языке Haskell имеется возможность автоматического построения функций доступа к элементам структур, а для этого необходимо просто использовать именованные поля данных. В этом случае приведённое выше определение типа будет выглядеть примерно так:

```
data ContainerPlace
  = CP
    {
      row     :: Int,
      section :: Int,
      place   :: Int,
      level   :: Int
    }
```

В этом случае транслятор языка автоматически построит функции для доступа к одиночным значениям данной структуры, которые имеют такие же наименования, как и приведённые в определении типа.

Таким же самым образом можно обновлять одиночные значения:

```
store2D container r s p = container{row = r, section = s, place = p}
```

Достаточно в фигурных скобках после параметра соответствующего типа перечислить после запятой «присваивания» новых значений элементам через функции доступа, и сам объект обновит своё состояние. При этом необходимо отметить, что использование традиционного подхода вместе с этим вполне возможно, то есть разработчик программ самостоятельно может написать функции для доступа. Но необходимости в этом нет никакой. Более того, разработчик может использовать и механизм сопоставления с образцами и создания копии объекта с изменёнными параметрами вручную — использование именованных полей лишь даёт новые возможности по упрощению процесса программирования.

Обновление состояния объекта производится через присваивание именованному полю нового значения. Это положение не должно смущать — никаких деструктивных действий не производится, ибо они запрещены в чистом функциональном программировании. Слово «присваивание» употреблено здесь образно, только для наименования операции (=). На самом деле происходит создание нового объекта, в котором содержится новое значение заданного поля, но при этом надо понимать, что вполне возможно, в некоторых трансляторах в качестве оптимизирующего метода может быть реализовано и деструктивное присваивание, когда значение поля в старом объекте меняется на новое. Естественно, это может производиться только в случае, когда старый объект уже не нужен. О старом же объекте в любом случае, как это принято, позаботится сборщик мусора.

2.3. Синонимы типов

Синонимы типов используются для упрощения записей имён типов. Иногда типы некоторых выражений являются достаточно сложными. Это может происходить при использовании алгебраических типов данных, в конструкторах которых используется большое количество типов, либо в типах функций. Для того чтобы исключить многократное повторение одинаковых длинных последовательностей, можно определять синонимы типов. Эта техника также полезна и в качестве объявления единственной программной сущности, для изменения которой в случае необходимости вносить изменения было бы можно только в одном месте программного кода.

Синонимы определяются при помощи ключевого слова **type**. После него записывается наименование синонима типа с заглавной буквы и перечисляются переменные типов, если это необходимо. После записывается символ (=) и выражение, определяющее тип. Если используются переменные типов, то все они должны использоваться в выражении типа. Например, следующие синонимы можно определить для разных нужд:

```
type List a = [a]

type ListInteger = [Integer]

type Function a b = a -> b
```

Использование синонимов типов имеет определённые ограничения. Например, их конструкторы невозможно частично применять к своим аргументам, в то время как конструкторы алгебраических типов данных можно. Рекурсивные алгебраические типы данных допустимы, а синонимы типов нет. Взаимно рекурсивные синонимы типов возможны только при использовании промежуточных алгебраических типов данных. Например, определения:

```
type Record a = [Circular a]

data Circular a = Tag [Record a]
```

возможны, в то время как определения:

```
type Record a = [Circular a]

type Circular a = [Record a]
```

вызовут ошибку.

Синонимы типов представляют собой удобный, но строго синтаксический механизм, который делает сигнатуры программных сущностей более читабельными. Синоним и его определение, — полностью взаимозаменяемы. Единственное место, где нельзя использовать синоним типа вместо самого типа, который заменяется синонимом — определения экземпляров классов. В них можно использовать только сами типы.

Однако в языке Haskell имеется один специальный синоним, который поддерживается на уровне трансляторов. Этот синоним — `String`, который определён следующим образом:

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

Да, строки в языке Haskell являются просто списками символов (тип `Char`), ни больше ни меньше. Единственное, чем этот синоним отличается от прочих — особая поддержка на уровне синтаксиса. В целях экономии усилий и нервов разработчиков программного обеспечения создатели языка Haskell позволили записывать строки в виде собственно строк, заключённых в кавычки (`"`). Ибо было бы очень неудобно записывать строки примерно следующим образом:

```
['H', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!']
```

Эта запись абсолютно тождественна такой: `"Hello, world!"`

2.4. Параметрический полиморфизм

При использовании параметрического полиморфизма конструкторы алгебраических типов данных могут быть определены достаточно обобщённо для того, чтобы тип содержал внутри себя значения совершенно различных типов. Эта техника уже неоднократно неявно использовалась в приводимых примерах, когда вместо конкретных типов данных внутри конструктора использовались некоторые переменные, обычно обозначаемые строчными буквами латинского алфавита, начиная с его начала, например `a`, `b` и т. д.

Вместо таких переменных может быть подставлен любой тип, чтобы конкретизировать полиморфный алгебраический тип данных. Интуитивно это должно быть понятно: такой тип данных, как, к примеру, список, может содержать внутри себя данные любого типа. Как это записать? Конечно, используя переменные типов:

```
data List a
  = Nil
  | Cons a (List a)
```

При определении полиморфного алгебраического типа данных необходимо помнить об одном непререкаемом правиле. Если какой-либо (или какие-либо) конструктор данных использует переменные типа, то все эти переменные должны быть перечислены после наименования типа перед символом (`=`). Это правило ограничивает использование переменных типа с одинаковым наименованием в разных смыслах в разных конструкторах одного и того же алгебраического типа данных. Поэтому при конкретизации типа, представленного переменной, она заменяется конкретным типом везде, где имеет вхождение.

Язык Haskell поддерживает различные виды полиморфизма данных и функций. В этом разделе был рассмотрен только параметрический полиморфизм алгебраических типов данных, но в дальнейшем изложении будут упомянуты иные виды полиморфизма: ограничение контекстом использования и перегрузка имён функций (полиморфизм *ad-hoc*). Все они будут кратко описаны в соответствующих разделах главы 3..

2.5. Типы функций

Необходимо отметить, что понимание функций в функциональном программировании достаточно серьёзно отличается от их восприятия в императивных языках. Дело в том, что в рамках функциональной парадигмы функция является программной сущностью, которая обладает типом, является объектом, над которым можно производить действия: во-первых, передавать в другие функции в качестве фактического значения; а во-вторых, возвращать в качестве вычисленного значения. Такое положение вещей является прямым следствием из принятой модели типизации языка Haskell (статическая типизация Хиндли-Милнера), в рамках которой у функций имеются типы.

2.5.1. Функции как программные сущности с типом

Каждая функция в языке Haskell имеет определённый тип. Такой тип функций уже несколько раз встречался в тексте этого справочника в описаниях сигнатур функций, которые обычно стоят перед определением функции. Строка, в которой имеются символы (`::`), и есть сигнатура, то есть определение (и ограничение) типа функции.

Для формализации типов функций используется единственная операция: (`->`). Она связывает типы аргументов (формальных параметров) и тип возвращаемого результата. Тип функции определяется на основании следующих простых правил:

1) Константная функция без формальных параметров имеет тип, равный типу возвращаемого результата. Это вполне логично, ибо такая функция по своей сути является просто константой.

2) Функция с одним аргументом имеет тип `(ArgType -> ResType)`, где `ArgType` — тип единственного аргумента, а `ResType` — тип возвращаемого результата.

3) Функция с n аргументами имеет тип `(Arg1Type -> (... -> (ArgNType -> ResType)...))`. Эта запись показывает, что операция (`->`) правоассоциативна, а поэтому лишние скобки можно опустить для удобства представления и восприятия: `Arg1Type -> ... -> ArgNType -> ResType`.

В качестве примера сигнатур функций можно привести такие сигнатуры из стандартного модуля `Prelude` для некоторых интересных функций и бинарных операций:

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

flip :: (a -> b -> c) -> b -> a -> c

($) :: (a -> b) -> a -> b
```

Операция (`.`) используется для композиции функций в полном соответствии с математическим определением термина «композиция» (подробно описывается на стр. 171). Функция `flip` получает на вход функцию двух аргументов и два значения, а возвращает значение переданной на вход первым параметром функции на переданных же аргументах, но в обратном порядке (описывается на стр. 133). Операция (`$`) является синонимом операции применения функции к своим аргументам, но имеет самый низкий приоритет, а потому может использоваться для сшивания последовательных применений разных функций для того, чтобы использовать поменьше скобок (детально описывается также на стр. 171).

Как видно, в приведённых типах функций очень важное значение имеет заключение определённых частей типов в круглые скобки. Уже было сказано, что в силу правой ассоциативности операции (`->`) лишние скобки можно опустить. Но если имеется необходимость расставить скобки иным образом, это необходимо указывать явно. В каких же случаях необходимо такое явное указание? В случаях, если формальными аргументами функций являются другие функции, ибо операция (`->`) используется только для создания функциональных типов. Этот вопрос самым детальным образом прорабатывается ниже в разделе 2.5.3..

Здесь остаётся отметить, что сигнатура типа функции является ничем иным, как ограничением на тип. Символ (`::`) может использоваться именно как ограничитель типов, при помощи которого можно ограничить тип функции (или любой иной программной сущности). Это иногда необходимо делать в целях оптимизации, а можно и ограничивать типы для решения определённой задачи. Всё это связано с тем, что механизм вывода типов, как это доказано теоретически, выводит наиболее общий тип, стремясь сделать его полиморфным.

В качестве примера можно привести достаточно надуманную функцию, которая возводит первый аргумент в степень, показатель которой представлен вторым аргументом. Эта функция может быть определена следующим образом:

```
power x 0 = 1
power x y = x * power x (y - 1)
```

Если записать это определение без сигнатуры, то транслятор языка Haskell автоматически выведет тип этой функции в виде:

```
power :: (Num a, Num b) => a -> b -> a
```

Это значит, что типы обоих аргументов (`a` и `b` соответственно) могут не совпадать, но при этом оба типа должны быть экземплярами класса `Num`, то есть таких величин, над которыми можно производить арифметические операции (подробно о классах типов и их экземплярах написано в главе 3.).

Однако если имеется необходимость ограничить использование функции только типом `Int` (ограниченные целые числа) для обоих аргументов, то можно явно записать:

```
power :: Int -> Int -> Int
power x 0 = 1
power x y = x * power x (y - 1)
```

В этом случае транслятор языка Haskell будет в своей работе использовать только этот тип функции `power`, что позволит, в свою очередь, довольно существенно её оптимизировать.

2.5.2. Каррирование и частичное применение

Общий вид типа функции в виде `(Arg1Type -> (... -> (ArgNType -> ResType)...))` наводит на мысль о том, что каждая функция может рассматриваться в качестве функции одного аргумента, при этом результатом исполнения такой функции будет тоже функция. Действительно, если мысленно обозначить в представленном выражении типа часть `(... -> (ArgNType -> ResType)...)` как `ResType*`, то типом исходной функции будет выражение `(Arg1Type -> ResType*)`, а это в силу определения типа функции есть тип функции одного аргумента.

И действительно, все функции в языке Haskell воспринимаются именно таким образом. Каждая неконстантная функция может быть рассмотрена в качестве функции одного аргумента, в результате работы которой получается новая функция, которая ожидает на вход оставшиеся аргументы. Этот процесс называется частичным применением, а функции в таком понимании называются каррированными.

Суть частичного применения можно пояснить следующим образом. Пусть имеется некоторая функция `f`, которая принимает на вход n аргументов:

```
f x1 x2 ... xn = ...
```

И пусть имеется некоторая функция `g`, которая определена следующим образом:

```
g = f 0
```

В теле определения функции `g` имеет место частичное применение функции `f` с фактическим значением первого аргумента равным 0. Что происходит в этом случае? Транслятор языка Haskell выполняет создание функции `g` на основании тела функции `f`, подставляя в нём на места вхождения формального параметра `x1` поданное на вход фактическое значение 0. Результатом будет являться функция с $(n-1)$ аргументами, которая и будет являться определением функции `g`.

Конечно, на самом деле этот процесс гораздо сложнее, но в общих чертах суть частичного применения может быть понята именно на таком примере. Язык Haskell полностью поддерживает эту замечательную технологию, поэтому при программировании на нём важно помнить, что сколько бы аргументов у функции не было, её можно использовать для частичного применения. При этом надо понимать, что частичное применение может производиться не только с одним аргументом, но и с любым другим количеством оных. Если число фактических значений, которые переданы на вход функции, меньше количества формальных параметров, то всегда имеет место частичное применение.

Именно по этой причине в функциональном программировании принята бесскобочная запись применения функции к своим аргументам. Этим такая запись отличается от традиционной, принятой в математике, где аргументы функции заключаются в круглые скобки и разделяются запятыми. С точки зрения функционального программирования все математические функции, записанные в та-

ком виде, являются функциями одного аргумента, при этом аргументом является кортеж, размерность которого равна количеству аргументов функции с математической точки зрения. Такой взгляд на функции также имеет место в функциональном программировании, а функции такого вида называются, соответственно, некаррированными.

В стандартном модуле `Prelude` определена пара функций для применения некаррированных аргументов к каррированным функциям и наоборот (для функций двух аргументов). Под некаррированным аргументом здесь, само собой, понимается кортеж значений. Эти функции: `curry` и `uncurry` соответственно (детально описываются на стр. 129 и стр. 166):

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


uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f p = f (fst p) (snd p)
```

При их частичном применении (когда на вход функциям подаётся только другая функция) результатом будет функция, которая выполняет те же самые действия, что и исходная функция, но имеет противоположный признак каррированности (и это, кстати, видно по записи типов функции: в них скобки расставлены не совсем обычным образом). Например, операция `(*)`, как и любая иная, является каррированной. Но применение `uncurry (*)` делает её некаррированной, а потому на вход это применение начинает ожидать пару значений:

```
> (uncurry (*)) (10, 2)
```

После исполнения этой команды в интерпретаторе результатом, естественно, будет значение `20`.

2.5.3. Функции высшего порядка

Многие примеры, рассмотренные выше, имеют одну интересную особенность. В функциях в качестве формальных параметров ожидаются другие функции. На это указывают и типы таких функций, и сам их смысл. И действительно, в языке Haskell сплошь и рядом функции могут передаваться в качестве входных параметров в другие функции. Это — достаточно естественный процесс, и связан

он опять-таки с тем, что функции имеют типы, которые определяются таким образом, как описано ранее.

Под термином «функция высшего порядка» понимается функция, которая может принимать на вход в качестве фактических параметров другие функции. Иногда этот термин также относят к функциям, которые могут возвращать другие функции в качестве результата своей работы. Но, как описано чуть ранее, в этом случае функцией высшего порядка можно назвать любую каррированную функцию с количеством формальных параметров больше одного.

Можно привести в качестве примера наиболее интересные функции из стандартного модуля `Prelude`, которые являются функциями высшего порядка:

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

Функция `map` (более детально описывается на стр. 356) принимает на вход функцию и список, а возвращает список, который состоит из результатов применения заданной функции к элементам заданного списка. Эта функция может использоваться для быстрого получения списка значений некоторой функции на элементах некоторого списка.

Другие известные функции высшего порядка — различные виды свёрток списка значений в одно. Например, левоассоциативная свёртка определяется следующим образом (подробно описывается на стр. 253):

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

Эта функция «сворачивает» заданный список при помощи заданной функции, используя в качестве начального значения второй аргумент. При помощи этой функции, а также функции для правоассоциативной свёртки (`foldr`, описана на стр. 254) определяются многие частные виды свёртки, в том числе и функции для суммирования элементов списка, получения их произведения и т. д.

В общем виде свёртка осуществляет последовательное применение заданной функции ко всем элементам списка. Левая или правая ассоциативность свёртки определяет способ, при помощи которого применяется функция. Для функций, расстановка скобок для которых в последовательном ряду применений важна (например, для операций вычитания (`-`) или деления (`/`)), ассоциативность свёрт-

ки также важна. Пояснить ассоциативность свёртки можно следующими рассуждениями:

Для левоассоциативной свёртки вызов функции `foldl`

```
foldl f z [x1, x2 .. xn]
```

тождественнен последовательному применению (эта запись не имеет смысла на языке Haskell, троеточия используются в качестве замещающего символа):

```
(...((z `f` x1) `f` x2) ... `f` xn)
```

Правоассоциативная свёртка работает иначе. Вызов функции `foldr`

```
foldr f z [x1, x2 .. xn]
```

опять-таки тождественнен последовательному применению:

```
(x1 `f` (x2 `f` ... (xn `f` z) ... ))
```

Наконец, осталось привести в качестве примера ещё одну очень интересную функцию из стандартного модуля `Prelude`. Это функция `zipWith`, которая принимает на вход функцию и два списка, а возвращает список, состоящий из значений функции, полученных при помощи её применения к значениям исходных списков. Её определение выглядит следующим образом (а более детально описывается на стр. 269):

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = (f x y):(zipWith f xs ys)
zipWith _ _      _      = []
```

Эту функцию можно рассматривать в качестве аналога функции `map` для двух списков, но её применение находится в несколько иной плоскости. Например, её можно использовать для быстрого сложения элементов двух списков либо для создания списка пар, первым элементом которых являются значения из первого списка, а вторым — из второго соответственно.

Таким образом, понимание сути функций высшего порядка позволяет успешно использовать их на практике, что, в свою очередь, помогает эффективно и просто решать разнообразные задачи, решения которых методами императивного программирования занимают многие и многие строки кода.

Глава 3.

Классы типов и экземпляры классов

Третий тип программных сущностей языка Haskell — классы типов. Здесь имеется одна ловушка, в которую могут попасть те, кто применяет на практике объектно-ориентированный стиль. В языке Haskell, а также в некоторых схожих с ним функциональных языках программирования под термином «класс» понимается совсем не то, что подразумевает объектно-ориентированное программирование. Если в рамках объектно-ориентированного подхода под классом понимается тип данных, то в функциональном программировании (а вернее, в модели статической типизации, принятой в языке Haskell) класс типов — это, скорее, интерфейс работы с данными.

3.1. Класс как интерфейс

Итак классы типов в языке Haskell больше всего похожи на интерфейсы в таких языках, как Java или IDL. Действительно, классы типов описывают наборы методов (функций), которые применимы для работы с теми или иными типами данных, для которых объявлены экземпляры заданных классов. Чтобы не быть абстрактно-голословным, достаточно рассмотреть некоторые примеры использования классов типов в стандартном модуле `Prelude`.

Один из самых простых классов, которые описаны в стандартном модуле `Prelude`, это класс `Eq` — класс типов сравнимых величин. Он определяется следующим образом:

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


  x == y = not (x /= y)
  x /= y = not (x == y)
```

Это — определение класса с именем `Eq`, экземпляром которого может быть некоторый тип `a`. Внутри этого класса определяются два метода в виде бинарных операций: (==) и (/=). Эти операции имеют заданный тип. Более того, ниже приведены выражения этих операций друг через друга, что позволяет транслятору языка Haskell самостоятельно вычислять методы класса, которые выражены через другие методы или внешние функции, в случаях, когда разработчик программного обеспечения явно не указал реализацию заявленных методов для некоторого типа.

Если рассматривать абстрактно, то по подобной декларации невозможно сказать, для чего предназначен этот класс `Eq`. Конечно, если рассматривать семантику его названия, а также сигнатуры функций, то можно предположить, что все типы, которые являются экземплярами данного класса, являются типами сравнимых величин, то есть таких значений, которые можно сравнивать друг с другом при помощи операций (==) и (/=).

Однако, как видно, семантика класса нигде не указана. Да и нет в синтаксисе языка Haskell специальных средств для указания семантики классов и их экземпляров. Поэтому каждый специалист, изучающий язык Haskell, должен помнить, что классы типов — это всего лишь абстрактные декларации верхнего уровня, которые группируют внутри себя методы. Эта группировка используется для того, чтобы указать набор интерфейсных методов над определёнными типами данных. Семантика же класса описывается внешним способом — комментариями к классу или в документации. Сам транслятор языка Haskell, таким образом, не в курсе, для чего используется тот или иной класс типов.

Само собой разумеется, что в перечислении методов класса можно указывать не только бинарные операции, но и вообще любые функции. Единственное ограничение на метод класса — в его сигнатуре обязательно должна присутствовать

переменная типа, использованная в заголовке декларации после имени класса. В рассмотренном примере класса `Eq` такой переменной является переменная `a`. Например, можно рассмотреть декларацию класса `Ord` из стандартного модуля `Prelude`:

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

  compare x y | x == y    = EQ
              | x <= y    = LT
              | otherwise = GT

  x <= y = compare x y /= GT
  x < y  = compare x y == LT
  x >= y = compare x y /= LT
  x > y  = compare x y == GT

  max x y | x <= y    = y
          | otherwise = x

  min x y | x <= y    = x
          | otherwise = y
```

Этот класс уже более сложный, чем класс `Eq`. В нём определяются 7 методов, четыре из которых являются бинарными операциями. Для каждого метода описывается сигнатура. Ниже, после декларации сигнатур методов, опять приводится выражение методов друг через друга. Эта секция необязательна, но если методы можно выразить друг через друга, хорошим тоном в определении классов является указание таких взаимосвязей. Это позволяет снять некоторую часть работы с тех программистов, кто будет реализовывать экземпляры классов.

Директива `(Eq a) =>` будет рассматриваться ниже в разделе 3.2.. Здесь необходимо лишь упомянуть, что эта директива называется «контекстом».

Для чего необходим класс `Ord`? Транслятор языка Haskell никогда не ответит на этот вопрос. Он только лишь сможет перечислить методы, которые необходимо реализовать для некоторого типа данных, чтобы этот тип являлся экземпляром данного класса. Поэтому необходимо повториться о том, что семантика класса описывается вне программ на языке Haskell. Только программист, описавший класс `Ord`, скажет, что этот класс является классом типов упорядоченных величин. Впрочем, об этом можно догадаться по наименованию его методов.

Другими словами, классы типов — это такая сущность, которая гарантирует для типов, являющихся экземплярами данного класса, наличия определённых методов, которые обрабатывают значения этих типов. Более того, для всех таких типов наименования методов будут одинаковыми — они будут иметь названия, определённые в классе. Собственно, наличие класса типов и является гарантией для функций о том, что определённые в классе методы существуют для обрабатываемых значений (в дальнейшем такие функции для классов будут называться «прикладными функциями»).

То есть классы используются в языке Haskell для полиморфизма ad hoc — перегрузки имён функций. Действительно, достаточно определить экземплярами некоторого класса пару типов, чтобы одинаковые функции (методы класса) могли работать над значениями этих типов. Это — естественный способ перегружать имена функций и пользоваться непараметрическим полиморфизмом.

С другой стороны, классы — это объединение типов в группы по назначению. В стандартном модуле `Prelude`, к примеру, описаны классы для целочисленных значений, для дробных значений и т. д. Все эти классы используются для того, чтобы можно было использовать соответствующие операции над различными типами данных. И все такие типы данных являются в чём-то схожими друг с другом, поскольку являются экземплярами одинаковых классов.

Для того чтобы закрепить понимание класса типов как интерфейса доступа к значениям типов данных, можно попробовать реализовать свой собственный класс. Например, в стандартном модуле `Prelude` нет класса, который бы отвечал за значения, которые потенциально могут быть логическими, то есть использоваться в той или иной логике. Такой класс можно определить следующим образом:

```
class Logic a where
  neg    :: a -> a      -- Отрицание
  (<&&>) :: a -> a -> a -- Конъюнкция
  (<||>) :: a -> a -> a -- Дизъюнкция
  (<~|>) :: a -> a -> a -- Исключающее ИЛИ
  (<=>>) :: a -> a -> a -- Импликация
  (<==>) :: a -> a -> a -- Эквивалентность
```

Класс `Logic` определяет 6 методов, из которых пять являются бинарными операциями. Кстати, для методов класса, являющихся по смыслу операциями, можно пользоваться определением приоритета и ассоциативности при помощи ключевых слов **infix**, **infixl** и **infixr** (см. подраздел 1.1.4.). Это можно делать потому, что методы класса являются декларациями верхнего уровня, их видно извне класса — это такие же функции, как и прочие, определяемые разработчиком. Поэтому при определении методов класса также необходимо помнить, что такие методы не могут иметь одинаковые наименования с функциями, определяемыми вне классов.

Определить приоритет и ассоциативность перечисленных в классе `Logic` бинарных операций можно следующим образом:

```
infixl 7 <&&>, <~|>
infixr 6 <=>>, <==>
infixl 5 <||>
```

Кроме того, все бинарные операции, используемые в формальной логике, можно связать друг с другом. Для этого можно использовать следующие определения методов друг через друга:

```
x <&&> y = neg (neg x <||> neg y)
x <||> y = neg (neg x <&&> neg y)
x <~|> y = (neg x <&&> y) <||> (x <&&> neg y)
x <=>> y = neg x <||> x <&&> y
x <==> y = (neg x <&&> neg y) <||> x <&&> y
```

Таким образом, класс `Logic` является классом типов, значения которых могут быть использованы в качестве логических значений. Например, таким типом является тип `Bool`. Другим типом может быть тип `Float`, который может использоваться для представления значений бесконечнозначных логик. В разделе 3.3.

этот класс будет использоваться для объяснения того, как можно определить экземпляры классов.

3.2. Контекст и прикладные функции

Всё вышеперечисленное так и не позволило ответить на самый главный вопрос: для чего нужны классы, особенно принимая во внимание их столь абстрактную природу? Действительно, какой смысл создавать классы, если их семантика определяется вне программы? Только ради перегрузки имён функций? Но для этого можно было использовать другие, более понятные механизмы — например, перегрузка имён функций в языке C не использует никаких классов и очень даже понятна.

Однако основное использование классов типов в языке Haskell определяется системой типизации, которая используется в этом языке программирования. Читатель наверняка уже неоднократно замечал, что в сигнатурах функций используются записи, подобные таким: `(Eq a) =>`, `(Ord a, Num b) =>` и т. д. Эти директивы, как уже было сказано, называются контекстом использования переменных типов. Они обозначают, что в типе, указанном после данной директивы, переменные типов должны быть экземплярами соответствующих классов: класса `Eq` или класса `Ord` и класса `Num`.

Другими словами, запись вида

```
gcd :: Integral a => a -> a -> a
```

означает, что переменная типа `a` в этой записи может являться только экземпляром класса `Integral`.

В модели типизации языка Haskell классы типов используются в качестве ограничений на значения переменных типов. Такие ограничения позволяют не рассматривать всё множество типов данных при вычислении типов программных сущностей, но ограничиваться только рассмотрением классов типов. Впрочем, часто сами ограничения вычисляются механизмом вывода типов, что впоследствии позволяет уже транслятору языка использовать такие ограничения на рассматриваемые типы данных при создании функций, поддерживающих параметрический полиморфизм.

Таким образом, если в сигнатуре прикладной функции имеется контекст, то такая функция может использоваться только с соответствующими типами дан-

ных. Это позволяет создавать функции, которые «не знают» о природе типов своих параметров, но, тем не менее, могут совершать над значениями таких типов определённые операции, описанные в классе. В разделе 3.3. будут приведены примеры создания прикладных функций для значений неопределённых типов, о которых лишь известно, что они являются экземплярами определённых классов.

Остаётся отметить, что контекст можно использовать не только в определениях сигнатур функций, но и в определении алгебраических типов данных (там, где используются параметрические переменные типов), а также в определениях классов. При определении типов значение контекста абсолютно такое же, как и при определении типов функций, — ограничения на использование параметрических переменных.

Собственно, такое же значение и при использовании контекста в определениях классов, но здесь имеется один нюанс.Рассмотрим, к примеру, определение класса `Num` из стандартного модуля `Prelude`, определяющего типы величин, которые могут использоваться в качестве чисел, над которыми нельзя проводить операцию деления (`/`):

```
class (Eq a, Show a) => Num a where
  (+)         :: a -> a -> a
  (-)         :: a -> a -> a
  (*)         :: a -> a -> a
  negate      :: a -> a
  abs         :: a -> a
  signum      :: a -> a
  fromInteger :: Integer -> a
  fromInt     :: Int -> a


  x - y    = x + negate y
  fromInt  = fromIntegral
  negate x = 0 - x
```

Как видно, в этом определении использован контекст `(Eq a, Show a) =>`. Это значит, что использованная параметрическая переменная типа `a` должна обозначать такие и только такие типы данных, которые являются экземплярами одновременно и класса `Eq`, и класса `Show`. Это вполне понятно и следует из определения понятия «контекст». Но здесь этот контекст распространяется на все методы класса `Num`.

Иногда говорят, что использование контекста при определении классов являет собой наследование классов. Действительно, в приведённом выше примере класс `Num` можно понимать в качестве класса-потомка от классов `Eq` и `Show`. Правда, принимая во внимание всё описанное в этом разделе относительно понимания класса как ограничителя на тип данных, это не совсем то наследование, которое рассматривается в объектно-ориентированном программировании. Но общие моменты, тем не менее, имеются.

Если рассмотреть какой-либо тип класса `Num`, то для него должны иметься все перечисленные методы этого класса. Но контекст в определении класса `Num` также обязывает этот тип быть экземплярами классов `Eq` и `Show`, а это значит, что для него должны быть определены соответствующие методы этих классов. Собственно, то же самое имеется и в объектно-ориентированном понимании наследования — в классах-потомках определены методы из базовых классов. Правда, обычно методы базовых классов в объектно-ориентированных языках определяются для классов-потомков автоматически, а в функциональном программировании определять экземпляры классов из контекста необходимо самостоятельно (не всегда, конечно, но об этом в разделе 3.5.).

Так что видно, что кроме полиморфизма в функциональном программировании также поддерживается концепция наследования. Более того, инкапсуляция (то есть абстракция типов данных) также может быть реализована в языке Haskell, о чём будет достаточно написано в разделе 4.2.. Сейчас же пришло время перейти к изучению четвёртого типа программных сущностей языка Haskell — экземпляров классов, о которых уже много раз говорилось в этой главе, но ни разу не расшифровывалось, что же это такое.

3.3. Экземпляр — связь между типом и классом

Система типизации в языке Haskell предоставляет поистине удивительную технологию, которая позволяет создавать дополнения к интерфейсам (наборам функций, оперирующих с заданным типом данных) «на лету». Эта технология основывается на понятии экземпляра класса.

Экземпляры классов очень важны, поскольку отвечают за связь классов и типов, что, в конечном итоге, ведёт к тем свойствам системы типизации, реализованной в языке Haskell, которые были кратко описаны в предыдущих разделах.

Если рассматривать объектную модель программных сущностей языка Haskell, то в ней экземпляры классов будут представлены связью (или параметризованной связью) между классами и типами. Действительно, много уже было сказано про то, что классы типов и типы данных связываются друг с другом для того, чтобы определить для значений заданного типа определённое классом поведение (методы, описываемые классом). Собственно, экземпляр класса и есть такая связь.

В языке Haskell экземпляр класса определяется практически так же, как и сам класс, за исключением того, что для этого используется ключевое слово **instance**, а вместо сигнатур методов после декларации экземпляра приводятся их реализации для конкретного типа. Например, следующим образом в стандартном модуле `Prelude` определён экземпляр класса `Eq` (этот класс определяет типы сравнимых величин) для списков:

```
instance Eq a => Eq [a] where
  []     == []     =  True
  (x:xs) == (y:ys) =  x == y && xs == ys
  _      == _      =  False
```

Как видно, в определении экземпляров также можно использовать контекст. И действительно, сравнивать списки значений можно тогда и только тогда, когда можно сравнивать сами значения, заключённые в списки. Об этом и говорит контекст в определении этого экземпляра. Собственно, этот пример и показывает, как необходимо определять экземпляры. После ключевого слова **instance** записывается наименование класса (после контекста, если он необходим), затем — наименование типа. В этом случае данный тип заменит собой параметрическую переменную типа в декларации класса, и все методы класса получат определённый тип, зависящий от типа, для которого определяется экземпляр. То есть если у операции (`==`) при объявлении внутри класса `Eq` был заявлен тип `a -> a -> Bool`, то при определении экземпляра для типа `[]` произойдёт конкретизация параметрической переменной, в связи с чем операция (`==`) для типа `[]` будет иметь тип `[a] -> [a] -> Bool` (здесь, конечно, имеется коллизия имён параметрических переменных, но читать этот пример необходимо так, как будто бы в декларации класса и его экземпляра используются разные переменные `a`).

После имени типа в определении экземпляра записывается ключевое слово **where**, после которого уже следует перечисление реализаций методов для кон-

кретного типа данных. Тут необходимо отметить, что реализовывать можно только те методы, которые необходимы (обычно это особо оговаривается в комментариях к определению класса). Те методы, которые имеют выражение через другие методы и функции по умолчанию, можно не определять — транслятор языка Haskell построит их самостоятельно.

В приведённом выше примере как раз не реализуется операция (`/=`) для типа `[]`. Транслятор построит её самостоятельно, взяв за основу определение этой операции из класса:

```
x /= y = not (x == y)
```

Тут видно, что просто инвертируется результат выполнения операции (`/=`), поэтому для списков полное определение этой операции будет сгенерировано следующим образом:

```
(/=) :: Eq a => [a] -> [a] -> Bool
xs /= yx = not (xs == ys)
```

Хотя, конечно же, можно было бы определить эту операцию подобно операции (`==`):

```
(/=) :: Eq a => [a] -> [a] -> Bool
[]     /= []     =  False
(x:xs) /= (y:ys) =  x /= y || xs /= ys
_      == _      =  True
```

Вероятно, такое определение будет работать немного быстрее (не будет надобности в одном вызове функции `not` в самом крайнем случае, когда два переданных на вход списка отличаются друг от друга только последним элементом).

Остаётся упомянуть, что сама программная сущность «экземпляр класса» в языке Haskell безымянна, а потому для произвольной уникальной пары (класс, тип) можно определить только один экземпляр. После дальнейшего примера будет рассказано про то, как обойти это ограничение.

3.3.1. Экземпляры класса `Logic`

В разделе 3.1. был представлен класс `Logic`, который можно использовать для описания методов работы с типами величин, представляющих собой логические значения.

В этом классе определены классические методы из алгебры логики, а также несколько дополнительных, вроде импликации и эквивалентности (при желании, конечно, можно было добавить в определение этого класса и методы для вычисления таких операций, как «стрелка Пирса» или «штрих Шеффера»). Само собой разумеется, что все эти методы применимы к булевским значениям истинности, которые в языке Haskell представляются типом `Bool`. Для того чтобы можно было применять такие методы на значениях типа `Bool`, необходимо определить его экземпляром класса `Logic`:

```
instance Logic Bool where
  neg True  = False
  neg False = True

  True <&&> True = True
  _    <&&> _    = False
```

Здесь определяются только две главные операции — отрицание и конъюнкция. Понятно, что остальные операции выражаются через эти две. Для класса `Logic`, в свою очередь, определены выражения по умолчанию для остальных операций, кроме приведённых для типа `Bool`. В этом заключается помощь разработчика класса тем программистам, кто создаёт экземпляры этого класса.

Теперь можно использовать любые методы класса `Logic` для определения собственных функций, работающих с логическими значениями (независимо от конкретного типа данных, при помощи которого представляются эти самые значения истинности). Такие функции называются прикладными. Например, раз в классе `Logic` не определены операции «стрелка Пирса» и «штрих Шеффера», то их можно определить при помощи таких прикладных функций:

```
peirce :: Logic a => a -> a -> a
peirce x y = neg (x <||> y)


schaeffer :: Logic a => a -> a -> a
schaeffer x y = neg (x <&&> y)
```

Как видно, в этих прикладных функциях нет никакого указания на то, какой тип данных они обрабатывают. Имеется просто указание, что тип таких данных обязательно должен быть экземпляром класса `Logic`, и это будет гарантией того, что для значений этого типа определены функции `neg`, (`<&&>`) и (`<||>`), а пото-

му результат выполнения этих функций можно будет получить и использовать в своём вычислительном процессе. Поэтому такой механизм связывания классов и типов данных является достаточно гибким для того чтобы решать подобные задачи.

Дополнительно это положение можно пояснить на таком примере. Пусть внезапно кому-то понадобилось в своём проекте использовать троичную логику для представления значений истинности. Для этого разработчик создаёт простой тип данных:

```
data Ternary
  = TFalse     -- Ложь
  | TUndefined -- Неопределённость
  | TTrue      -- Истина
```

Как сделать так, чтобы разработанные выше функции `peirce` и `schaeffer` работали со значениями этого нового типа данных? Разумеется, что эта задача решается при помощи определения экземпляра класса `Logic`:

```
instance Logic Ternary where
  neg TFalse     = TTrue
  neg TUndefined = TUndefined
  neg TTrue      = TFalse


  TFalse <&&> _     = TFalse
  _ <&&> TFalse     = TFalse
  TUndefined <&&> _ = TUndefined
  _ <&&> TUndefined = TUndefined
  _ <&&> _          = TTrue
```

Это определение достаточно для того чтобы определённые ранее функции `peirce` и `schaeffer` работали со значениями типа `Ternary`. Более того, они будут возвращать на этих значениях абсолютно правильный результат, который определяется результатом работы методов класса `Logic`. Здесь видна удивительная способность этой технологии в языке Haskell — тип `Ternary` и экземпляр класса `Logic` для него могут быть определены в совершенно стороннем модуле, который никак не связан с первоначальным, где определены класс и прикладные функции к нему. Однако такие функции будут вполне успешно работать с новыми типами данных, о которых первоначальный разработчик класса и функций мог даже и не предполагать.

Наконец, ещё один пример может показать, что новые классы могут использоваться и для организации нового понимания для уже имеющихся типов данных. Например, в математике имеются расширения классической двузначной логики в бесконечнозначные. Одним из самых известных таких расширений является нечёткая логика, значения истинности которой лежат в интервале $[0, 1]$. Другим расширением является логика антиномий, значения истинности которой лежат немного в другом интервале: $[-\infty, \infty]$. Для представления таких значений истинности можно воспользоваться типом `Float` (к примеру). Что сделать, чтобы с этим типом данных работали методы класса `Logic` и его прикладные функции? Всё тоже самое — необходимо определить экземпляр:

```
instance Logic Float where
  neg   = (0 -)
  <&&> = min
  <||> = max
```

Теперь действительные числа можно использовать в качестве значений истинности в бесконечнозначной логике антиномий.

3.4. Изоморфные типы

В языке Haskell, который, как известно, является нестрогим, ошибочные вычисления и вычисления, которые не могут быть остановлены (например, бесконечная рекурсия без точки выхода), обозначаются символом ($\perp$). Этот символ обозначает неопределённое значение, при этом система типизации построена так, что любой алгебраический тип данных неявно включает в множество своих значений ($\perp$).

Кроме того, для алгебраических типов данных невозможно определить более одного экземпляра заданного класса (этот вопрос будет детально раскрыт далее), а для синонимов типов экземпляры вообще определять нельзя. Поэтому нет возможности описать иное поведение для значений некоторого типа данных, если такое поведение через некоторый класс уже описано. Это накладывает определённые ограничения, которые не всегда позволяют разработчику программного обеспечения выразить те идеи, которые он желает.

Для решения указанных проблем в языке Haskell имеется возможность для определения изоморфных типов данных, то есть таких, которые тождествен-

ны с некоторым типом относительно множества значений. Изоморфные типы определяются с помощью ключевого слова **newtype**. Например:

```
newtype PointsList = PL [Point3D]
```

В этом определении после ключевого слова **newtype** стоит наименование изоморфного типа, после которого через символ определения (=) записывается конструктор изоморфного типа и тип данных, для которого определяется этот изоморфный тип. В приведённом выше определении записано, что тип `PointsList` является в точности списком на значениях типа `Point3D`. Для создания значений этого типа используется конструктор данных `PL`.

Резонный вопрос, почему не определить то же самое как синоним или даже новый алгебраический тип данных? Например, вот так:

```
type PointsListS = [Point3D]


data PointsListADT = PLadt [Point3D]
```

Про синоним типов `PointsListS` даже не стоит говорить, ибо, как сказано ранее, это просто новое наименование для типа `[Point3D]`, не более. Для синонимов происходит простая синтаксическая замена там, где они встречаются. Для понимания различия конструкторов `PL` и `PLadt` необходимо ввести ещё два определения:

```
f1 (PL _) = []


f2 (PLadt _) = []
```

И вот здесь и кроется кардинальное отличие. Если конструкторам `PL` и `PLadt` передать в качестве входного параметра неопределённое значение ($\perp$), а затем полученные данные передать на вход функциям `f1` и `f2` соответственно, то результаты их выполнения будут различаться. Если вызов `f2 (PLadt ⊥)` вернёт пустой список `[]`, то вызов `f1 (PL ⊥)` вернёт значение ($\perp$). Другими словами, `(N ⊥)` всегда эквивалентен ($\perp$), а `(D ⊥)` неэквивалентен ($\perp$), если только конструктор `D` не строгий (здесь: `N` — конструктор изоморфного типа, `D` — конструктор алгебраического типа данных).

3.4.1. Определение нескольких экземпляров для уникальной пары (класс, тип)

В языке Haskell запрещено создание нескольких экземпляров некоторого класса для одного и того же типа данных. Действительно, в противном случае транслятор языка не смог бы определить, какой именно экземпляр использовать при рассмотрении прикладных функций, когда такие прикладные функции работают с экземплярами некоторых классов. В стандарте языка запрещено даже определять разные экземпляры одного и того же класса для типов, являющихся общим и конкретизированным вариантом одного алгебраического типа данных (например, нельзя создать экземпляры класса `Eq` для типов `Maybe a` и `Maybe Int`, поскольку более общий тип, использующий параметрический полиморфизм, перекрывает собой конкретизированный, и в этом случае транслятор также находится в затруднении). Хотя последнее ограничение снято в некоторых компиляторах (например, GHC), оно обычно входит в состав нестандартизированных расширений языка.

Как же быть, если требуется создать новый экземпляр класса для какого-то типа данных, при этом экземпляр такой уже существует и имеет иную функциональность, нежели планирует создать разработчик программного обеспечения? Такая ситуация встречается достаточно часто — создатель класса может создать для него и экземпляры для встроенных типов, а разработчик, который будет использовать этот класс у себя в проекте, может захотеть переопределить экземпляры. Проблема усугубляется тем, что все экземпляры классов всегда неявно импортируются из импортируемых модулей при импорте соответствующих классов и типов данных, и отключить эту возможность нельзя (поскольку у экземпляров нет идентификаторов).

Для обхода такого ограничения в языке Haskell как раз и используются изоморфные типы данных. Конечно, это будет уже немного не тот тип данных, однако его функциональность будет соответствовать оригинальному типу. Вместе с тем для него можно создать новый экземпляр некоторого класса, который уже существует для исходного типа.

Например, пусть для нужд решения некоторой задачи необходимо переопределить операцию сравнения для целых чисел. Вполне возможно, что в рамках этой задачи «равными» считаются целые числа, которые равны между собой по модулю 10 (имеют одинаковый остаток от целочисленного деления числа

на 10). Просто так определить новый экземпляр класса `Eq` для типа `Int` нельзя, такой экземпляр уже существует в стандартном модуле `Prelude`. Поэтому необходимо определить изоморфный класс:

```
newtype Int10 = Int10 Int
```

Для этого типа уже можно определить экземпляр класса `Eq`. Вспоминая функциональную особенность поставленной задачи, для типа `Int10` экземпляр класса `Eq` определяется следующим образом:

```
instance Eq Int10 where
  (Int10 x) == (Int10 y) = (x `mod` 10) == (y `mod` 10)
```

Этот подход немного неприятен с эстетической точки зрения, но он достаточен для решения поставленной задачи. Все прикладные функции будут автоматически работать с новыми изоморфными типами данных. Единственное, что иногда может потребоваться, — создание особенных функций для работы непосредственно с изоморфными типами, в сигнатурах которых будет явно прописан этот тип. И хотя это является дурным тоном при программировании на языке Haskell, такое может иногда встречаться при программировании крупных проектов. В любом случае при разработке программ необходимо стремиться создавать такие определения функций, которые будут наиболее общими, а для этого необходимо пользоваться системой классов и их экземпляров в языке Haskell.

3.5. Автоматическое построение экземпляров

В языке Haskell имеется возможность автоматического построения экземпляров классов для некоторых классов из стандартного модуля `Prelude`. У разработчика программного обеспечения в таком случае нет необходимости явно определять экземпляры. Для этих целей используется ключевое слово **deriving**, которое записывается после объявления алгебраического типа данных или изоморфного типа. После этого ключевого слова идёт перечисление классов, для которых необходимо автоматически построить экземпляры. Данное перечисление заключается в круглые скобки, если классов несколько. Если класс один, то можно просто привести его наименование. Например:

```
newtype MyBool = MB Bool
  deriving Show


data Number
  = N Integer
  | F Float
  deriving (Eq, Ord, Show, Read)
```

Сразу же необходимо отметить, что для автоматически определяемых экземпляров классов действуют все те же правила и ограничения, что и для экземпляров классов, которые программист определяет самостоятельно. Например, если у некоторого класса имеются базовые классы, то при автоматическом построении экземпляра для этого класса необходимо быть уверенным, что экземпляр для базового класса тоже строится или уже существует.

Классами из стандартного модуля `Prelude`, для которых можно автоматически строить экземпляры, являются: `Eq`, `Ord`, `Enum`, `Bounded`, `Show` и `Read`. Впрочем, для некоторых классов, которые определены в стандартных библиотеках, тоже можно использовать методику автоматического построения экземпляров. Обычно такие классы достаточно просты, чтобы транслятор языка Haskell мог самостоятельно понять, как строить экземпляр для них. Например, для класса `Ix` можно автоматически построить экземпляры.

Тела методов классов, для которых автоматически строятся экземпляры, создаются транслятором языка Haskell при помощи применения несложных синтаксических правил, работающих с определениями класса и типа, которые связываются экземпляром. Пусть имеется некоторый тип `T`, объявленный следующим образом (для определённости и простоты рассмотрения пусть это будет алгебраический тип данных; для изоморфных типов рассуждения и правила такие же):

$$\begin{aligned}\texttt{data}\, cx \Rightarrow \mathrm{T}\, a_1 \ldots a_k = {} & \mathrm{K}_1\, t_{11} \ldots t_{1k_1} \\ & \mid \ldots \\ & \mid \mathrm{K}_\mathrm{n}\, t_{n1} \ldots t_{nk_n} \\ & \texttt{deriving}\, (\mathrm{C}_1, \ldots, \mathrm{C}_\mathrm{m})\end{aligned}$$

Тогда автоматическое объявление экземпляра для класса `C` возможно при выполнении следующих условий:

1) Класс `C` является одним из классов `Eq`, `Ord`, `Enum`, `Bounded`, `Show` или `Read`.

2) Существует контекст cx' такой, что cx' `=>` `C` t_{ij} выполняется для всех компонентов типа t_{ij}.

3) Если класс `C` является классом `Bounded`, то тип `T` должен представлять собой либо перечисление (все его конструкторы не принимают на вход аргументов), либо иметь только один конструктор.

4) Если класс `C` является классом `Enum`, то тип `T` должен безусловно представлять собой перечисление.

5) Не должно существовать никакого явного определения экземпляра класса `C` для типа `T`.

Определения изоморфных типов при помощи ключевого слова **newtype** в этом случае просто трактуется как объявление алгебраического типа данных с одним конструктором.

Каждое автоматически создаваемое определение экземпляра по описываемой технологии в данном случае будет иметь вид:

$$\texttt{instance}\ (cx, cx') \texttt{=>}\ \mathrm{C_i}\ (\mathrm{T}\ a_1 \ldots a_k)\ \texttt{where}\ \{\ d\ \}$$

где d строится автоматически в зависимости от методов класса $\mathrm{C_i}$ и определения `T`. При этом контекст cx' является самым узким контекстом, который удовлетворяет пункту 2 перечисленных выше принципов. Для взаимно рекурсивных типов данных транслятору языка Haskell может потребоваться произвести много дополнительных вычислений для его получения.

То, каким именно способом автоматически строятся экземпляры стандартных классов для алгебраических типов данных и изоморфных типов, можно прочитать в специализированной литературе. Описание этих методов выходит за рамки этого справочника.

3.6. Окончательные замечания о системе типов в языке Haskell

Таким образом, после изучения типов, классов и их экземпляров, должно быть ясно, что эти три программные сущности являются в системе типизации

языка Haskell самостоятельными и слабо зависящими друг от друга. Это значит, что типы данных, классы типов и экземпляры классов могут определяться совершенно независимо друг от друга, при этом экземпляры будут собой связывать типы и классы.

То, что записывается в контексте (`Class a =>`), является всего лишь ограничением вида «должен существовать экземпляр указанного класса `Class` для заданного типа `a`». Это значит, что типы данных и их реализации для определённых классов существуют раздельно и определяются только экземплярами.

В качестве отличного способа объяснения этого подхода можно рассмотреть пример, когда три разработчика программного обеспечения практически независимо друг от друга (и даже не зная о существовании друг друга) могут совместно поработать над одной задачей. Пусть первый программист создал очень интересный класс `VeryInterestingClass`, который успешно использовал в своём проекте. Независимо от него второй программист определил для решения каких-то своих задач некоторый тоже весьма интересный тип данных `InterestingDataType`. Он тоже успешно использовал это определение в своём проекте, после чего, убедившись, что тип данных достаточно полезен, опубликовал его в виде библиотеки.

Третий программист, ознакомившись с результатами работы первых двух разработчиков, которые даже и знать-то друг о друге не знают, не говоря уже о решённых друг другом задачах, внезапно осознаёт, что результаты работы первых двух программистов можно использовать для решения одной очень непростой задачи, над которой долгое время безуспешно бились многие и многие умы. Он быстро создаёт некоторую функцию `megaImportantFunction`, в сигнатуре которой без колебаний записывает (троеточие, само собой, не входит в синтаксис языка Haskell, но использовано здесь для умолчания по поводу реализации умозрительной в данном примере функции):

```
megaImportantFunction :: VeryInterestingClass a => ...
```

Что делать дальше? Ведь тип `InterestingDataType` совершенно не связан с классом `VeryInterestingClass`, но именно такая связь требуется для найденного решения. Конечно, третий программист может самостоятельно пересоздать все прикладные функции класса и этого типа данных, что позволит избежать связывания. Но зачем это делать, если можно просто определить новый экземпляр класса:

```
instance VeryInterestingClass InterestingDataType where
  ...
```

И тут выходит, что нет никакой необходимости искать ни первого программиста, создавшего класс `VeryInterestingClass`, чтобы он включил в его определение возможность работать с типом `InterestingDataType`, ни второго, который мог бы сделать это со своей стороны. Достаточно в своём модуле реализовать экземпляр этого класса для типа `InterestingDataType`, что позволит использовать значения этого типа как в функции `megaImportantFunction`, так и во всех прикладных функциях класса автоматически. И самое главное, исходные модули остаются нетронутыми.

Всё это — серьёзнейшее преимущество системы типизации языка Haskell.

Глава 4.

Модули

Как и любой другой высокоразвитый язык программирования, язык Haskell имеет продуманную систему ведения модулей, при помощи которой можно решать задачи не только группировки программных сущностей в отдельные файлы исходных кодов (модули) для последующего многостороннего использования, но и задачу абстракции типов данных. Эти аспекты системы модулей в языке Haskell будут рассмотрены в этой главе.

4.1. Система модулей

Модули в языке Haskell являются одним из пяти типов программных сущностей, которыми оперирует этот язык для построения программ. Модули — это контейнерные декларации самого верхнего уровня, которые включают в себя определения других программных сущностей: функций, типов данных, классов типов и экземпляров классов. Основной способ определения модуля выглядит следующим образом:

```
module ModuleName where
```

В этом определении используются ключевые слова **module** и **where**, между которыми заключается наименование модуля. Это наименование должно начинаться с заглавной буквы. Название модуля необязательно должно совпадать с именем файла, в котором модуль находится, но при таком совпадении транслятор языка Haskell может самостоятельно найти требуемый модуль.

Каждый файл с исходными кодами на языке Haskell может (и должен) содержать ровно один модуль. Указанная выше строка определения модуля должна быть первой значащей строкой (не комментарием) в любом исходном файле. Однако можно опустить это определение, тогда транслятор языка Haskell будет использовать имя модуля по умолчанию — `Main`.

4.1.1. Экспорт программных сущностей

Модули в языке Haskell могут самостоятельно определять, какие программные сущности, определённые в самом модуле, могут быть видимы извне. Для этого используется секция экспорта в определении модуля. В примере выше эта секция была пропущена, что означает для транслятора языка, что такой модуль экспортирует все программные сущности, которые в нём определены.

С другой стороны, если между наименованием модуля и ключевым словом **where** перечислить наименования программных сущностей, то извне данного модуля будут доступны только перечисленные элементы модуля. Таким образом определяется интерфейс модуля. Перечисление экспортируемых программных сущностей производится через запятую в круглых скобках `()`:

```
module SomeModule (
  SomeType,
  someFunction
) where
```

В этом примере модуль `SomeModule` экспортирует только две программные сущности: тип (или класс, это определить невозможно, хотя по имени должно быть понятно, что тип) `SomeType` и функцию `someFunction`. Этот модуль может содержать внутри себя и другие определения, но они не будут видны извне модуля. Однако перечисленные программные сущности должны быть определены внутри модуля в обязательном порядке. В противном случае возникнет ошибка.

Что интересно, конструкторы данных в типах тоже могут быть экспортированы по желанию разработчика программного обеспечения. В приведённом выше примере у типа `SomeType` не экспортируется ни одного конструктора, поэтому этот тип может быть использован ограниченно — извне модуля можно будет пользоваться только идентификатором `SomeType`. В случае, если необходимо экспортировать определённые конструкторы этого типа, то их также необходимо

перечислить через запятую в круглых скобках (), но уже после наименования типа:

```
module SomeModule (
  SomeType
  (
    One,
    Two
  ),
  someFunction
) where
```

Если необходимо экспортировать все конструкторы некоторого типа, то их можно не перечислять в круглых скобках, заменив двумя точками:

```
module SomeModule (
  SomeType(..),
  someFunction
) where
```

Такой подход к раздельному экспорту (и импорту) конструкторов данных у типов позволяет создавать абстрактные типы данных, у которых скрыта функциональность по созданию объектов этого типа. Эта технология подробно описана в разделе 4.2.. Абсолютно такой же подход используется и для методов экспортируемых классов — можно выборочно экспортировать только те методы классов, которые необходимы извне модуля.

4.1.2. Импорт сторонних модулей

Вполне естественно, что оформленные таким образом модули в языке Haskell можно импортировать в другие модули для использования тех программных сущностей, которые экспортируются вовне импортируемыми модулями. Это — главное предназначение системы модулей в любом языке программирования. На это основано и построение библиотек, и повторное использование программных сущностей.

Импортирование сторонних модулей производится при помощи ключевого слова **import**. После этого ключевого слова должно располагаться наименование импортируемого модуля, после которого может находиться перечисление тех программных сущностей, которые импортируются в текущий модуль. В случае

если такого списка нет, в текущий модуль импортируются все программные сущности, которые экспортируются импортируемым модулем. Например:

```
import Data.List


import Data.Char (
  toLower,
  toUpper
)


import Data.Tree (
  Tree
  (
    Node
  )
)
```

Как видно, при перечислении импортируемых программных сущностей действуют те же самые правила, которые используются и при перечислении экспортируемых элементов. Для импортируемых типов данных можно также перечислять импортируемые конструкторы данных. При этом необходимо учесть, что множество импортируемых программных сущностей в любом случае должно быть подмножеством экспортируемых. В случае если для импорта указана некоторая программная сущность, не перечисленная в списке экспортируемых в импортируемом модуле, возникнет ошибка.

Остаётся рассмотреть случай, когда в импортируемых модулях имеются программные сущности с одинаковыми идентификаторами. Как поступать в такой ситуации? Для этих целей в языке Haskell можно использовать либо сокрытие программных сущностей при импорте, либо квалификацию оных посредством имени модуля, в котором они описаны. Ниже рассматриваются эти методы.

Сокрытие программных сущностей при импорте модуля

Иногда бывает проще при импорте скрыть некоторые программные сущности из импортируемого модуля, нежели перечислять огромный список тех, что импортируются. Это можно сделать при помощи ключевого слова **hiding**, которое должно располагаться после наименования модуля при импорте. После этого

слова опять же в круглых скобках `()` через запятую перечисляются идентификаторы тех программных сущностей, которые скрываются при импорте. Например:

```
import SomeModule hiding (
  someFunction
)
```

Само собой разумеется, что в этом случае можно скрывать только те программные сущности, которые экспортируются импортируемым модулем. Во всяком случае при импорте нескольких модулей всегда необходимо добиваться ситуации, когда при импорте не происходит коллизии имён программных сущностей — все коллизии должны быть разрешены при помощи сокрытия или квалификации (см. ниже).

Необходимо отметить, что подобным образом невозможно скрыть типы данных и синонимы типов. Если в нескольких импортируемых модулях имеются типы с одинаковыми идентификаторами, то необходимо использовать квалификацию имён.

Квалифицированный импорт модуля

Иногда в описанном выше случае, когда при импорте двух модулей получается коллизия имён программных сущностей, в проекте для работы необходимы обе сущности. Что делать? Вопрос несложный — язык Haskell предоставляет возможность использовать квалификацию идентификаторов при импорте. Это значит, что при использовании внешних (импортированных) программных сущностей их наименования при вызове должны префиксироваться идентификатором модуля.

Для этих целей необходимо не просто импортировать заданный модуль, но и указать, что модуль импортируется квалифицированно. Тогда все идентификаторы, получаемые из этого модуля, должны при использовании квалифицироваться наименованием модуля. Это достигается при помощи использования ключевого слова **qualified**:

```
import qualified SomeModule
```

После такой директивы все программные сущности из модуля `SomeModule` должны использоваться только с одноимённым префиксом, отделённым от идентификатора программной сущности точкой:

```
otherFunction data = map SomeModule.somefunction data
```

Такой подход позволяет избежать неопределённости в наименованиях. Однако после квалифицированного импорта неквалифицированные идентификаторы импортированных программных сущностей уже использовать нельзя, даже если с такими идентификаторами нет никаких коллизий.

Единственная проблема, которая возникает с квалификацией идентификаторов из импортируемых модулей, — использование точки в качестве операции композиции (`.`), которая определена в стандартном модуле `Prelude`. Если квалифицированный идентификатор участвует в композиции функций, то транслятор языка Haskell может не понять, какая точка за что отвечает. В этом случае необходимо просто отделить операцию композиции (`.`) от своих аргументов пробелами.

Переименование модуля при квалифицированном импорте

Остаётся отметить такой аспект импортирования модулей, как переименование квалифицированного модуля. Иной раз разработчик программного обеспечения назовёт свой модуль слишком длинным наименованием, поэтому его использование в качестве префикса становится делом не только затруднительным, но и весьма сильно портящим исходные коды программ. Поэтому при импорте можно переименовать модуль, дав ему короткое название. Для этого используется ключевое слово **as**:

```
import qualified AModuleWithAVeryLongName as M
```

В этом случае квалификация идентификаторов из модуля `AModuleWithAVeryLongName` должна производиться при помощи префикса `M`. Более того, при переименовании модулей не накладывается ограничения, которое запрещает разным модулям давать одинаковые имена. Если при импорте таких модулей не создаётся коллизий идентификаторов, то для разных модулей можно использовать одинаковые идентификаторы, введённые ключевым словом **as**. Однако это не очень хорошая практика, поскольку приводит к путанице.

4.2. Абстракция данных при помощи модулей

Абстрактным типом данных называют такой тип, который предоставляет для работы с элементами этого типа определённый набор функций, а также возможность создавать элементы этого типа при помощи специальных функций. Вся внутренняя структура такого типа спрятана от разработчика программного обеспечения — в этом и заключается суть абстракции. Абстрактный тип данных определяет набор независимых от конкретной реализации типа функций для оперирования его значениями.

В теории и практике программирования абстрактные типы данных обычно представляются в виде интерфейсов, которые скрывают соответствующие реализации типов. Программисты работают с абстрактными типами данных исключительно через их интерфейсы, поскольку реализация может в будущем измениться. Такой подход соответствует принципу инкапсуляции в объектно-ориентированном программировании. Сильной стороной этой методики является именно сокрытие реализации. Раз вовне опубликован только интерфейс доступа к значениям типа, то пока реализация этого типа поддерживает этот интерфейс, независимо от того, как меняется сама внутри, все программы, работающие с заданным абстрактным типом данных, будут продолжать работать.

Между абстрактными типами данных и структурами данных, которые реализуют абстрактные типы, существует очень тонкое различие, которое иногда некоторые программисты уловить не могут. Например, такой абстрактный тип данных, как список $List(A)$, может быть реализован при помощи массива, линейного списка той или иной направленности и т. д. Однако над списком $List(A)$ определён набор функций, который будет работать всегда, независимо от того, как сам список реализован.

Педантичный читатель спросит: «Причём же здесь модули?» Действительно, в какой-то мере абстракция данных уже рассмотрена в должной мере в предыдущей главе справочника. Однако остаётся рассмотреть, каким же образом можно скрывать реализацию данных.

Рассмотренные в предыдущем разделе механизмы экспорта и импорта программных сущностей в языке Haskell наводят на мысль о том, что сокрытие реализаций типов можно производить при помощи ограничения экспорта из создаваемого модуля. Действительно, раз ближайшей к интерфейсу программной сущностью по своей сути является класс в языке Haskell, то и оперировать при экс-

порте из модулей необходимо только классами и предлагаемыми ими методами. Само собой разумеется, что сами типы данных, реализующие функциональность класса, экспортироваться не должны.

Это значит, что во всех прикладных функциях невозможно будет использовать ни конструкторов типов, ни их конструкторов данных. В свою очередь это означает, что в сигнатурах функций нельзя будет указывать конкретный тип, но можно будет только указывать ограничение (`Class a =>`). Более того, невозможно будет пользоваться механизмом сопоставления с образцами, поскольку образцы сформировать будет нельзя (нет операций для конструирования данных). В этом и заключается высшая степень абстракции данных в языке Haskell.

Этот подход можно сравнить с объектно-ориентированной парадигмой, в которой абстракция данных является одним из главенствующих подходов. Расширяя понятие объектно-ориентированного класса на язык Haskell, можно сказать, что любая функция, у которой в сигнатуре указан контекст, ограничивающий применение функции экземплярами какого-либо класса (в смысле языка Haskell), является методом этого класса, хотя она и не определена внутри класса. При использовании этих функций нет никакой разницы между такими функциями и методами, указанными в определении класса. При этом контекст гарантирует, что для того чтобы создать некий экземпляр класса («реализацию интерфейса» в понимании объектно-ориентированной парадигмы), нужно определить не все методы, а совсем немного (подмножество тех, что перечислены в определении класса). В этом состоит естественный способ расширения интерфейсов, который никак не влияет на уже написанный код, позволяя добавлять к интерфейсу новые методы.

4.3. Кое-что ещё о модулях

Наконец, остаётся отметить, что имеются и иные аспекты использования модулей в языке Haskell. Правда, все эти тонкости можно пересчитать по пальцам, поэтому они просто перечислены ниже.

1) В связи с тем, что такая программная сущность, как экземпляр класса в языке Haskell, не имеет наименования, она экспортируется модулем и импортируется внешним модулем всегда в случае экспорта или импорта и класса, и типа, для которого определён экземпляр.

2) Трансляторы языка Haskell ищут файлы с модулями для включения в текущем каталоге проекта либо в специальных каталогах, в которых собраны модули стандартных библиотек. При коллизии имён модулей приоритет использования имеет тот, который находится в каталоге проекта.

3) В стандартных библиотеках языка Haskell принята система именования модулей, которая включает в наименование относительный путь к файлу модуля от точки входа в корневой каталог модулей. При этом все части наименования должны начинаться с заглавной буквы и разделяться точкой (`.`). Например:

```
import Data.List
```

Такая директива импорта означает, что модуль `List` находится в файле `/data/list.hs` в каталоге модулей.

Глава 5.

Сводная информация

Таким образом, перечисленные и описанные в предыдущих главах программные сущности являются тем, что предоставляет язык Haskell для работы на поприще создания программного обеспечения. Действительно, в языке Haskell всего пять видов программных сущностей:

1) функции;
2) типы данных;
3) классы типов;
4) экземпляры классов;
5) модули.

Все объекты предметной области и отношения между ними выражаются при помощи перечисленных сущностей. Поэтому в качестве справочной информации о языке Haskell остаётся привести список зарезервированных ключевых слов и операций, которые сами по себе нее могут быть использованы в качестве идентификаторов программных сущностей, создаваемых разработчиком программы. Все такие зарезервированные идентификаторы сведены в следующие таблицы: первая перечисляет ключевые слова, вторая — зарезервированные символы операций соответственно.

Слово	**Значение**	**Использование**
as	Отделение синонима импортируемого модуля при использовании квалифицированных имён программных сущностей	Модули
case	Организация множественного ветвления вычислительного процесса в зависимости от значений какого-либо образца	Функции
class	Определение класса типов	Классы
data	Определение алгебраического типа данных	Типы
default	Определение типа переменных по умолчанию для случаев, когда тип переменной невозможно однозначно вывести	Типы
deriving	Указание для транслятора языка вывести экземпляры заданных классов для указанного типа самостоятельно	Типы
do	Выполнение последовательности монадических действий, связанных операциями (>>) или (>>=)	Функции
else	Определение альтернативного потока вычислений в конструкции **if-then-else**, то есть того потока, который вычисляется в случае ложности логического выражения **if**	Функции
if	Организация двоичного ветвления в зависимости от значения булевского выражения	Функции
import	Импорт определений из внешнего модуля	Модули
in	Отделение выражения в конструкции **let-in** от локальных определений	Функции

Слово	**Значение**	**Использование**
infix	Определение приоритета бинарной операции, для которой неприменимы правила ассоциативности	Функции
infixl	Определение приоритета левоассоциативных бинарных операций	Функции
infixr	Определение приоритета правоассоциативных бинарных операций	Функции
instance	Определение экземпляра класса	Экземпляры
let	Префиксное определение локальных функций, которое к тому же является выражением	Функции
module	Определение модуля и экспортируемых им программных сущностей	Модули
newtype	Определение изоморфного типа данных	Типы
of	Отделение альтернатив во множественном ветлении **case**	Функции
qualified	Описание того факта, что программные сущности из заданного внешнего модуля импортируются квалифицированными	Модули
then	Определение потока вычислений в условном выражении **if-then-else** для случая, если охраняющее условное выражение принимает истинное значение	Функции
type	Определение синонима типа	Типы

Слово	**Значение**	**Использование**
where	Используется для различных целей. *Классы* и *экземпляры*: отделение перечислений сигнатур и определений методов от заголовка определения. *Модули*: отделение определений программных сущностей модуля от заголовка определения. *Функции*: постфиксное определение локальных функций, которое не является выражением	К, М, Ф, Э
(_)	Маска подстановки в образцах, которая символизирует произвольное значение образца, не используемое в дальнейших вычислениях	Функции

В столбце «Использование» данной таблицы указываются те программные сущности, при работе с которыми используется то или иное ключевое слово. Эту информацию можно использовать для поиска описаний ключевых слов в предыдущих главах книги.

В следующей таблице перечислены зарезервированные символы и последовательности символов, которые используются для различных целей в языке Haskell. Все они используются при определениях функций, поэтому детально рассматриваются в главе 1..

Символы	**Значение**
(..)	Пропущенные значения в генераторах списков. Используется как синтаксическая уловка для методов класса `Enum`
(:)	Конструктор списка, создающий пару
(::)	Определитель типа программной сущности. Обычно используется в сигнатурах функций для ограничения типов функций

Символы	Значение
(=)	Символ для записи определения функций, типов, и синонимов типов. Слева от него — идентификатор определяемой сущности, слева — описание
(\)	Определение безымянной функции в виде λ-терма
(|)	Символ для разделения выражений охраны в определениях функций и внутри альтернатив оператора **case**
(<-)	Сопоставление с образцом в монадическом действии, указанном в списке ключевого слова **do**
(->)	Символ, который используется в нескольких значениях: 1) В сигнатурах типов разделяет типы операндов и результата функции. 2) В безымянных λ-термах заменяет точку (.) из математической нотации. 3) В операторе множественного ветвления **case** используется для указания ветви вычислений в зависимости от значения условного выражения
(@)	Связывает образец как единое целое и его внутреннюю структуру (именованный образец)
(~)	Определяет ленивый образец
(=>)	Разграничивает контекст и тип в сигнатурах типов программных сущностей

Кроме того, в идентификаторах операций нельзя использовать скобки: обычные (круглые) — «(» и «)», квадратные — «[» и «]», а также фигурные — «{» и «}». Все эти скобки непосредственно используются в синтаксисе языка Haskell.

Ну и наконец, необходимо отметить, что в языке Haskell имеется одно ограничение на имена идентификаторов, вернее даже, — соглашение об именовании программных сущностей, которое входит в синтаксис языка. Это соглашение гласит, что имена функций должны всегда начинаться со строчной буквы, а имена типов, классов и модулей — с заглавной (экземпляры классов имён не имеют). Нарушение данного соглашения будет вести к синтаксическим ошибкам.

Часть II.

Стандартные библиотеки

Вторая часть справочника содержит описания программных сущностей в стандартных модулях и библиотеках языка Haskell, которые всегда поставляются вместе с любыми трансляторами языка (многие из этих модулей даже входят в состав стандарта языка Haskell-98). Описание модулей выполняется в унифицированном виде: сначала описываются все алгебраические типы данных (если имеются), затем классы и их экземпляры и, наконец, функции. Каждая глава второй части посвящена отдельному модулю, соответственно каждая глава разбита на разделы «Алгебраические типы данных», «Классы и экземпляры» и «Функции».

Также в начале каждой главы приводится краткое описание предназначения модуля.

Необходимо отметить, что описание модулей и библиотек приводится в соответствии с версиями таких модулей и библиотек, прилагаемых к интерпретатору HUGS 98 версии от сентября 2006 года и компилятору GHC версии 6.6.1. На деле состав программных сущностей, их способ определения и наличие самих модулей может различаться как от версии интерпретатора HUGS 98, так и от вида транслятора языка Haskell.

Начиная с главы 7. в этой части описываются пакеты модулей в составе иерархической системы модулей. Описание проводится только в рамках стандартных (базовых) модулей, поскольку каждый транслятор языка Haskell может иметь в составе своей иерархии модулей как другие пакеты, так и иные модули в составе стандартных пакетов. Нестандартные пакеты и модули не описываются в этом справочнике.

При описании алгебраических типов данных применяется идиома абстракции типов. Это означает, что для определённых в модулях экземпляров классов не описывается способ реализации, а всего лишь просто упоминается, для каких классов определены экземпляры у описываемого типа. В свою очередь для каждого класса также указано, какие типы обладают экземплярами данного класса.

Необходимо отметить, что в этом справочнике не описываются устаревшие модули, которые до сих пор присутствуют в стандартной поставке языка Haskell, однако постепенно будут выводиться из неё, а потому категорически не рекомендуются к использованию. Описание таких модулей бессмысленно, но стоит перечислить их, чтобы было ясно, чем пользоваться нельзя. Это модули: `FunctorM` (вместо него предлагаются модули `Foldable` — см. раздел 8.11. и `Traversable` — см. раздел 8.28.), `PackedString` (вместо него предлагается модуль `ByteString`

с намного большей функциональностью — см. раздел 8.4.), `Queue` (вместо него необходимо использовать модуль `Sequence`, где определено на порядок больше функций для работы с очередями вида FIFO — см. раздел 8.25.).

Ввиду некоторых ограничений системы вёрстки в нижеследующих описаниях функций и прочих программных сущностей в заголовках не используется символ подчёркивания (_), вместо него используется дефис (-). В последующих определениях используется обычная нотация для имён функций. Это необходимо иметь в виду при использовании справочника.

Остаётся упомянуть, что у каждого стандартного модуля в поставке языка Haskell имеется ответственный, с которым можно связаться по определённому электронному адресу для решения вопросов, связанных с поддержкой и дополнением модулей. Это сделано в целях дополнительной стандартизации, так как получается, что за стандартные библиотеки, входящие в поставку любого компилятора, отвечает один человек. Если не указано иного, то с ответственным за стандартные модули можно связаться по адресу `libraries@haskell.org`.

Глава 6.

Стандартный модуль `Prelude`

Стандартный модуль `Prelude` является начальным файлом, который всегда загружается интерпретатором в начале своей работы или импортируется транслятором языка Haskell в любой другой модуль. Определённые в нём алгебраические типы данных, классы и функции доступны всегда и везде (если не отключены при помощи явного импорта), а потому модуль содержит самые необходимые и часто используемые определения, связанные с обработкой примитивных типов, кортежей и списков.

6.1. `Prelude`: Алгебраические типы данных

Тип: `()`
Описание: тип `()` является примитивным типом для описания пустых значений. Любое пустое значение должно иметь этот тип (аналог типа `void` в языке C).
Определение:

```
data () = ()
  deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)
```

Является экземпляром классов `Eq`, `Ord`, `Ix`, `Enum`, `Read`, `Show` и `Bounded`. Определён внутри транслятора.
Тип: `Bool`
Описание: тип `Bool` предназначен для представления булевских значений истинности, то есть значений «ИСТИНА» и «ЛОЖЬ».

Определение:

```
data Bool
  = False
  | True
  deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)
```

Является экземпляром классов Eq, Ord, Ix, Enum, Read, Show и Bounded.

Тип: Char

Описание: тип Char является примитивным, определён внутри транслятора и предназначен для представления однобайтовых символов.

Определение:

```
data Char
```

Является экземпляром классов Eq, Ord, Ix, Enum, Read, Show и Bounded.

Тип: Maybe

Описание: тип Maybe используется для представления значений, которые могут быть «пустыми». Например, некоторые функции в результате своей работы могут не возвратить никакого результата (из-за ошибки или ещё по какой-то причине), но при этом важно показать, что функция именно не вернула никакого результата. Для этих целей используется часть Nothing, которая не хранит значений. Если же значение необходимо вернуть, то используется часть Just, в которой хранится значение произвольного типа a.

Определение:

```
data Maybe a
  = Nothing
  | Just a
  deriving (Eq, Ord, Read, Show)
```

Является экземпляром классов Eq, Ord, Read, Show, Functor и Monad.

Тип: Either

Описание: тип Either используется для представления значений, у которых может быть один из двух заявленных типов — a или b. Обычно этот тип используется в процессе отлова ошибок методом исключений — в одной из частей (например, Left) хранится строковое описание ошибки в случае ошибочных вычислений, в то время как в другой части хранится обычное значение.

Определение:

```
data Either a b
  = Left a
  | Right b
  deriving (Eq, Ord, Read, Show)
```

Является экземпляром классов Eq, Ord, Read и Show.

Тип: Ordering

Описание: тип Ordering является вспомогательным типом для представления отношения порядка в операциях вида (<), (>=) и т. д.

Определение:

```
data Ordering
  = LT
  | EQ
  | GT
  deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)
```

Является экземпляром классов Eq, Ord, Ix, Enum, Read, Show и Bounded.

Тип: []

Описание: тип [] представляет собой список произвольных элементов одного типа. Является одним из основных типов, используемым в языке Haskell для работы.

Определение:

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

Является экземпляром классов Eq, Ord, Read, Show, Functor и Monad. Определён внутри транслятора.

Тип: (,)

Описание: набор типов для представления кортежей произвольных размеров. Данные типы определены внутри транслятора и являются однообразными.

Определение:

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

Стандарт предполагает, что любой транслятор должен определять подобные типы для представления кортежей длиной вплоть до четырнадцати. Бóльшие длины кортежей могут также предоставляться трансляторами, но стандарт языка не регламентирует их наличие.

Данные типы являются экземплярами классов Eq, Ord, Ix, Read и Show.

Тип: Int

Описание: тип Int является примитивным и используется для представления ограниченных целочисленных значений. Величина значений лежит в интервале [-2147483648, 2147483647] ($[-2^{31}, 2^{31} - 1]$).

Определение:

```
data Int
```

Является экземпляром классов Eq, Ord, Num, Integral, Ix, Enum, Read, Show и Bounded.

Тип: Integer

Описание: тип Integer является примитивным и используется для представления неограниченных целочисленных значений.

Определение:

```
data Integer
```

Является экземпляром классов Eq, Ord, Num, Integral, Ix, Enum, Read, Show и Bounded.

Тип: Float

Описание: тип Float является примитивным и используется для представления действительных значений одинарной точности.

Определение:

```
data Float
```

Является экземпляром классов Eq, Ord, Num, Real, Fractional, Floating, RealFloat, Enum, Read и Show.

Тип: `Double`

Описание: тип `Double` является примитивным и используется для представления действительных значений двойной точности.

Определение:

```
data Double
```

Является экземпляром классов `Eq`, `Ord`, `Num`, `Real`, `Fractional`, `Floating`, `RealFloat`, `Enum`, `Read` и `Show`.

Тип: `Ratio`

Описание: тип `Ratio` предназначен для представления дробей с числителем и знаменателем.

Определение:

```
data Integral a => Ratio a
  = !a :% !a
  deriving (Eq)
```

Является экземпляром классов `Eq`, `Ord`, `Num`, `Real`, `Fractional`, `RealFrac`, `Enum`, `Read` и `Show`.

В целях удобства представления для этого алгебраического типа данных определён синоним `Rational`:

```
type Rational = Ratio Integer
```

Этот синоним используется для представления обычных целочисленных дробей.

Тип: `Exception`

Описание: тип `Exception` является сложным перечислением для представления различных вариантов исключений (ошибочных ситуаций), которые могут возникать при работе функций из стандартного модуля `Prelude`.

Определение:

```
 data Exception
  = ArithException ArithException | ArrayException ArrayException | AssertionFailed String
  | AsyncException AsyncException | BlockedOnDeadMVar | Deadlock  | DynException Dynamic
  | ErrorCall String              | ExitException ExitCode        | IOException IOException
  | NoMethodError String          | NonTermination                | PatternMatchFail String
  | RecConError String            | RecSelError String            | RecUpdError String
```

Является экземпляром класса Show.

Тип: ArithException

Описание: тип ArithException используется для представления исключений, связанных с выполнением арифметических операций.

Определение:

```
data ArithException
  = Overflow
  | Underflow
  | LossOfPrecision
  | DivideByZero
  | Denormal
  deriving (Eq, Ord)
```

Является экземпляром классов Eq, Ord и Show.

Тип: ArrayException

Описание: тип ArrayException используется для представления исключений, связанных с выполнением функций над массивами.

Определение:

```
data ArrayException
  = IndexOutOfBounds String
  | UndefinedElement String
  deriving (Eq, Ord)
```

Является экземпляром классов Eq, Ord и Show.

Тип: AsyncException

Описание: тип AsyncException используется для представления исключений, связанных с синхронизацией вычислений.

Определение:

```
data AsyncException
  = StackOverflow
  | HeapOverflow
  | ThreadKilled
  deriving (Eq, Ord)
```

Является экземпляром классов Eq, Ord и Show.

Тип: `ExitCode`

Описание: тип `ExitCode` предназначен для представления кодов завершения операций.

Определение:

```
data ExitCode
  = ExitSuccess
  | ExitFailure Int
  deriving (Eq, Ord, Read, Show)
```

Является экземпляром классов `Eq`, `Ord`, `Read` и `Show`.

Тип: `IOException`

Описание: тип `IOException` используется для представления исключений, связанных с операциями ввода/вывода.

Определение:

```
data IOException
  = IOError
    {
      ioe_handle      :: Maybe Handle,
      ioe_type        :: IOErrorType,
      ioe_location    :: String,
      ioe_description :: String,
      ioe_filename    :: Maybe FilePath
    }
  deriving (Eq)
```

Является экземпляром классов `Eq` и `Show`.

В целях совместимости со старыми версиями определён синоним:

```
type IOError = IOException
```

Тип: `IOErrorType`

Описание: тип `IOErrorType` является перечислением, характеризующим тип ошибки ввода/вывода.

Определение:

```
data IOErrorType = AlreadyExists | NoSuchThing  | ResourceBusy     | ResourceExhausted
                 | EOF           | IllegalOperation     | PermissionDenied | UserError
                 | ProtocolError | UnsupportedOperation | OtherError | DotNetException
  deriving (Eq)
```

Является экземпляром классов Eq и Show.

Тип: IO

Описание: тип IO является примитивным монадическим типом для представления результатов операций ввода/вывода.

Определение:

```
data IO a
```

Является экземпляром классов Functor, Monad и Show.

Тип: IOMode

Описание: тип IOMode является перечислением и предназначен для представления способа открытия ресурсов (файлов, потоков).

Определение:

```
data IOMode
  = ReadMode
  | WriteMode
  | AppendMode
  | ReadWriteMode
  deriving (Eq, Ord, Ix, Bounded, Enum, Read, Show)
```

Является экземпляром классов Eq, Ord, Ix, Enum, Read, Show и Bounded.

6.2. Prelude: Классы и их экземпляры

Класс: Bounded

Описание: экземпляры этого класса представляют собой множества, ограниченные сверху и снизу. Это означает, что имеются минимальный элемент множества и максимальный элемент множества. Однако само множество вполне может быть и неупорядоченным — в нём может отсутствовать отношение порядка.

Определение:

```
class Bounded a where
  minBound, maxBound :: a
```

Экземпляры: (), Bool, Char, Ordering, Int.

Класс: Enum

Описание: значения из множеств-экземпляров класса Enum могут быть пронумерованы. Это означает, что любому элементу такого типа можно поставить в со-

ответствие некоторое целое число, уникальное для конкретного значения типа. Таким образом может быть определен порядок следования элементов такого типа.

Определение:

```
class Enum a where
  succ, pred     :: a -> a
  toEnum         :: Int -> a
  fromEnum       :: a -> Int
  enumFrom       :: a -> [a]
  enumFromThen   :: a -> a -> [a]
  enumFromTo     :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]


  succ                 = toEnum . (1 +)      . fromEnum
  pred                 = toEnum . subtract 1 . fromEnum
  enumFrom x           = map toEnum [fromEnum x ..]
  enumFromThen x y     = map toEnum [fromEnum x,
                                     fromEnum y ..]
  enumFromTo x y       = map toEnum [fromEnum x .. fromEnum y]
  enumFromThenTo x y z = map toEnum [fromEnum x,
                                     fromEnum y .. fromEnum z]
```

Экземпляры: `()`, `Bool`, `Char`, `Ordering`, `Int`, `Integer`, `Float`, `Double`, `Enum (Ratio a)`.

Для работы с типами данных, являющимися экземплярами класса `Enum`, в языке Haskell имеется возможность получения различных перечислений при помощи синтаксиса с двумя точками (`..`). Любой тип данных класса `Enum` поддерживает такой синтаксис. Например, если имеется такое определение типа `Digits`:

```
data Digits = Zero  | One
            | Two   | Three
            | Four  | Five
            | Six   | Seven
            | Eight | Nine
  deriving Enum
```

то вполне можно использовать значения этого перечисления в генераторах:

```
evens :: [Digits]
evens = [Zero, Two..Eight]
```

```
odds :: [Digits]
odds = [One, Three..Nine]
```

Класс: Eq

Описание: определяет класс типов, над которыми определены отношения равенства. Это означает, что элементы таких типов можно сравнивать друг с другом, получая значения истинности булевского типа («ИСТИНА» или «ЛОЖЬ»).

Определение:

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

  x == y = not (x /= y)
  x /= y = not (x == y)
```

Экземпляры: (), Bool, Char, Maybe, Either, Ordering, [a], (a, b), Int, Integer, Float, Double, Ratio a.

Класс: Floating

Описание: этот класс определяет поведение всех числовых типов, чьи элементы являются числами с плавающей точкой.

Определение:

```
class (Fractional a) => Floating a where
  pi                  :: a
  exp, log, sqrt      :: a -> a
  (**), logBase       :: a -> a -> a
  sin, cos, tan       :: a -> a
  asin, acos, atan    :: a -> a
  sinh, cosh, tanh    :: a -> a
  asinh, acosh, atanh :: a -> a
```

```
  pi          = 4 * atan 1
  x ** y      = exp (log x * y)
  logBase x y = log y / log x
  sqrt x      = x ** 0.5
  tan x       = sin x / cos x
  sinh x      = (exp x - exp (-x)) / 2
  cosh x      = (exp x + exp (-x)) / 2
  tanh x      = sinh x / cosh x
  asinh x     = log (x + sqrt (x * x + 1))
  acosh x     = log (x + sqrt (x * x - 1))
  atanh x     = (log (1 + x) - log (1 - x)) / 2
```

Экземпляры: `Float`, `Double`.

Класс: `Fractional`

Описание: этот класс является шаблоном для любого типа, элементами которого являются дробные (рациональные) числа. Все такие типы должны иметь определённую операцию деления. Кроме того, каждый элемент должен иметь обратное значение относительно операции деления. Также все значения этого класса должны иметь возможность преобразования из рациональных чисел.

Определение:

```
class (Num a) => Fractional a where
  (/)          :: a -> a -> a
  recip        :: a -> a
  fromRational :: Rational -> a
  fromDouble   :: Double -> a


  recip x    = 1 / x
  fromDouble = fromRational . toRational
  x / y      = x * recip y
```

Экземпляры: `Float`, `Double`, `Ratio a`.

Класс: `Functor`

Описание: монадический класс для описания возможности производить проекции структур данных друг на друга. Любой тип, являющийся экземпляром этого класса, должен иметь единственную функцию, которая позволяет преобразовать данные этого типа в соответствии с определением некоторой заданной функции. Таким образом, единственный метод этого класса определяет функцию высшего порядка.

Определение:

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

Экземпляры: Maybe, [a], IO.

Класс: Integral

Описание: этот класс определяет шаблон для любого типа, который содержит в себе любые целые числовые элементы.

Определение:

```
class (Real a, Enum a) => Integral a where
  quot, rem, div, mod :: a -> a -> a
  quotRem, divMod     :: a -> a -> (a, a)
  even, odd           :: a -> Bool
  toInteger           :: a -> Integer
  toInt               :: a -> Int

  n `quot` d = q where (q, r) = quotRem n d
  n `rem` d  = r where (q, r) = quotRem n d
  n `div` d  = q where (q, r) = divMod n d
  n `mod` d  = r where (q, r) = divMod n d

  divMod n d = if (signum r == - signum d) then (q - 1, r + d)
                                           else qr
    where qr@(q, r) = quotRem n d

  even n = n `rem` 2 == 0
  odd    = not . even
  toInt  = toInt . toInteger
```

Экземпляры: Int, Integer, Ratio a.

Класс: Ix

Описание: этот класс является описанием шаблона для таких типов, значения которых могут выступать в качестве индексов в массивах данных.

Определение:

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


  rangeSize r@(l, u) | l > u     = 0
                     | otherwise = index r u + 1
```

Экземпляры: `()`, `Bool`, `Char`, `Ordering`, `(a, b)`, `Int`, `Integer`.

Класс: `Monad`

Описание: монадический класс, определяющий методы для связывания переменных и организации передачи их значений из одного вычислительного процесса в другой в чёткой последовательности. Любой тип данных, являющийся экземпляром данного класса, определяет некоторый императивный подмир в рамках функционального программирования для выполнения последовательных действий определённого характера, который зависит от специфики типа.
Определение:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a

  p >> q = p >>= \_ -> q
  fail s = error s
```

Экземпляры: `Maybe`, `[a]`, `IO`.

Класс: `Num`

Описание: это — класс для всех числовых типов. Любой экземпляр этого класса должен поддерживать элементарные арифметические операции (такие как сложение, вычитание и умножение).

Определение:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate        :: a -> a
  abs, signum   :: a -> a
  fromInteger   :: Integer -> a
  fromInt       :: Int -> a


  x - y    = x + negate y
  fromInt  = fromIntegral
  negate x = 0 - x
```

Экземпляры: `()`, `Bool`, `Char`, `Ordering`, `Int`, `Integer`, `Float`, `Double`.

Класс: `Ord`

Описание: шаблон для типов, над экземплярами которых определён порядок следования.

Определение:

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


  compare x y | x == y    = EQ
              | x <= y    = LT
              | otherwise = GT


  x <= y = compare x y /= GT
  x <  y = compare x y == LT
  x >= y = compare x y /= LT
  x >  y = compare x y == GT


  max x y | x >= y    = x
          | otherwise = y


  min x y | x <= y    = x
          | otherwise = y
```

Экземпляры: `()`, `Bool`, `Char`, `Maybe`, `Either`, `Ordering`, `[a]`, `(a, b)`, `Int`, `Integer`, `Float`, `Double`, `Ratio a`.

Класс: `Read`

Описание: шаблон для типов, элементы которых имеют строковое представление.
Определение:

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


class Read a where
  readsPrec :: Int -> ReadS a
  readList  :: ReadS [a]


  readList = readParen False (\r -> [pr | ("[", s) <- lex r,
                                          pr <- readl s ])
    where readl  s = [([], t)   | ("]", t) <- lex s] ++
                     [(x:xs, u) | (x, t)   <- reads s,
                                  (xs, u)  <- readl' t]
          readl' s = [([], t)   | ("]", t) <- lex s] ++
                     [(x:xs, v) | (",", t) <- lex s,
                                  (x, u)   <- reads t,
                                  (xs, v)  <- readl' u]
```

Экземпляры: `()`, `Bool`, `Char`, `Maybe`, `Either`, `Ordering`, `[a]`, `(a, b)`, `Int`, `Integer`, `Float`, `Double`, `Ratio a`.

Класс: `Real`

Описание: этот класс покрывает все числовые типы, элементы которых могут быть представлены как отношения (типичный пример — рациональные числа).
Определение:

```
class (Num a, Ord a) => Real a where
  toRational :: a -> Rational
```

Экземпляры: `Int`, `Integer`, `Float`, `Double`.

Класс: `RealFloat`

Описание: класс, объединяющий в себе свойства классов `RealFrac` и `Floating`, но при этом дополнительно описывающий некоторые функции для работы с числами, представленными в виде значений с плавающей точкой.

Определение:

```
class (RealFrac a, Floating a) => RealFloat a where
  floatRadix     :: a -> Integer
  floatDigits    :: a -> Int
  floatRange     :: a -> (Int, Int)
  decodeFloat    :: a -> (Integer, Int)
  encodeFloat    :: Integer -> Int -> a
  exponent       :: a -> Int
  significand    :: a -> a
  scaleFloat     :: Int -> a -> a
  isNaN          :: a -> Bool
  isInfinite     :: a -> Bool
  isDenormalized :: a -> Bool
  isNegativeZero :: a -> Bool
  isIEEE         :: a -> Bool
  atan2          :: a -> a -> a

  exponent x = if (m == 0) then 0
                           else n + floatDigits x
    where (m, n) = decodeFloat x

  significand x = encodeFloat m (- floatDigits x)
    where (m, _) = decodeFloat x

  scaleFloat k x = encodeFloat m (n + k)
    where (m, n) = decodeFloat x

  atan2 y x | x > 0           = atan (y / x)
            | x == 0 && y > 0 = pi/2
            | x < 0 && y > 0  = pi + atan (y / x)
            | (x <= 0 && y < 0) ||
              (x < 0 && isNegativeZero y) ||
              (isNegativeZero x &&
               isNegativeZero y)          = - atan2 (-y) x
            | y == 0 && (x < 0 ||
                         isNegativeZero x) = pi
            | x == 0 && y == 0             = y
            | otherwise                    = x + y
```

Экземпляры: `Float`, `Double`.

Класс: `RealFrac`

Описание: класс, объединяющий в себе свойства классов `Real` и `Fractional`, но при этом дополнительно описывающий некоторые функции для работы с числами, представленными в виде значений с плавающей точкой, а именно функции для округления величин.

Определение:

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


  truncate x = m where (m, _) = properFraction x


  round x = let (n, r) = properFraction x
                m      = if (r < 0) then n - 1
                                    else n + 1
            in case (signum (abs r - 0.5)) of
                 -1 -> n
                 0  -> if (even n) then n
                                   else m
                 1  -> m


  ceiling x = if (r > 0) then n + 1
                         else n
    where (n, r) = properFraction x


  floor x = if (r < 0) then n - 1
                       else n
    where (n, r) = properFraction x
```

Экземпляры: `Float`, `Double`, `Ratio a`.

Класс: `Show`

Описание: шаблон для типов, элементы которых имеют графически представляемую форму. Это означает, что все элементы таких типов можно передать в функцию `show` для вывода на экран (в консоль).

Определение:

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

```
class Show a where
  show      :: a -> String
  showsPrec :: Int -> a -> ShowS
  showList  :: [a] -> ShowS


  show x          = showsPrec 0 x ""
  showsPrec _ x s = show x ++ s
  showList []     = showString "[]"
  showList (x:xs) = showChar '[' . shows x . showl xs
    where showl []     = showChar ']'
          showl (x:xs) = showChar ',' . shows x . showl xs
```

Экземпляры: `()`, `Bool`, `Char`, `Maybe`, `Either`, `Ordering`, `[a]`, `(a, b)`, `Int`, `Integer`, `Float`, `Double`, `Ratio a`, `IO`, `IOError`.

6.3. Prelude: Функции

Функция: `abs`
Описание: возвращает модуль заданного числа.
Определение:

```
abs :: Num a => a -> a
abs x | x >= 0    = x
      | otherwise = -x
```

Функция: `absReal`
Описание: возвращает модуль заданного числа. Используется вместо функции `abs` в определениях методов класса `Num`.
Определение:

```
absReal :: Ord a => a -> a
absReal x | x >= 0    = x
          | otherwise = -x
```

Функция: `all`
Описание: при применении к предикату и списку возвращает `True`, если все элементы заданного списка удовлетворяют предикату, и `False` в противном случае. Аналогична функции `any`.

Определение:

```
all :: (a -> Bool) -> [a] -> Bool
all p xs = and (map p xs)
```

Функция: `and`

Описание: осуществляет конъюнкцию всех значений заданного булевского списка (см. также `or`).

Определение:

```
and :: [Bool] -> Bool
and xs = foldr (&&) True xs
```

Функция: `any`

Описание: при применении к предикату и списку возвращает `True`, если все элементы заданного списка удовлетворяют предикату, и `False` в противном случае. Синоним функции `all`.

Определение:

```
any :: (a -> Bool) -> [a] -> Bool
any p xs = and (map p xs)
```

Функция: `appendFile`

Описание: функция для дозаписывания информации в заданный файл.

Определение:

```
appendFile :: FilePath -> String -> IO ()
```

Функция определена в виде примитива.

Функция: `approxRational`

Описание: функция для аппроксимации рационального числа с заданной точностью.

Определение:

```
approxRational :: RealFrac a => a -> a -> Rational
approxRational x eps = simplest (x - eps) (x + eps)
  where
    simplest x y | y < x = simplest y x         | x == y = xr
                 | x > 0 = simplest' n d n' d'  | y < 0  = - simplest' (-n') d' (-n) d
                 | otherwise = 0 :% 1
```

```
    where xr@(n :% d) = toRational x
          (n' :% d')  = toRational y


simplest' n d n' d' | r == 0    = q :% 1
                    | q /= q'   = (q + 1) :% 1
                    | otherwise = (q * n'' + d'') :% n''


    where (q, r)       = quotRem n d
          (q', r')     = quotRem n' d'
          (n'' :% d'') = simplest' d' r' d r
```

Функция: asciiTab

Описание: функция, просто возвращающая таблицу символов ASCII.

Определение:

```
asciiTab :: [(String, String)]
asciiTab = zip ['\NUL'..' ']
               ["NUL", "SOH", "STX", "ETX", "EOT", "ENQ", "ACK", "BEL",
                "BS",  "HT",  "LF",  "VT",  "FF",  "CR",  "SO",  "SI",
                "DLE", "DC1", "DC2", "DC3", "DC4", "NAK", "SYN", "ETB",
                "CAN", "EM",  "SUB", "ESC", "FS",  "GS",  "RS",  "US",
                "SP"]
```

Функция: asTypeOf

Описание: синоним функции const для приведения типов.

Определение:

```
asTypeOf :: a -> a -> a
asTypeOf = const
```

Функция: atan

Описание: тригонометрическая функция для вычисления арктангенса заданного числа.

Определение:

```
atan :: Floating a => a -> a
```

Функция определена в виде примитива.

Функция: `break`

Описание: принимает на вход предикат и список, разбивает входной список на два выходных списка, возвращаемых в виде кортежа. Точкой разделения исходного списка служит первый элемент, для которого заданный предикат принимает истинное значение. Если предикат не выполняется ни для одного из элементов, то первым элементом кортежа является исходный список целиком, а вторым — пустой список.

Определение:

```
break :: (a -> Bool) -> [a] -> ([a], [a])
break p xs = span p' xs
  where p' x = not (p x)
```

Функция: `catch`

Описание: функция для «связывания» действия ввода/вывода с обработчиком ошибки, которая может произойти во время этого действия.

Определение:

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

Функция определена в виде примитива.

Функция: `ceiling`

Описание: возвращает наименьшее целое, которое не меньше аргумента. Эта функция связана с функцией `floor`.

Определение:

```
ceiling :: (RealFrac a, Integer b) => a -> b
```

Функция определена в виде примитива.

Функция: `chr`

Описание: получает на вход целое в промежутке от 0 до 255, возвращает символ, кодом которого является это целое. Является функцией, обратной функции `ord`. Если функция будет применена к целому числу, находящемуся за пределами данного интервала, то в результате возникнет ошибка.

Определение:

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

Функция определена в виде примитива.

Функция: `concat`

Описание: получает на вход список списков, объединяет их с использованием оператора (++).

Определение:

```
concat :: [[a]] -> [a]
concat xs = foldr (++) [] xs
```

Функция: `concatMap`

Описание: функция, совмещающая в себе действия функций `map` и `concat`. Получает на вход функцию, возвращающую список списков, а также исходный список, к которому применяется заданная функция. Результат её работы сращивается конкатенацией (++).

Определение:

```
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f
```

Функция: `const`

Описание: функция, возвращающая свой первый аргумент при заданных двух.

Определение:

```
const :: a -> b -> a
const k _ = k
```

Функция: `cos`

Описание: тригонометрическая функция косинуса, аргументы которой считаются заданными в радианах.

Определение:

```
cos :: Floating a => a -> a
```

Функция определена в виде примитива.

Функция: `curry`

Описание: функция для получения каррированной функции из некаррированной. Получает на вход некаррированную функцию и два её аргумента в обычном для языка Haskell стиле. Возвращает результат заданной функции на этих аргументах, собранных в пару (кортеж). Является обратной по действию к функции `uncurry`.

Определение:

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

Функция: `cycle`

Описание: функция, осуществляющая бесконечное применение конкатенации (++) к заданному списку. В результате получается бесконечный список, состоящий из элементов первоначального списка.

Определение:

```
cycle :: [a] -> [a]
cycle [] = error "Prelude.cycle: empty list"
cycle xs = xs'
  where xs' = xs ++ xs'
```

Функция: `denominator`

Описание: возвращает знаменатель дробного числа. Работает в паре с функцией `numerator`.

Определение:

```
denominator :: Integral a => Ratio a -> a
denominator (x :% y) = y
```

Функция: `digitToInt`

Описание: преобразует символьное представление одной цифры в соответствующее целое значение.

Определение:

```
digitToInt :: Char -> Int
digitToInt c | isDigit c = fromEnum c - fromEnum '0'
             | c >= 'a' && c <= 'f' = fromEnum c - fromEnum 'a' + 10
             | c >= 'A' && c <= 'F' = fromEnum c - fromEnum 'A' + 10
             | otherwise = error "Char.digitToInt: not a digit"
```

Функция: `div`

Описание: выполняет целочисленное деление своих целых аргументов и возвращает результат этой операции.

Определение:

```
div :: Integral a => a -> a -> a
```

Функция определена в виде примитива.

Функция: `doReadFile`

Описание: получает на вход строку с именем файла, возвращает строку с его содержимым. Возвращает ошибку, если файл не может быть открыт или не найден.

Определение:

```
doReadFile :: String -> String
```

Функция определена в виде примитива.

Функция: `doubleToFloat`

Описание: функция для приведения числа типа `Double` к числу одинарной точности (тип `Float`).

Определение:

```
doubleToFloat :: Double -> Float
```

Функция определена в виде примитива.

Функция: `doubleToRatio`

Описание: функция для приведения числа типа `Double` к виду обычной дроби.

Определение:

```
doubleToRatio :: Integral a => Double -> Ratio a
doubleToRatio x | n >= 0    = (fromInteger m * fromInteger b ^ n) % 1
                | otherwise = fromInteger m % (fromInteger b ^ (-n))
  where (m, n) = decodeFloat x
        b      = floatRadix x
```

Функция: `doubleToRational`

Описание: функция для преобразования числа типа `Double` (действительного) к рациональному.

Определение:

```
doubleToRational :: Double -> Rational
```

Функция определена в виде примитива.

Функция: `drop`

Описание: принимает на вход целое число и список, возвращает список, из начала которого удалено указанное первым аргументом число элементов. Если число элементов списка меньше, чем требуется удалить из начала, то возвращается пустой список.

Определение:

```
drop :: Int -> [a] -> [a]
drop 0 xs             = xs
drop _ []             = []
drop n (_:xs) | n > 0 = drop (n - 1) xs
drop _ _              = error "PreludeList.drop: negative argument"
```

Функция: `dropWhile`

Описание: принимает на вход некоторый предикат и список, удаляет элементы из начала списка до тех пор, пока удаляемые элементы удовлетворяют предикату.

Определение:

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

Функция: `either`

Описание: функция для работы со значениями типа `Either`. Применяет заданную функцию к левой или правой части значения и возвращает результат этой функции.

Определение:

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either l r (Left x)  = l x
either l r (Right y) = r y
```

Функция: `elem`

Описание: принимает на вход значение и список; возвращает `True`, если заданное значение принадлежит списку, и `False` в противном случае. Элементы списка должны иметь тот же тип, что и значение.

Определение:

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

Функция: `error`

Описание: принимает на вход строку, создает значение-ошибку с прикрепленным сообщением. Ошибка эквивалентна неопределённому значению ($\perp$). Любая попытка доступа к подобному значению приводит к завершению программы. Сообщение выводится на экран для отладки.

Определение:

```
error :: String -> a
```

Функция определена в виде примитива.

Функция: `exp`

Описание: вычисляет экспоненту (значение `exp n` эквивалентно e^n).

Определение:

```
exp :: Floating a => a -> a
```

Функция определена в виде примитива.

Функция: `filter`

Описание: принимает на вход предикат и список, возвращает список, содержащий все элементы исходного списка, для которых предикат является истинным.

Определение:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [k | k <- xs, p k]
```

Функция: `flip`

Описание: применяется к бинарным функциям. Возвращает значение заданной функции, подсчитанное на аргументах в обратном порядке.

Определение:

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

Функция: `floatProperFraction`

Описание: функция для получения из действительного числа пары, состоящей из целой и дробной его частей.

Определение:

```
floatProperFraction :: (RealFrac a, Integral b) => a -> (b, a)
floatProperFraction x | n >= 0    = (fromInteger m * fromInteger b ^ n, 0)
                      | otherwise = (fromInteger w, encodeFloat r n)
  where (m, n) = decodeFloat x
        b      = floatRadix x
        (w, r) = quotRem m (b ^ (-n))
```

Функция: `floatToRational`

Описание: функция для преобразования действительного числа типа `Float` в рациональное.

Определение:

```
floatToRational :: Float  -> Rational
```

Функция определена в виде примитива.

Функция: `floor`

Описание: возвращает наибольшее целое число, которое не больше заданного аргумента. С этой функцией связана функция `ceiling`.

Определение:

```
floor :: (RealFrac a, Integral b) => a -> b
```

Функция определена в виде примитива.

Функция: `foldl`

Описание: сворачивает заданный список с использованием заданного бинарного оператора и начального значения (свёртка производится по ассоциации влево).

Определение:

```
foldl :: (a -> b -> a) -> a -> [b] -> a1
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Функция: `foldl'`

Описание: строгий аналог функции `foldl`. Делает то же самое.

Определение:

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f a []     = a
foldl' f a (x:xs) = (foldl' f $! f a x) xs
```

Функция: `foldl1`

Описание: левоассоциативная свёртка непустых списков. В качестве начального значения берётся голова списка. См. функцию `foldl`.

Определение:

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs
```

Функция: `foldr`

Описание: сворачивает заданный список с использованием заданного бинарного оператора и начального значения (свёртка производится по ассоциации вправо).

Определение:

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

Функция: `foldr1`

Описание: правоассоциативная свёртка для непустых списков. В качестве начального значения берётся голова списка. См. функцию `foldr`.

Определение:

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr 1 f xs)
```

Функция: `fromInt`

Описание: преобразует число из типа `Int` в целочисленный тип из класса `Num`.

Определение:

```
fromInt :: Num a => Int -> a
```

Функция определена в виде примитива.

Функция: `fromInteger`

Описание: преобразует число из типа `Integer` в целочисленный тип из класса `Num`.

Определение:

```
fromInteger :: Num a => Integer -> a
```

Функция определена в виде примитива.

Функция: `fromIntegral`

Описание: функция для преобразования заданного числа в значение перечислимого множества. Осуществляет простое преобразование в целое число и обратно.

Определение:

```
fromIntegral :: (Integral a, Num b) => a -> b
fromIntegral = fromInteger . toInteger
```

Функция: `fst`

Описание: возвращает первый элемент кортежа, состоящего из двух элементов. См. также описание функции `snd`.

Определение:

```
fst :: (a, b) -> a
fst (x, _) = x
```

Функция: `gcd`

Описание: функция для получения наибольшего общего делителя заданного числа. Связана с функцией `lcm`.

Определение:

```
gcd :: Integral a => a -> a -> a
gcd 0 0 = error "Prelude.gcd: 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)
```

Функция: `getChar`

Описание: функция для чтения из стандартного потока ввода одного символа.

Определение:

```
getChar :: IO Char
```

Функция определена в виде примитива.

Функция: `getContents`

Описание: функция для чтения содержимого файла в строку символов.

Определение:

```
getContents :: IO String
```

Функция определена в виде примитива.

Функция: `getLine`

Описание: функция для получения строки с клавиатуры. Возвращает строку, обернутую в монаду `IO`.

Определение:

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

Функция: `head`

Описание: возвращает первый элемент непустого списка. При применении к пустому списку результатом будет выведено сообщение об ошибке.

Определение:

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

Функция: `id`

Описание: функция тождества, возвращает значение своего аргумента.

Определение:

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

Функция: `init`

Описание: возвращает список без последнего аргумента. Исходный список должен содержать, по крайней мере, один элемент. На пустом списке функция генерирует сообщение об ошибке.

Определение:

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

Функция: `interact`

Описание: функция для применения к содержимому файла некоторой заданной функции. Считывает весь файл в одну строку, после чего применяет к ней переданную в качестве аргумента функцию.

Определение:

```
interact :: (String -> String) -> IO ()
interact f = getContents >>= (putStr . f)
```

Функция: `intToDigit`

Описание: функция для возвращения символьного представления цифры. Работает на натуральных числах от 0 до 15.

Определение:

```
intToDigit :: Int -> Char
intToDigit i | i >= 0  && i <=  9 = toEnum (fromEnum '0' + i)
             | i >= 10 && i <= 15 = toEnum (fromEnum 'a' + i - 10)
             | otherwise          = error "Char.intToDigit: not a digit"
```

Функция: `intToRatio`

Описание: функция для преобразования целого числа в рациональное. В качестве числителя берется заданное число, в качестве знаменателя — 1.

Определение:

```
intToRatio :: Integral a => Int -> Ratio a
intToRatio x = fromInt x :% 1
```

Функция: `ioError`

Описание: функция для обработки ошибок, связанных с действиями ввода/вывода.

Определение:

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

Функция определена в виде примитива.

Функция: `isAlpha`

Описание: принимает на вход некоторый символ. Возвращает `True`, если это алфавитный символ, и `False` в противном случае.

Определение:

```
isAlpha :: Char -> Bool
isAlpha c = isUpper c || isLower c
```

Функция: `isAlphaNum`

Описание: предикат для определения, является ли заданный символ цифрой или алфавитным символом. Возвращает `True`, если таковым является.

Определение:

```
isAlphaNum :: Char -> Bool
isAlphaNum c = isAlpha c || isDigit c
```

Функция: `isAscii`

Описание: предикат для определения, является ли заданный символ стандартным символом кодировки ASCII. Возвращает `True`, если таковым является.

Определение:

```
isAscii :: Char -> Bool
isAscii c = fromEnum c < 128
```

Функция: `isControl`

Описание: предикат для определения, является ли заданный символ контрольным. К контрольным символам относятся символы, чьи коды меньше кода пробела, а также символ удаления предыдущего символа. Возвращает `True`, если таковым является.

Определение:

```
isControl :: Char -> Bool
isControl c = c < ' ' || c == '\DEL'
```

Функция: `isDigit`

Описание: принимает на вход некоторый символ. Возвращает `True`, если это символьное представление цифры, и `False` в противном случае.

Определение:

```
isDigit :: Char -> Bool
isDigit c = c >= '0' && c <= '9'
```

Функция: `isHexDigit`

Описание: предикат для определения, является ли заданный символ шестнадцатеричной цифрой. Возвращает `True`, если таковым является.

Определение:

```
isHexDigit :: Char -> Bool
isHexDigit c = isDigit c ||
               c >= 'A' && c <= 'F' ||
               c >= 'a' && c <= 'f'
```

Функция: `isLower`

Описание: принимает на вход некоторый символ. Возвращает `True`, если это алфавитный символ в нижнем регистре (строчная буква латинского алфавита), и `False` в противном случае.

Определение:

```
isLower :: Char -> Bool
isLower c = c >= 'a' && c <= 'z'
```

Функция: `isOctDigit`

Описание: предикат для определения, является ли заданный символ восьмеричной цифрой. Возвращает `True`, если таковым является.

Определение:

```
isOctDigit :: Char -> Bool
isOctDigit c = c >= '0' && c <= '7'
```

Функция: `isPrint`

Описание: предикат для определения, является ли заданный символ печатаемым. Возвращает `True`, если таковым является.

Определение:

```
isPrint :: Char -> Bool
isPrint c = c >= ' ' &&  c <= '~'
```

Функция: `isSpace`

Описание: принимает на вход некоторый символ. Возвращает `True`, если этот символ является пробельным (пустым), и `False` в противном случае.

Определение:

```
isSpace :: Char -> Bool
isSpace c = c == ' '  ||
            c == '\t' ||
            c == '\n' ||
            c == '\r' ||
            c == '\f' ||
            c == '\v'
```

Функция: `isUpper`

Описание: принимает на вход некоторый символ. Возвращает `True`, если это алфавитный символ в верхнем регистре (заглавная буква латинского алфавита), и `False` в противном случае.

Определение:

```
isUpper :: Char -> Bool
isUpper c = c >= 'A' && c <= 'Z'
```

Функция: `iterate`

Описание: применяет заданную функцию ко второму аргументу и возвращает бесконечный список таких применений: `[x, f x, f (f x), ...]`.

Определение:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

Функция: `last`

Описание: применяется к непустому списку, возвращает последний элемент заданного списка.

Определение:

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

Функция: `lex`

Описание: функция для осуществления лексического анализа заданной строки, то есть выделения лексем во входном для анализа потоке символов. Возвращает пару строк — первым элементом пары является найденная лексема, вторым — остаток строки.

Определение:

```
lex :: ReadS String
lex ""                  = [("", "")]
lex (c:s) | isSpace c   = lex (dropWhile isSpace s)
lex (’’:s)              = [(’’:ch ++ "’", t) |
                           (ch, ’’:t) <- lexLitChar s,
                           ch /= "’"]
lex (’"’:s)             = [(’"’:str, t) |
                           (str,t) <- lexString s]
  where lexString (’"’:s) = [("\"", s)]
        lexString s       = [(ch ++ str, u) |
                             (ch, t) <- lexStrItem s,
                             (str, u) <- lexString t]
        lexStrItem (’\:’&’:s)           = [("\\&", s)]
        lexStrItem (’\:c:s) | isSpace c = [("", t) |
                      ’\:t <- [dropWhile isSpace s]]
        lexStrItem s = lexLitChar s
lex (c:s) | isSingle c = [([c], s)]
          | isSym c    = [(c:sym, t) |
                          (sym, t) <- [span isSym s]]
          | isAlpha c  = [(c:nam, t) |
                          (nam, t) <- [span isIdChar s]]
          | isDigit c  = [(c:ds ++ fe, t) |
                          (ds, s) <- [span isDigit s],
                          (fe, t) <- lexFracExp s]
          | otherwise  = []
  where isSingle c =  c ‘elem‘ ",;()[]{}_‘"
        isSym c    =  c ‘elem‘ "!@#$%&*+./<=>?\\^|:-~"
        isIdChar c =  isAlphaNum c || c ‘elem‘ "_’"
        lexFracExp (’.’:s) = [(’.’:ds ++ e, u) |
                              (ds, t) <- lexDigits s,
                              (e, u) <- lexExp t]
        lexFracExp s = [("", s)]
        lexExp (e:s) | e ‘elem‘ "eE" = [(e:c:ds, u) |
                                        (c:t) <- [s],
                                        c ‘elem‘ "+-",
                                        (ds, u) <- lexDigits t] ++
```

```
                                        [(e:ds, t) |
                                          (ds, t) <- lexDigits s]
      lexExp s = [("", s)]
```

Функция: `lexDigits`

Описание: лексический анализатор для чисел. Возвращает, как и функция `lex`, пару строк — первым элементом пары является найденная лексема (число), вторым — остаток строки.

Определение:

```
lexDigits :: ReadS String
lexDigits = nonnull isDigit
```

Функция: `lexLitChar`

Описание: лексический анализатор для печатаемых символов. Возвращает, как и функция `lex`, пару строк — первым элементом пары является найденная лексема (символ), вторым — остаток строки.

Определение:

```
lexLitChar :: ReadS String
lexLitChar ('\:s) = [('\:esc, t) | (esc, t) <- lexEsc s]
  where lexEsc (c:s)     | c `elem` "abfnrtv\\\"'" = [([c], s)]
        lexEsc ('^':c:s) | c >= '@' && c <= '_'    = [(['^', c], s)]
        lexEsc s@(d:_) | isDigit d = lexDigits s
        lexEsc s@(c:_) | isUpper c =
          let table = ('\DEL',"DEL") : asciiTab
          in case [(mne, s') | (c, mne) <- table,
                               ([], s') <- [lexmatch mne s]]
             of (pr:_) -> [pr]
             []     -> []
        lexEsc _ = []
lexLitChar (c:s) = [([c], s)] lexLitChar ""    = []
```

Функция: `lexmatch`

Описание: функция для тестирования на схожесть начала заданных списков (строк). Возвращает пару списков, полученных из входных отсечением начальных совпадающих частей. Если входные списки полностью совпадают, то возвращается пара пустых списков. Если входные списки полностью разнятся, то они склеиваются в пару без изменений.

Определение:

```
lexmatch :: (Eq a) => [a] -> [a] -> ([a], [a])
lexmatch (x:xs) (y:ys) | x == y  = lexmatch xs ys
lexmatch xs ys                   = (xs, ys)
```

Функция: `lcm`

Описание: возвращает наибольшее общее кратное двух целочисленных аргументов.

Определение:

```
lcm :: Integral a => a -> a -> a
lcm _ 0 = 0
lcm 0 _ = 0
lcm x y = abs ((x `quot` gcd x y) * y)
```

Функция: `length`

Описание: возвращает число элементов в ограниченном списке.

Определение:

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

Функция: `lines`

Описание: применяется к списку символов, содержащему переносы строки. Возвращает список списков, разрывая исходный список в строки, используя символ переноса строки в качестве разделителя. Символы переноса строки вырезаются из исходного списка. Данная функция является обратной функции `unlines`.

Определение:

```
lines :: String -> [String]
lines []     = []
lines (x:xs) = l : ls
  where (l, xs') = break (== '\n') (x:xs)
        ls | xs' == [] = []
           | otherwise = lines (tail xs')
```

Функция: `log`

Описание: возвращает натуральный логарифм своего аргумента.

Определение:

```
log :: Floating a => a -> a
```

Функция определена в виде примитива.

Функция: `lookup`

Описание: функция для поиска ассоциированного значения с заданным в отображении. Отображение задается списком вида (значение, ассоциированное значение). Результат возвращается в виде значения типа `Maybe`. Значение `Nothing` возвращается, когда поиск неуспешен.

Определение:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
lookup k []                         = Nothing
lookup k ((x, y):xys) | k == x      = Just y
                      | otherwise = lookup k xys
```

Функция: `maybe`

Описание: функция для применения другой заданной функции к значению, упакованному в контейнерный тип `Maybe`. Возвращает обычное значение. По выполняемым вычислениям похожа на функцию `either`.

Определение:

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  = n
maybe n f (Just x) = f x
```

Функция: `map`

Описание: принимает на вход функцию и список любого типа. Возвращает список, где каждый элемент является результатом применения функции к соответствующему элементу исходного списка.

Определение:

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

Функция: `mapM`

Описание: функция для применения заданной функции ко всем элементам монады, в которую упакован исходный список. Заданная первым элементом функция

оборачивает свой результат в монадический тип, после чего список этих монадических результатов разворачивается в монаду, в которой содержится список.
Определение:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f = sequence . map f
```

Функция: mapM-
Описание: выполняет то же самое, что и функция mapM, но не возвращает результата. Используется только тогда, когда результат не важен, но важны только побочные эффекты, предоставляемые монадой.
Определение:

```
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f = sequence_ . map f
```

Функция: max
Описание: принимает на вход два значения одного типа, значения которого можно сравнивать. Возвращает максимальное значение из двух заданных в соответствии с действием оператора (>=).
Определение:

```
max :: Ord a => a -> a -> a
max x y | x >= y    = x
        | otherwise = y
```

Функция: maximum
Описание: применяется к непустому списку, для элементов которого определены операции сравнения. Возвращает максимальное значение из исходного списка.
Определение:

```
maximum :: Ord a => [a] -> a
maximum xs = foldl1 max xs
```

Функция: min
Описание: принимает на вход два значения одного типа, значения которого можно сравнивать. Возвращает минимальное значение из двух заданных в соответствии с действием оператора (<=).

Определение:

```
min :: Ord a => a -> a -> a
min x y | x <= y    = x
        | otherwise = y
```

Функция: `minimum`
Описание: применяется к непустому списку, для элементов которого определены операции сравнения. Возвращает минимальное значение из исходного списка.
Определение:

```
minimum :: Ord a => [a] -> a
minimum xs = foldl1 min xs
```

Функция: `mod`
Описание: возвращает остаток от деления одного целочисленного аргумента на другой.
Определение:

```
mod :: Integral a => a -> a -> a
```

Функция определена в виде примитива.

Функция: `nonnull`
Описание: функция для получения пары, состоящей из начала заданной строки (второй аргумент функции), удовлетворяющей заданному предикату (первый аргумент), и остатка строки. Используется в функциях для лексического анализа серии `lex`.
Определение:

```
nonnull :: (Char -> Bool) -> ReadS String
nonnull p s = [(cs, t) | (cs@(_:_), t) <- [span p s]]
```

Функция: `not`
Описание: возвращает логическое отрицание от булевского аргумента.
Определение:

```
not :: Bool -> Bool
not True  = False
not False = True
```

Функция: `null`

Описание: предикат для определения того, является ли заданный список пустым или нет. Возвращает `True` на пустых списках.

Определение:

```
null :: [a] -> Bool
null []    = True
null (_:_) = False
```

Функция: `numerator`

Описание: функция для получения числителя из дробного значения. Работает в паре с функцией `denominator`.

Определение:

```
numerator :: Integral a => Ratio a -> a
numerator (x :% y) = x
```

Функция: `numericEnumFrom`

Описание: функция для получения списка значений из некоторого упорядоченного перечислимого множества, начинающихся с заданного значения. По сути, строит неограниченную арифметическую прогрессию, начиная с заданного элемента.

Определение:

```
numericEnumFrom :: Real a => a -> [a]
numericEnumFrom n = n : (numericEnumFrom $! (n + 1))
```

Функция: `numericEnumFromThen`

Описание: функция для получения списка значений из некоторого упорядоченного перечислимого множества, начинающихся с заданного значения, после которого идёт второе заданное значение (на основании двух этих значений вычисляется разность, при помощи которой определяются все последующие значения получаемого списка). По сути, строит неограниченную арифметическую прогрессию, начиная с заданного элемента и в соответствии с вычисленной разностью.

Определение:

```
numericEnumFromThen :: Real a => a -> a -> [a]
numericEnumFromThen n m = iterate ((m - n) +) n
```

Функция: `numericEnumFromTo`

Описание: функция для получения списка значений из некоторого упорядоченного перечислимого множества, начинающихся с заданного значения и оканчивающихся вторым заданным значением. По сути, строит ограниченную арифметическую прогрессию, начиная с первого заданного элемента и заканчивая вторым.
Определение:

```
numericEnumFromTo :: Real a => a -> a -> [a]
numericEnumFromTo n m = takeWhile (<= m) (numericEnumFrom n)
```

Функция: `numericEnumFromThenTo`

Описание: функция для получения списка значений из некоторого упорядоченного перечислимого множества, начинающихся с заданного значения, после которого идёт второе заданное значение (на основании двух этих значений вычисляется разность, при помощи которой определяются все последующие значения получаемого списка), а сам список ограничен третьим заданным значением. По сути, строит ограниченную арифметическую прогрессию, начиная с заданного элемента и в соответствии с вычисленной разностью и заканчивая третьим заданным значением.
Определение:

```
numericEnumFromThenTo :: Real a => a -> a -> a -> [a]
numericEnumFromThenTo n n' m  = takeWhile p (numericEnumFromThen n n')
  where p | n' >= n   = (<= m)
          | otherwise = (>= m)
```

Функция: `or`

Описание: применяется к списку булевских значений, возвращает их дизъюнкцию (см. также описание функции `and`).
Определение:

```
or :: [Bool] -> Bool
or xs = foldr (||) False xs
```

Функция: `ord`

Описание: применяется к символу, возвращает его код ASCII как значение типа `Integer`.

Определение:

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

Функция определена в виде примитива.

Функция: `otherwise`

Описание: синоним значения булевского значения `True`, который используется при определении функций через охрану. Создан специально для того, чтобы определения функций соответствовали математической нотации.

Определение:

```
otherwise :: Bool
otherwise = True
```

Функция: `pi`

Описание: возвращает отношение длины окружности к её диаметру (число π).

Определение:

```
pi :: Floating a => a
```

Функция определена в виде примитива.

Функция: `putChar`

Описание: функция для вывода в стандартный поток вывода заданного символа.

Определение:

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

Функция определена в виде примитива.

Функция: `putStr`

Описание: принимает строку в качестве аргумента и возвращает действие ввода/вывода в качестве результата. Побочным эффектом применения функции `putStr` является вывод заданной строки на экран.

Определение:

```
putStr :: String -> IO ()
```

Функция определена в виде примитива.

Функция: putStrLn

Описание: функция, выводящая на экран заданную строку и завершающая её вывод символом перевода строки. По своему эффекту тождественна функции putStr, за исключением описанного нюанса.

Определение:

```
putStrLn :: String -> IO ()
putStrLn s = do putStr s
                putChar '\n'
```

Функция: print

Описание: функция для вывода на экран заданного значения, которое может быть выведено на экран (тип такого значения должен быть экземпляром класса Show).

Определение:

```
print :: Show a => a -> IO ()
print = putStrLn . show
```

Функция: product

Описание: применяется к списку чисел, возвращает произведение всех чисел, входящих в список.

Определение:

```
product :: Num a => [a] -> a
product xs = foldl (*) 1 xs
```

Функция: protectEsc

Описание: функция для «защиты» символов, выраженных при помощи escape-последовательностей. Заменяет символ (\) на два таких символа.

Определение:

```
protectEsc :: (Char -> Bool) -> ([Char] -> a) -> [Char] -> a
protectEsc p f = f . cont
  where cont s@(c:_) | p c = "\\&" ++ s
        cont s             = s
```

Функция: rationalToDouble

Описание: функция для преобразования заданного рационального числа в действительное двойной точности (тип Double).

Определение:

```
rationalToDouble :: Rational -> Double
```

Функция определена в виде примитива.

Функция: `rationalToFloat`

Описание: функция для преобразования заданного рационального числа в действительное одинарной точности (тип `Float`).

Определение:

```
rationalToFloat :: Rational -> Float
```

Функция определена в виде примитива.

Функция: `rationalToRealFloat`

Описание: функция для преобразования дробного значения, составленного из двух целых чисел, в действительное число.

Определение:

```
rationalToRealFloat :: RealFloat a => Ratio Integer -> a
rationalToRealFloat x = x'
  where x'     = f e
        f e    = if (e' == e)
                    then y
                    else f e'
    where y       = encodeFloat (round (x * (1 % b)^^e)) e
          (_, e') = decodeFloat y
        (_, e) = decodeFloat (fromInteger (numerator x) `asTypeOf` x' /
                              fromInteger (denominator x))
        b      = floatRadix x'
```

Функция: `read`

Описание: функция для получения из заданной строки определённого значения, которое может быть представлено при помощи строки (типы таких значений должны быть экземплярами класса `Read`).

Определение:

```
read :: Read a => String -> a
read s =  case [x | (x, t) <- reads s,
                    ("", "") <- lex t] of
            [x] -> x
            []  -> error "Prelude.read: no parse"
            _   -> error "Prelude.read: ambiguous parse"
```

Функция: readDec

Описание: функция для получения из заданной строки беззнакового числа в десятичном представлении.

Определение:

```
readDec :: Integral a => ReadS a
readDec = readInt 10 isDigit (\d -> fromEnum d - fromEnum '0')
```

Функция: readHex

Описание: функция для получения из заданной строки беззнакового числа в шестнадцатеричном представлении.

Определение:

```
readHex :: Integral a => ReadS a
readHex = readInt 16 isHexDigit hex
  where hex d = fromEnum d - (if (isDigit d)
                              then fromEnum '0'
                              else fromEnum (if (isUpper d)
                                             then 'A'
                                             else 'a') - 10)
```

Функция: readField

Описание: функция для получения из строки определённых полей данных в зависимости от их типов.

Определение:

```
readField :: Read a => String -> ReadS a
readField m s0 = [r | (t, s1) <- lex s0,
                      t == m,
                      ("=", s2) <- lex s1,
                      r <- reads s2]
```

Функция: `readFile`

Описание: функция для полного чтения содержимого файла.

Определение:

```
readFile :: FilePath -> IO String
```

Функция определена в виде примитива.

Функция: `readFloat`

Описание: функция для получения из заданной строки числового значения с плавающей точкой.

Определение:

```
readFloat :: RealFloat a => ReadS a
readFloat r = [(fromRational ((n % 1) * 10^^(k - d)), t) |
               (n, d, s) <- readFix r,
               (k, t) <- readExp s]
  where readFix r = [(read (ds ++ ds'), length ds', t) |
                     (ds, s) <- lexDigits r,
                     (ds', t) <- lexFrac s]
        lexFrac ('.':s) = lexDigits s
        lexFrac s       = [("", s)]
        readExp (e:s) | e `elem` "eE" = readExp' s
        readExp s                     = [(0, s)]
        readExp' ('-':s) = [(-k, t) | (k, t) <- readDec s]
        readExp' ('+':s) = readDec s
        readExp' s       = readDec s
```

Функция: `readInt`

Описание: функция для получения из заданной строки целочисленного значения в заданном базисе.

Определение:

```
readInt :: Integral a => a -> (Char -> Bool) -> (Char -> Int) -> ReadS a
readInt radix isDig digToInt s = [(foldl1 (\n d -> n * radix + d)
                                          (map (fromIntegral . digToInt)
                                          ds), r) |
                                  (ds, r) <- nonnull isDig s]
```

Функция: readIO

Описание: функция, выполняющая те же действия, что и функция read, но в случаях ошибочных ситуаций не останавливающая вычислительный процесс, а выбрасывающая исключение.

Определение:

```
readIO :: Read a => String -> IO a
readIO s = case [x | (x, t) <- reads s,
                     ("", "") <- lex t] of
  [x] -> return x
  []  -> ioError (userError "PreludeIO.readIO: no parse")
  _   -> ioError (userError "PreludeIO.readIO: ambiguous parse")
```

Функция: readLitChar

Описание: функция для получения из заданной строки набора символов, которые могут быть отображены на экране.

Определение:

```
readLitChar :: ReadS Char
readLitChar ('\:s) = readEsc s
  where readEsc ('a':s)  = [('\a', s)]
        readEsc ('b':s)  = [('\b', s)]
        readEsc ('f':s)  = [('\f', s)]
        readEsc ('n':s)  = [('\n', s)]
        readEsc ('r':s)  = [('\r', s)]
        readEsc ('t':s)  = [('\t', s)]
        readEsc ('v':s)  = [('\v', s)]
        readEsc ('\:s) = [('\, s)]
        readEsc ('"':s)  = [('"', s)]
        readEsc (''':s) = [(''', s)]
        readEsc ('^':c:s) | c >= '@' && c <= '_'
          = [(toEnum (fromEnum c - fromEnum '@'), s)]
        readEsc s@(d:_) | isDigit d = [(toEnum n, t) |
                                       (n,t) <- readDec s]
        readEsc ('o':s)  = [(toEnum n, t) |
                            (n, t) <- readOct s]
        readEsc ('x':s)  = [(toEnum n, t) |
                            (n, t) <- readHex s]
        readEsc s@(c:_) | isUpper c
          = let table = ('\DEL',"DEL") : asciiTab
            in case [(c, s') | (c, mne) <- table,
                               ([], s') <- [lexmatch mne s]]
               of (pr:_) -> [pr]
```

```
                []     -> []
      readEsc _        = []
readLitChar (c:s)    = [(c, s)]
```

Функция: `readLn`

Описание: функция для чтения некоторого значения, которое имеет символьное представление, с клавиатуры. Тип читаемого значения должен являться экземпляром класса `Read`.

Определение:

```
readLn :: Read a => IO a
readLn = do l <- getLine
            r <- readIO l
            return r
```

Функция: `readOct`

Описание: функция для получения из заданной строки беззнакового числа в восьмеричном представлении.

Определение:

```
readOct :: Integral a => ReadS a
readOct = readInt 8 isOctDigit (\d -> fromEnum d - fromEnum '0')
```

Функция: `readParen`

Описание: функция для получения из заданной строки некоторого значения, заключенного в скобки. Первое значение функции (булевское) определяет, важно ли само наличие скобок.

Определение:

```
readParen :: Bool -> ReadS a -> ReadS a
readParen b g = if (b) then mandatory
                       else optional
  where optional r  = g r ++ mandatory r
        mandatory r = [(x, u) | ("(", s) <- lex r,
                                (x, t) <- optional s,
                                (")", u) <- lex t]
```

Функция: `reads`

Описание: функция для чтения некоторого значения из заданной строки символов. Является синонимом метода `readsPrec` класса `Read`.

Определение:

```
reads :: Read a => ReadS a
reads = readsPrec 0
```

Функция: readSigned

Описание: функция для получения из заданной строки знакового числового значения.

Определение:

```
readSigned :: Real a => ReadS a -> ReadS a
readSigned readPos = readParen False read'
  where read' r  = read'' r ++ [(-x, t) | ("-", s) <- lex r,
                                          (x, t) <- read'' s]
        read'' r = [(n, s) | (str, s) <- lex r,
                             (n, "") <- readPos str]
```

Функция: realFloatToRational

Описание: функция для получения реального дробного значения действительного числа.

Определение:

```
realFloatToRational :: Real a => ReadS a -> ReadS a
realFloatToRational x = (m % 1) * (b % 1)^^n
  where (m, n) = decodeFloat x
        b      = floatRadix x
```

Функция: realToFrac

Описание: функция для перевода заданного действительного числа в рациональное представление.

Определение:

```
realToFrac :: (Real a, Fractional b) => a -> b
realToFrac = fromRational . toRational
```

Функция: reduce

Описание: функция для сокращения дроби. Возвращает дробь, которую нельзя сократить.

Определение:

```
reduce :: Integral a => a -> a -> Ratio a
reduce x y | y == 0    = error "Ratio.%: zero denominator"
           | otherwise = (x `quot` d) :% (y `quot` d)
  where d = gcd x y
```

Функция: `repeat`

Описание: принимает на вход некоторое значение, возвращает неограниченный список, составленный только из этого значения.

Определение:

```
repeat :: a -> [a]
repeat x = xs
  where xs = x:xs
```

Функция: `replicate`

Описание: принимает на вход целое число (положительное или 0) и некоторое значение, возвращает список, содержащий указанное количество копий этого значения.

Определение:

```
replicate :: Int -> a -> [a]
replicate n x = take n (repeat x)
```

Функция: `reverse`

Описание: применяется к ограниченному списку любого типа, возвращает список элементов исходного списка в обратном порядке.

Определение:

```
reverse :: [a] -> [a]
reverse xs = foldl (flip (:)) [] xs
```

Функция: `round`

Описание: округляет свой аргумент до ближайшего целого.

Определение:

```
round :: (RealFrac a, Integral b) => a -> b
```

Функция определена в виде примитива.

Функция: `scanl`

Описание: функция для получения списка применений некоторой заданной функции к элементам заданного списка. Работает так же, как и функция `foldl`, однако возвращает не только конечный результат, но и весь список промежуточных.

Определение:

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q xs = q : (case xs of
                      []   -> []
                      x:xs -> scanl f (f q x) xs)
```

Функция: `scanl1`

Описание: функция, делающая то же самое, что и функция `scanl`, но работающая на непустых списках. В качестве начального значения использует голову заданного списка.

Определение:

```
scanl1 :: (a -> a -> a) -> [a] -> [a]
scanl1 f (x:xs) = scanl f x xs
```

Функция: `scanr`

Описание: функция, осуществляющая правоассоциативное сканирование заданного списка (по аналогии с функцией `scanl`).

Определение:

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr f q0 []     = [q0] scanr f q0 (x:xs) = f x q : qs
  where qs@(q:_) = scanr f q0 xs
```

Функция: `scanr1`

Описание: функция, работающая так же, как и функция `scan`, но на непустых списках. В качестве начального значения используется голова списка.

Определение:

```
scanr1 :: (a -> a -> a) -> [a] -> [a]
scanr1 f [x]    = [x] scanr1 f (x:xs) = f x q : qs
  where qs@(q:_) = scanr1 f xs
```

Функция: `sequence`

Описание: функция для последовательного выполнения списка действий, обёрнутых определённой монадой. В качестве результата возвращает список значений, обёрнутый исходной монадой.

Определение:

```
sequence :: Monad m => [m a] -> m [a]
sequence []     = return []
sequence (c:cs) = do x  <- c
                     xs <- sequence cs
                     return (x:xs)
```

Функция: `sequence_`

Описание: функция, осуществляющая те же действия, что и функция `sequence`, но не возвращающая результата. Используется тогда, когда важным являются побочные эффекты, предоставляемые монадой, а не результаты вычислений.

Определение:

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

Функция: `show`

Описание: преобразует значение, тип которого должен быть из класса `Show`, в его строковое представление.

Определение:

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

Функция определена в виде примитива.

Функция: `showChar`

Описание: функция для преобразования заданного символа в строку. Используется в функции `show` для отображения символов.

Определение:

```
showChar :: Char -> ShowS
showChar = (:)
```

Функция: `showField`

Описание: функция для преобразования в строку определённого значения поля. Используется в функции `show` для отображения полей.

Определение:

```
showField :: Show a => String -> a -> ShowS
showField m v = showString m . showChar '=' . shows v
```

Функция: showInt

Описание: функция для преобразования в строку заданного целого положительного числа. Используется в функции show для отображения целых положительных чисел.

Определение:

```
showInt :: Integral a => a -> ShowS
showInt n r | n < 0 = error "Numeric.showInt: can't show negative numbers"
            | otherwise = let (n', d) = quotRem n 10
                              r'      = toEnum (fromEnum '0' +
                                                fromIntegral d) : r
                          in if (n' == 0) then r'
                                          else showInt n' r'
```

Функция: showLitChar

Описание: функция для преобразования в строку заданного символа, который может быть отображен на экране. Используется в функции show для отображения таких символов.

Определение:

```
showLitChar :: Char -> ShowS
showLitChar c | c > '\DEL' = showChar '\ .
                protectEsc isDigit (shows (fromEnum c))
showLitChar '\DEL'         = showString "\\DEL"
showLitChar '\             = showString "\\\\"
showLitChar c | c >= ' '   = showChar c
showLitChar '\a'           = showString "\\a"
showLitChar '\b'           = showString "\\b"
showLitChar '\f'           = showString "\\f"
showLitChar '\n'           = showString "\\n"
showLitChar '\r'           = showString "\\r"
showLitChar '\t'           = showString "\\t"
showLitChar '\v'           = showString "\\v"
showLitChar '\SO'          = protectEsc ('H' ==) (showString "\\SO")
showLitChar c = showString ('\ : snd (asciiTab !! fromEnum c))
```

Функция: showParen

Описание: функция для преобразования в строку заданного значения, обрамлённого в скобки. Первый входной аргумент булевского типа используется для указания того, обязательны ли скобки. Используется в функции show для отображения таких значений.

Определение:

```
showParen :: Bool -> ShowS -> ShowS
showParen b p = if (b) then showChar '(' . p . showChar ')'
                       else p
```

Функция: shows

Описание: функция для преобразования в строку некоторого заданного значения. Является синонимом метода showsPrec класса Show.

Определение:

```
shows :: Show a => a -> ShowS
shows = showsPrec 0
```

Функция: showSigned

Описание: функция для преобразования в строку заданного числа со знаком. Используется в функции show для отображения чисел со знаком.

Определение:

```
showSigned :: Real a => (a -> ShowS) -> Int -> a -> ShowS
showSigned showPos p x = if (x < 0)
                           then showParen (p > 6)
                                          (showChar '-' . showPos (-x))
                           else showPos x
```

Функция: showString

Описание: функция для преобразования заданной строки в строку. Используется в функции show для отображения строк.

Определение:

```
showString :: String -> ShowS
showString = (++)
```

Функция: sin

Описание: тригонометрическая функция для вычисления синуса. Аргумент принимается в радианах.

Определение:

```
sin :: Floating a => a -> a
```

Функция определена в виде примитива.

Функция: `signumReal`

Описание: функция для получения знака заданного числа.

Определение:

```
signumReal :: (Num a, Num b, Ord a) => a -> b
signumReal x | x == 0    =  0
             | x > 0     =  1
             | otherwise = -1
```

Функция: `snd`

Описание: возвращает второй элемент кортежа из двух элементов. См. также описание функции `fst`.

Определение:

```
snd :: (a, b) -> b
snd (_, y) = y
```

Функция: `sort`

Описание: сортирует список в возрастающем порядке. Элементы списка должны иметь тип, являющийся экземпляром класса `Ord`.

Определение:

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

Функция: `span`

Описание: получает на вход предикат и список, разделяет список на два, возвращаемые в виде кортежа, так что элементы в первом списке берутся из исходного, пока удовлетворяют заданному предикату, а элементы второго списка — остальные элементы списка.

Определение:

```
span :: (a -> Bool) -> [a] -> ([a], [a])
span p [] = ([], [])
span p xs@(x:xs') | p x       = (x:ys, zs)
                  | otherwise = ([], xs)
  where (ys, zs) = span p xs'
```

Функция: `splitAt`

Описание: получает на вход целое число (положительное или 0) и список, разделяет список на два, возвращаемых при помощи кортежа. Место разделения исходного списка соответствует заданному числу. Если целое больше, чем длина списка, функция возвращает исходный список в первом значении кортежа.

Определение:

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt 0 xs = ([], xs)
splitAt _ [] = ([], [])
splitAt n (x:xs) | n > 0 = (x:xs', xs>>)
  where (xs', xs>>) = splitAt (n - 1) xs
splitAt _ _ = error "PreludeList.splitAt: negative argument"
```

Функция: `sqrt`

Описание: возвращает квадратный корень из заданного числа.

Определение:

```
sqrt :: Floating a => a -> a
```

Функция определена в виде примитива.

Функция: `subtract`

Описание: вычитает первый аргумент из второго.

Определение:

```
subtract :: Num a => a -> a -> a
subtract = flip (-)
```

Функция: `sum`

Описание: складывает элементы ограниченного списка чисел.

Определение:

```
sum :: Num a => [a] -> a
sum xs = foldl (+) 0 xs
```

Функция: `tail`

Описание: применяется к непустому списку, возвращает список без его первого элемента.

Определение:

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

Функция: `take`

Описание: применяется к целому числу (положительному или 0) и списку, возвращает указанное количество элементов из начала списка.

Определение:

```
take :: Int -> [a] -> [a]
take 0 _  = []
take _ [] = []
take n (x:xs) | n > 0 = x : take (n - 1) xs
take _ _  = error "PreludeList.take: negative argument"
```

Функция: `takeWhile`

Описание: применяется к предикату и списку, возвращает список, содержащий элементы из начала списка, пока удовлетворяется предикат.

Определение:

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

Функция: `tan`

Описание: тригонометрическая функция тангенс, аргумент принимается в радианах.

Определение:

```
tan :: Floating a => a -> a
```

Функция определена в виде примитива.

Функция: `toLower`

Описание: преобразует алфавитный символ в верхнем регистре в соответствующий строчный алфавитный символ. Если функция применена к аргументу, который не является заглавным алфавитным символом, будет возвращен аргумент без изменений.

Определение:

```
toLower :: Char -> Char
toLower c | isUpper c = toEnum (fromEnum c -
                                fromEnum 'A' +
                                fromEnum 'a')
          | otherwise = c
```

Функция: `toUpper`

Описание: преобразует алфавитный символ в нижнем регистре в соответствующий заглавный алфавитный символ. Если функция применена к аргументу, который не является строчным алфавитным символом, аргумент будет возвращен без изменений.

Определение:

```
toUpper :: Char -> Char
toUpper c | isLower c = toEnum (fromEnum c -
                                fromEnum 'a' +
                                fromEnum 'A')
          | otherwise = c
```

Функция: `truncate`

Описание: удаляет дробную часть числа с плавающей точкой, оставляя только целую часть.

Определение:

```
truncate :: (RealFrac a, Integral b) => a -> b
```

Функция определена в виде примитива.

Функция: `uncurry`

Описание: функция для преобразования каррированной функции в некаррированную. Действует обратно эффекту функции `curry`.

Определение:

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f p = f (fst p) (snd p)
```

Функция: `undefined`
Описание: функция для представления неопределенных вычислений ($\perp$).
Определение:

```
undefined :: a
undefined | False = undefined
```

Функция: `unlines`
Описание: преобразует список строк в единую строку, вставляя символ переноса строки между ними. Это функция является обратной к функции `lines`.
Определение:

```
unlines :: [String] -> String
unlines xs = concat (map addNewLine xs)
  where addNewLine l = l ++ "\n"
```

Функция: `until`
Описание: функция для организации циклических вычислений заданной функции с передачей в качестве параметра на очередной итерации предыдущего вычисленного значения. В качестве сигнала об остановке цикла используется предикат, передаваемый первым аргументом. Когда его значение становится истинным, цикл останавливается.
Определение:

```
until :: (a -> Bool) -> (a -> a) -> a -> a
until p f x = if (p x) then x
                       else until p f (f x)
```

Функция: `unwords`
Описание: осуществляет конкатенацию списка строк в одну строку, вставляя пробелы между отдельными строками из исходного списка. Данная функция является обратной функции `words`.

Определение:

```
unwords :: [String] -> String
unwords [] = [] unwords ws = foldr1 addSpace ws
  where addSpace w s = w ++ (' ':s)
```

Функция: unzip

Описание: функция для преобразования списка пар в пару списков.

Определение:

```
unzip :: [(a, b)] -> ([a], [b])
unzip = foldr (\(a, b) ~(as, bs) -> (a:as, b:bs)) ([], [])
```

Функция: unzip3

Описание: функция для преобразования списка троек в тройку списков.

Определение:

```
unzip3 :: [(a, b, c)] -> ([a], [b], [c])
unzip3 = foldr (\(a, b, c) ~(as, bs, cs) -> (a:as, b:bs, c:cs))
               ([], [], [])
```

Функция: userError

Описание: функция, определяющая пользовательское сообщение об ошибке, возникающей в процессе выполнения действий ввода/вывода.

Определение:

```
userError :: String -> IOError
```

Функция определена в виде примитива.

Функция: words

Описание: разрывает строку на список слов, которые разделены одним или несколькими пробелами. Данная функция является обратной функции unwords.

Определение:

```
words :: String -> [String]
words s | findSpace == [] = []
        | otherwise       = w : words s>>
  where (w, s>>)  = break isSpace findSpace
        findSpace = dropWhile isSpace s
```

Функция: writeFile

Описание: функция для записи строки в некоторый файл (определяется по имени).

Определение:

```
writeFile :: FilePath -> String -> IO ()
```

Функция определена в виде примитива.

Функция: zip

Описание: применяется к двум спискам, возвращает список пар, каждая из которых сформирована из двух соответствующих элементов исходных списков. Если исходные списки имеют разную длину, длина результирующего списка будет как у наиболее короткого.

Определение:

```
zip :: [a] -> [b] -> [(a, b)]
zip xs ys = zipWith pair xs ys
  where pair x y = (x, y)
```

Функция: zip3

Описание: функция, осуществляющая те же действия, что и функция zip, но упаковывающая три значения в тройку.

Определение:

```
zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]
zip3 = zipWith3 (\a b c -> (a, b, c))
```

Функция: zipWith

Описание: применяется к бинарной функции и двум спискам, возвращает список значений, полученный применением функции к парам соответствующих значений списков.

Определение:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith z (a:as) (b:bs) = z a b : zipWith z as bs
zipWith _ _ _           = []
```

Функция: zipWith3
Описание: функция, осуществляющая те же действия, что и функция zip3, но упаковывающая три значения в тройку при помощи заданной функции.
Определение:

```
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
zipWith3 z (a:as) (b:bs) (c:cs) = z a b c : zipWith3 z as bs cs
zipWith3 _ _ _ _                = []
```

6.4. Prelude: Операторы

Операторы — это простые функции одного или двух аргументов. Бинарные (двухместные) операторы обычно ставятся между своих аргументов (инфиксная нотация), вместо того, чтобы ставиться слева, как это принято для функций. Многие операторы имеют символьное обозначение (например, (+)) для оператора сложения), однако можно писать и полные имена (для сложения — plus). Другие операторы имеют только текстовые названия (такие как div для осуществления целочисленного деления).

Таблица 6.1 перечисляет некоторые полезные операторы, определенные в стандартном модуле Prelude. Значения ассоциативности и приоритета для этих операторов также указаны в таблице.

Чем выше значение приоритета, тем сильнее оператор связан с аргументом, то есть тем раньше он выполняется в выражении. Применение функций имеет приоритет 10 и не уступает в этом вопросе ни одному оператору.

Ассоциативность — это последовательность применения операторов в выражении, определённая в языке Haskell для удобства создания функций. Без определения ассоциативности некоторые выражения могут быть двусмысленными. Например, выражение «8 - 2 - 1» можно интерпретировать двумя способами, каждый из который даст свой результат: «(8 - 2) - 1» или «8 - (2 - 1)». Ассоциативность определяет способ восстановления скобок в выражении, если они пропущены. Например, оператор вычитания имеет левую ассоциативность, поэтому транслятор языка Haskell выбирает первый способ интерпретации.

Выбор ассоциативности для представленных операторов является достаточно произвольным, однако, как правило, он совпадает с тем, как это принято в математике для соответствующей операции. Надо также отметить, что некоторые

операторы не являются ассоциативными и, как следствие, не могут быть применены в последовательности. Например, оператор равенства (==) не является ассоциативным, и поэтому следующее выражение не разрешено в языке Haskell: «2 == (1 + 1) == (3 - 1)».

Таблица 6.1. Операторы, определённые в стандартном модуле Prelude

Символ	Значение	Тип	Ас.	Пр.
!!	Индекс	[a] -> Int -> a	Л	9
.	Композиция	(a -> b) -> (c -> a) -> c -> b	П	9
^	Экспонента	(Integral b, Num a) => a -> b-> a	П	8
^^	Экспонента	(Fractional a, Integral b) => a -> b -> a	П	8
**	Экспонента	Floating a => a -> a -> a	П	8
*	Умножение	Num a => a -> a -> a	Л	7
/	Деление	Rational a => a -> a -> a	Л	7
quot	Целое деление	Num a => a -> a -> a	Л	7
rem	Остаток	Num a => a -> a -> a	Л	7
div	Целое деление	Num a => a -> a -> a	Л	7
mod	Остаток	Num a => a -> a -> a	Л	7
:%	Дробь	Integral a => a -> a -> Ratio a	Л	7
%	Сокращение	Integral a => a -> a -> Ratio a	Л	7
+	Сложение	Num a => a -> a -> a	Л	6
-	Вычитание	Num a => a -> a -> a	Л	6
:	Создание списка	a -> [a] -> [a]	П	5
++	Конкатенация	[a] -> [a] -> [a]	П	5
/=	Неравенство	Eq a => a -> a -> Bool	—	4
==	Равенство	Eq a => a -> a -> Bool	—	4
<	Меньше	Ord a => a -> a -> Bool	—	4
<=	Меньше или равно	Ord a => a -> a -> Bool	—	4
>	Больше	Ord a => a -> a -> Bool	—	4
>=	Больше или равно	Ord a => a -> a -> Bool	—	4
elem	Существование	Eq a => a -> [a] -> Bool	—	4
notElem	Несуществование	Eq a => a -> [a] -> Bool	—	4
&&	Логическое И	Bool -> Bool -> Bool	П	3
||	Логическое ИЛИ	Bool -> Bool -> Bool	П	2
>>	Связывание	m a -> m b -> m b	Л	1
>>=	Связывание	m a -> (a -> m b) -> m b	Л	1
=<<	Связывание	Monad m => (a -> m b) -> m a -> m b	П	1
$	Стр. композиция	(a -> b) -> a -> b	П	0
$!	Строгость	(a -> b) -> a -> b	П	0
seq	Строгость	a -> b -> b	П	0

Глава 7.

Пакет модулей `Control`

Пакет модулей `Control` содержит модули, применяющиеся для программирования различных управляющих сущностей, а также содержащие вспомогательные определения, вроде монад и функторов. Модули этого пакета расширяют стандартные определения монады и функтора, предоставляют программисту массу дополнительных возможностей для использования этих идиом в своей практике.

7.1. Модуль `Applicative`

Этот модуль описывает структуру, промежуточную между монадой и функтором. Эта структура предоставляет выстраивать чистые выражения в последовательности, но не предоставляет возможности связывания. Технически эта структура является строго-нестрогим моноидальным функтором — прикладным функтором. Детально о такой структуре описывается в [15]. Главный класс этого модуля очень полезен вместе с экземплярами класса `Traversable` (см. раздел 8.28.).

Использование:

```
import Control.Applicative
```

Главный класс модуля, описывающий прикладной функтор:

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Этот класс описывает функторы, для которых имеется возможность осуществления последовательных действий. Он содержит два метода: `pure` — используется для «втягивания» некоторого значения в функтор, а также `(<*>)` — последовательное применение функтора.

Каждый экземпляр класса `Applicative` должен удовлетворять следующим правилам:

1) *Тождество*:

```
pure id <*> v = v
```

2) *Композиция*:

```
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```

3) *Гомоморфизм*:

```
pure f <*> pure x = pure (f x)
```

4) *Взаимозаменяемость*:

```
u <*> pure y = pure ($ y) <*> u
```

Кроме того, экземпляры класса `Functor` должны дополнительно удовлетворять правилу:

```
fmap f x = pure f <*> x
```

Если тип `f` является монадой, то можно определить `pure = return` и `(<*>) = ap`.

Экземплярами класса `Applicative` являются следующие типы: `IO`, `Id`, `Maybe`, `ZipList`, `[]`, `Const`, `WrappedMonad`, `(,)`, `(->)` и `WrappedArrow`. все эти экземпляры описаны в рассматриваемом модуле.

Также в этом модуле описан дополнительный класс `Alternatives`, который определяет интерфейс к моноиду над функторами, для которых имеется возможность их выстраивания в последовательности действий. Описание этого класса выглядит следующим образом:

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

Метод empty является тождеством для операции (<|>), которая, в свою очередь, является ассоциативной бинарной операцией.

Экземплярами этого класса являются следующие типы: Maybe, [], WrappedMonad и WrappedArrow. Все эти экземпляры описаны в рассматриваемом модуле.

Дополнительно в данном модуле описывается набор утилитарных функций, которые предоставляют вспомогательные возможности для работы со значениями типов, являющихся экземплярами классов Applicative и Alternative. Ниже перечисляются все такие функции.

Функция: (<$>)

Описание: синоним функции fmap для исключения коллизии.

Определение:

```
infixl 4 <$>
(<$>) :: Functor f => (a -> b) -> f a -> f b
f <$> a = fmap f a
```

Функция: (<$)

Описание: функция для замены значения.

Определение:

```
infixl 4 <$
(<$) :: Functor f => a -> f b -> f a
(<$) = (<$>) . const
```

Функция: (*>)

Описание: выполнение последовательных действий с «уничтожением» первого аргумента.

Определение:

```
infixl 4 *>
(*>) :: Applicative f => f a -> f b -> f b
(*>) = liftA2 (const id)
```

Функция: (<*)

Описание: выполнение последовательных действий с «уничтожением» второго аргумента.

Определение:

```
infixl 4 <*
(<*) :: Applicative f => f a -> f b -> f a
(<*) = liftA2 const
```

Функция: (<**>)

Описание: вариант метода (<*>) с зарезервированными аргументами.

Определение:

```
infixl 4 <**>
(<**>) :: Applicative f => f a -> f (a -> b) -> f b
(<**>) = liftA2 (flip ($))
```

Функция: liftA

Описание: втягивает функцию в действие. Эта функция может использоваться в качестве входного аргумента для метода fmap в экземплярах класса Functor.

Определение:

```
liftA :: Applicative f => (a -> b) -> f a -> f b
liftA f a = pure f <*> a
```

Функция: liftA2

Описание: втягивает в действие функцию с двумя аргументами. Подобна функции liftA.

Определение:

```
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

Функция: liftA3

Описание: втягивает в действие функцию с тремя аргументами. Подобна функции liftA.

Определение:

```
liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
liftA3 f a b c = f <$> a <*> b <*> c
```

Функция: optional
Описание: функция для выбора альтернативы: 0 или 1.
Определение:

```
optional :: Alternative f => f a -> f (Maybe a)
optional v = Just <$> v <|> pure Nothing
```

Функция: some
Описание: функция для выбора альтернативы: 1 или более.
Определение:

```
some :: Alternative f => f a -> f [a]
some v = some_v
  where many_v = some_v <|> pure []
    some_v = (:) <$> v <*> many_v
```

Функция: many
Описание: функция для выбора альтернативы: 0 или более.
Определение:

```
many :: Alternative f => f a -> f [a]
many v = many_v
  where many_v = some_v <|> pure []
    some_v = (:) <$> v <*> many_v
```

Последние три функции используются в синтаксических анализаторах, для реализации которых впервые и был сделан этот модуль.

Остаётся отметить, что за этот модуль в поставке языка Haskell отвечает P. Патерсон, с которым можно связаться по адресу электронной почты ross@soi.city.ac.uk.

7.2. Модуль Arrow

Модуль **Arrow** содержит базовое определение стрелки, основанное на теоретической работе [9], а также отчасти на работе [21]. В этих статьях описаны правила, которым должны удовлетворять описываемые в модуле комбинаторы. Дополнительные материалы по стрелкам могут быть найдены по адресу в интернете http://www.haskell.org/arrows/.

Использование:

```
import Control.Arrow
```

Главный класс модуля, который описывает интерфейс стрелки, выглядит следующим образом:

```
class Arrow a where
  arr    :: (b -> c) -> a b c
  pure   :: (b -> c) -> a b c
  (>>>)  :: a b c -> a c d -> a b d
  first  :: a b c -> a (b, d) (c, d)
  second :: a b c -> a (d, b) (d, c)
  (***)  :: a b c -> a b' c' -> a (b, b') (c, c')
  (&&&)  :: a b c -> a b c' -> a b (c, c')
```

Это основной класс для описания стрелок. Любой его экземпляр должен в обязательном порядке определять метод `arr` или метод `pure`, которые являются синонимами. Другие комбинаторы класса выражены через базовый метод, но могут быть переопределены для достижения требуемых эффектов.

Метод `arr` втягивает функцию в стрелку. Программист обязательно должен определить либо этот метод, либо метод `pure` при разработке экземпляров класса `Arrow`. Метод `pure` является всего лишь синонимом для большего соответствия теории.

Операция (`>>>`) определяет композицию стрелок слева направо.

Метод `first` передаёт первый компонент входа через аргумент стрелки, после чего без изменений копирует остаток в выход. Метод `second` является зеркальной копией метода `first` для второго компонента входа.

Операция (`***`) разбивает вход для двух аргументов стрелки и комбинирует их выходы. Необходимо отметить, что в общем виде этот метод не является функтором. В целях оптимизации этот метод нужно переопределять для конкретных типов. Также и операция (`&&&`) предназначена для передачи входа в оба аргумента стрелки и последующего комбинирования выходов. Эта операция тоже должна быть переопределена для конкретных типов, хотя для неё и имеется реализация по умолчанию.

Экземплярами класса `Arrow` являются типы: (`->`) и `Kleisli`, причём тип `Kleisli` определён в этом модуле как

```
newtype Kleisli m a b
  = Kleisli
    {
      runKleisli :: (a -> m b)
    }
```

Этот изоморфный тип определяет стрелку Клейсли для монад.

Для класса `Arrow` определён ряд утилитарных функций. К числу таких функций относятся следующие.

Функция: `returnA`

Описание: стрелка тождества, которая играет роль метода `return` для стрелок.

Определение:

```
returnA :: Arrow a => a b b
returnA = arr id
```

Функция: (^>>)

Описание: прекомпозиция с чистой функцией.

Определение:

```
infixr 1 ^>>
(^>>) :: Arrow a => (b -> c) -> a c d -> a b d
f ^>> a = arr f >>> a
```

Функция: (>>^)

Описание: посткомпозиция с чистой функцией.

Определение:

```
infixr 1 >>^
(>>^) :: Arrow a => a b c -> (c -> d) -> a b d
a >>^ f = a >>> arr f
```

Функция: (<<<)

Описание: композиция стрелок справа налево.

Определение:

```
infixr 1 <<<
(<<<) :: Arrow a => a c d -> a b c -> a b d
f <<< g = g >>> f
```

Функция: (<<^)
Описание: прекомпозиция с чистой функцией (справа налево).
Определение:

```
infixr 1 <<^
(<<^) :: Arrow a => a c d -> (b -> c) -> a b d
a <<^ f = a <<< arr f
```

Функция: (^<<)
Описание: посткомпозиция с чистой функцией (справа налево).
Определение:

```
infixr 1 ^<<
(^<<) :: Arrow a => (c -> d) -> a b c -> a b d
f ^<< a = arr f <<< a
```

Дополнительно в модуле Arrow описаны вспомогательные классы, которые конкретизируют области применения стрелок. К таковым классам относятся следующие.
Класс: ArrowZero
Описание: определяет операцию получения нулевого элемента для моноидов.
Определение:

```
class Arrow a => ArrowZero a where
  zeroArrow :: a b c
```

Экземпляры: Kleisli.
Класс: ArrowPlus
Описание: определяет операцию сложения для моноидов.
Определение:

```
class ArrowZero a => ArrowPlus a where
  (<+>) :: a b c -> a b c -> a b c
```

Экземпляры: Kleisli.
Класс: ArrowChoice
Описание: определяет функции для осуществления ветвления алгоритма в тех стрелках, которые поддерживают ветвление. Имеет методы, которые соответствуют ключевым словам **if** и **case**. Любой экземпляр этого класса в обязательном порядке должен определять метод left.

Определение:

```
class Arrow a => ArrowChoice a where
  left  :: a b c -> a (Either b d) (Either c d)
  right :: a b c -> a (Either d b) (Either d c)
  (+++) :: a b c -> a b' c' -> a (Either b b') (Either c c')
  (|||) :: a b d -> a c d -> a (Either b c) d
```

Метод `left` «скармливает» помеченные входы аргументу стрелки, передавая остаток без изменений в выход. Метод `right` является зеркальным отражением метода `left`. Оба этих метода могут быть переопределены в экземплярах для более эффективной работы с конкретными типами данных.

Операция (+++) разделяет вход между двумя аргументами стрелки, перепомечает их и сливает выходы. Необходимо отметить, что в общем случае этот метод не является функтором. С другой стороны операция (|||) просто разделяет вход между двумя аргументами стрелки, после чего объединяет выходы.

Экземплярами класса являются типы: (->) и `Kleisli`.

Класс: `ArrowApply`

Описание: интерфейс для стрелок, у которых имеется операция приложения.

Определение:

```
class Arrow a => ArrowApply a where
  app :: a (a b c, b) c
```

Метод `app` предназначен, соответственно, для приложения стрелок.

Экземплярами класса являются типы: (->) и `Kleisli`.

Класс: `ArrowLoop`

Описание: интерфейс для стрелок, поддерживающих циклы. Единственный метод `loop` выражает вычисления, в которых выходное значение передаётся снова на вход, даже если вычисления производятся один раз.

Определение:

```
class Arrow a => ArrowLoop a where
  loop :: a (b, d) (c, d) -> a b c
```

Экземплярами класса являются типы: (->) и `Kleisli`.

Остаётся отметить, что за этот модуль в поставке языка Haskell отвечает Р. Патерсон, с которым можно связаться по адресу электронной почты `ross@soi.city.ac.uk`.

7.3. Модуль Concurrent

Модуль Concurrent предлагает разработчику программного обеспечения обобщённый интерфейс для работы с несколькими потоками управления в программах. Однако необходимо отметить, что многопоточность в языке Haskell не входит в стандарт, а определяется расширениями.

Использование:

```
import Control.Concurrent
```

Данный модуль автоматически включает в себя импорт подчинённых модулей Chan, MVar, QSem, QSemN и SampleVar, которые описываются ниже в этом разделе.

Многопоточность в языке Haskell является «лёгкой», это означает, что она не требует использования специализированных пакетов в операционных системах при работе с созданием потоков и с переключением контекстов. Работа с потоками обеспечивается внутренними средствами системы реального времени языка Haskell. Однако, если для целей программы при взаимодействии с модулями, написанными на других языках программирования, необходимо использовать средства операционных систем, можно пользоваться функцией forkOS вместо функции forkIO (описание ниже).

Потоки в языке Haskell могут взаимодействовать друг с другом при помощи абстракции MVar, которые описываются ниже в подразделе 7.3.2..

Основным типом данных, который используется для работы с потоками, является алгебраический тип данных ThreadId. Этот тип является абстрактным, а потому определяется следующим образом (на самом деле он обрабатывается внутри трансляторов языка Haskell особым образом):

```
type ThreadId = ()
```

Этот тип представляет идентификатор потока. Для него определены экземпляры классов Data, Eq, Ord, Show и Typeable. Причём экземпляр класса Ord используется для организации произвольного порядка над потоками. Соответственно, экземпляр класса Show используется для вывода информации о потоках, что может быть использовано в целях отладки многопоточных программ.

Для работы с потоками в рассматриваемом модуле определены следующие функции.

Функция: myThreadId

Описание: возвращает значение типа ThreadId для потока, из которого вызвана эта функция. Определена только в поставке компилятора GHC.

Определение:

```
myThreadId :: IO ThreadId
```

Функция: forkIO

Описание: отделяет от основного потока вычислений ещё один, в котором производятся вычисления в рамках монады IO, которая передана первым аргументом. Соответственно, функция возвращает идентификатор созданного потока. Данная функция создаёт «легковесный» поток без использования возможностей операционных систем. Для создания потока с использованием таких возможностей необходимо пользоваться функцией forkOS.

Определение:

```
forkIO :: IO () -> IO ThreadId
```

Функция определена в виде примитива.

Функция: killThread

Описание: останавливает заданный поток, идентификатор которого передаётся в качестве первого аргумента. Любая работа, которая проводилась останавливаемым потоком, не теряется, поскольку вычисления приостанавливаются до тех пор, пока они требуются в других потоках. Память, которая используется останавливаемым потоком, очищается сборщиком мусора только тогда, когда на неё нет больше ссылок извне. Функция доступна только в поставке компилятора GHC.

Определение:

```
killThread :: ThreadId -> IO ()
killThread tid = throwTo tid (AsyncException ThreadKilled)
```

Функция: throwTo

Описание: вызывает заданное исключение в заданном потоке. Эта функция не возвращает результата, пока в заданном потоке не произошло исключение, поэтому вызвавшему её потоку гарантируется, что исключение возникло. Это важное свойство в случаях конкурирующих потоков. Если два потока могут потребовать остановки друг друга, этот механизм гарантирует, что только один по-

ток сможет остановить другой, коллизии не произойдёт. При этом если целевой поток на момент вызова функции занят какими-то внешними вызовами, исключения не произойдёт, функция не вернёт результата, а поэтому внешний вызов целевого потока завершится. Функция доступна только в поставке компилятора GHC.

Определение:

```
throwTo :: ThreadId -> Exception -> IO ()
```

Функция определена в виде примитива.

Функция: `yield`

Описание: позволяет переключить контекст между текущими потоками. Может использоваться при определении многопоточных абстракций.

Определение:

```
yield :: IO ()
```

Функция определена в виде примитива.

Функция: `threadDelay`

Описание: останавливает текущий поток на заданное количество микросекунд. Нет гарантий того, что поток продолжит свою работу ровно по истечению заданного интервала времени (в зависимости от планирования работы), но гарантируется то, что он не начнёт свою работу раньше. Функция доступна только в поставке компилятора GHC.

Определение:

```
threadDelay :: Int -> IO ()
```

Функция определена в виде примитива.

Функция: `threadWaitRead`

Описание: блокирует текущий поток до тех пор, пока данные можно будет прочитать из заданного файла. Функция доступна только в поставке компилятора GHC.

Определение:

```
threadWaitRead :: Fd -> IO ()
```

Функция определена в виде примитива.

Функция: `threadWaitWrite`
Описание: блокирует текущий поток до тех пор, пока данные можно будет записать в заданный файл. Функция доступна только в поставке компилятора GHC.
Определение:

```
threadWaitWrite :: Fd -> IO ()
```

Функция определена в виде примитива.

Функция: `mergeIO`
Описание: создаёт потоки для обработки двух списков, которые начинают параллельно обрабатываться для последующего слияния результатов в одном списке. Функция доступна только в поставке компилятора GHC.
Определение:

```
mergeIO :: [a] -> [a] -> IO [a]
```

Функция определена в виде примитива.

Функция: `nmergeIO`
Описание: создаёт несколько потоков для обработки нескольких списков, которые начинают параллельно обрабатываться для последующего слияния результатов в одном списке. Функция доступна только в поставке компилятора GHC.
Определение:

```
nmergeIO :: [[a]] -> IO [a]
```

Функция определена в виде примитива.

Язык Haskell позволяет использовать средства операционных систем для работы с потоками. Не все трансляторы языка Haskell поддерживают этот механизм. На текущий момент единственным транслятором, который поддерживает потоки операционных систем, является компилятор GHC. Все последующие функции реализованы только в его поставке.

Функция: `rtsSupportsBoundThreads`
Описание: возвращает `True`, если текущий компилятор языка Haskell поддерживает потоки, основанные на механизмах операционных систем. Если эта функция возвратила `False`, то обе функции `forkOS` и `runInBoundThread` вызовут исключение.

Определение:

```
rtsSupportsBoundThreads :: Bool
```

Функция определена в виде примитива.

Функция: `forkOS`

Описание: как и функция `forkIO`, отщепляет новый поток, но использует при этом средства операционной системы. Из-за этого созданный поток становится ограничен свойствами текущей операционной системы.

Определение:

```
forkOS :: IO () -> IO ThreadId
```

Функция определена в виде примитива.

Функция: `isCurrentThreadBound`

Описание: возвращает `True` в случае, если текущий поток основан на средствах операционной системы.

Определение:

```
isCurrentThreadBound :: IO Bool
```

Функция определена в виде примитива.

Функция: `runInBoundThread`

Описание: запускает вычисления в монаде `IO`, переданные первым аргументом. Если текущий поток не основан на средствах операционной системы, создаётся временный поток, который основан на таких средствах.

Определение:

```
runInBoundThread :: IO a -> IO a
```

Функция определена в виде примитива.

Функция: `runInUnboundThread`

Описание: запускает вычисления в монаде `IO`, переданные первым аргументом. Если текущий поток основан на средствах операционной системы, создаётся временный поток, который не основан на таких средствах.

Определение:

```
runInUnboundThread :: IO a -> IO a
```

Функция определена в виде примитива.

Дополнительную информацию об этом модуле и о технике многопоточного программирования на языке Haskell можно получить из документации к модулю.

7.3.1. Модуль Chan

Модуль Chan содержит описания программных сущностей, которые позволяют работать с FIFO-каналами, не связанными с конкретной операционной системой. Данный модуль является «подчинённым» по отношению к модулю Concurrent, а потому его импорт должен выглядеть следующим образом:

```
import Control.Concurrent.Chan
```

Кроме того, сам модуль Concurrent уже включает в себя импорт модуля Chan, реимпортирует его определения, а потому в случае наличия в секции импорта модуля Concurrent можно использовать все программные сущности рассматриваемого модуля.

Главным типом данных, который используется для создания и работы с каналами, является алгебраический тип данных Chan, который к тому же является абстрактным и определённым внутри транслятора. Этот тип имеет только один экземпляр класса Typeable1. Однако наиболее интересным представляется набор функций, определённых в этом модуле.

Функция: newChan
Описание: создаёт и возвращает новый канал.
Определение:

```
newChan :: IO (Chan a)
```

Функция определена в виде примитива.

Функция: writeChan
Описание: записывает некоторое значение в заданный канал.
Определение:

```
writeChan :: Chan a -> a -> IO ()
```

Функция определена в виде примитива.

Функция: readChan
Описание: считывает и возвращает очередное значение из канала.

Определение:

```
readChan :: Chan a -> IO a
```

Функция определена в виде примитива.

Функция: dupChan

Описание: создаёт дубликат канала. Такой дубликат первоначально является пустым, но если какое-то значение записывается в какой-либо из каналов (исходный или дубликат), это значение становится доступным в обоих каналах. Так что эта функция создаёт некоторое подобие широковещательного канала, когда данные, записываемые кем-либо, становятся доступными любому.

Определение:

```
dupChan :: Chan a -> IO (Chan a)
```

Функция определена в виде примитива.

Функция: unGetChan

Описание: записывает некоторое значение назад в канал так, что оно становится следующим, которое можно считать из канала.

Определение:

```
unGetChan :: Chan a -> a -> IO ()
```

Функция определена в виде примитива.

Функция: isEmptyChan

Описание: возвращает значение `True`, если заданный канал пуст.

Определение:

```
isEmptyChan :: Chan a -> IO Bool
```

Функция определена в виде примитива.

Функция: getChanContents

Описание: возвращает ленивый список всех значений, содержащихся в заданном канале. Является аналогом функции `hGetContents` из модуля `IO` (см. стр. 494).

Определение:

```
getChanContents :: Chan a -> IO [a]
```

Функция определена в виде примитива.

Функция: writeList2Chan
Описание: записывает заданный список в канал.
Определение:

```
writeList2Chan :: Chan a -> [a] -> IO ()
```

Функция определена в виде примитива.

7.3.2. Модуль MVar

Модуль MVar содержит определения программных сущностей, предназначенных для работы с переменными синхронизации, которые требуются для синхронизации управления между несколькими потоками. Данный модуль является «подчинённым» по отношению к модулю Concurrent, поэтому его импорт выглядит следующим образом:

```
import Control.Concurrent.MVar
```

Кроме того, если модуль Concurrent уже подключён, импортировать модуль MVar нет необходимости.

Главный алгебраический тип данных, который определяет понятие переменной синхронизации, определён в рассматриваемом модуле под идентификатором MVar. Данный тип является абстрактным и определён внутри транслятора языка Haskell. Для него также определены экземпляры классов Typeable1, Data и Eq.

Каждая переменная синхронизации может быть пустой или наполненной некоторой информацией, необходимой для синхронизации управления между потоками. Все описываемые ниже прикладные функции работают с такими переменными.

Функция: newEmptyMVar
Описание: создаёт пустую переменную синхронизации.
Определение:

```
newEmptyMVar :: IO (MVar a)
```

Функция определена в виде примитива.

Функция: newMVar
Описание: создаёт переменную синхронизации, содержащую внутри себя заданное значение.

Определение:

```
newMVar :: a -> IO (MVar a)
```

Функция определена в виде примитива.

Функция: `takeMVar`
Описание: возвращает значение из заданной переменной синхронизации. Если заданная переменная синхронизации пуста, то функция ожидает, пока она не получит значение. После возврата функции заданная переменная синхронизации становится пустой. Если исполнение программного кода в нескольких потоках заблокировано этой функцией при помощи одной и той же переменной синхронизации, то при получении значения в эту переменную деблокируется только один поток (поскольку значение из переменной синхронизации вынимается). Остальные потоки ждут своей очереди в порядке блокировки (то есть деблокировка происходит в порядке FIFO). Это важное свойство функции можно использовать для управления несколькими потоками.
Определение:

```
takeMVar :: MVar a -> IO a
```

Функция определена в виде примитива.

Функция: `putMVar`
Описание: записывает заданное значение в переменную синхронизации. Если заданная переменная синхронизации уже содержит значение, функция `putMVar` будет ждать, пока переменная не станет пустой. Если исполнение программного кода в нескольких потоках заблокировано этой функцией при помощи одной и той же переменной синхронизации, то при записи значения в эту переменную деблокируется только один поток (остальные потоки ждут своей очереди — используется принцип FIFO). Это важное свойство функции можно использовать для управления несколькими потоками.
Определение:

```
putMVar :: MVar a -> a -> IO ()
```

Функция определена в виде примитива.

Функция: readMVar

Описание: эта функция является комбинацией функций takeMVar и putMVar. Она берёт значение из заданной переменной синхронизации, кладёт его обратно, но при этом возвращает полученное значение.

Определение:

```
readMVar :: MVar a -> IO a
readMVar m = block $ do a <- takeMVar m
                        putMVar m a
                        return a
```

Функция: swapMVar

Описание: заменяет значение в переменной синхронизации заданным значением.

Определение:

```
swapMVar :: MVar a -> a -> IO a
swapMVar mvar new = block $ do old <- takeMVar mvar
                               putMVar mvar new
                               return old
```

Функция: tryTakeMVar

Описание: неблокирующий вариант функции takeMVar. Эта функция немедленно пытается возвратить значение из заданной переменной синхронизации. Если переменная пуста, то функция не ожидает, когда она будет полной, а сразу возвращает значение Nothing. Если переменная синхронизации содержит некоторое значение, то это значение возвращается внутри конструктора Just, а сама переменная синхронизации становится пустой.

Определение:

```
tryTakeMVar :: MVar a -> IO (Maybe a)
```

Функция определена в виде примитива.

Функция: tryPutMVar

Описание: неблокирующий вариант функции putMVar. Эта функция немедленно пытается записать значение в заданную переменную синхронизации. Если переменная уже содержит некоторое значение, функция возвращает False, а переменная синхронизации нового значения не получает. Если же переменная синхронизации пуста, она получает новое значение, а рассматриваемая функция возвращает True.

Определение:

```
tryPutMVar :: MVar a -> a -> IO Bool
```

Функция определена в виде примитива.

Функция: isEmptyMVar

Описание: проверяет, пуста ли заданная переменная синхронизации. Если пуста, возвращает значение True, если нет — False соответственно. Необходимо отметить, что эта функция возвращает мгновенное отображение состояния переменной синхронизации, которое может измениться в следующий же момент. Необходимо с осторожностью пользоваться этой функцией, а для гарантированного результата нужно использовать функцию tryTakeMVar.

Определение:

```
isEmptyMVar :: MVar a -> IO Bool
```

Функция определена в виде примитива.

Функция: withMVar

Описание: безопасная обёртка для работы с содержимым переменных синхронизации. Эта функция безопасна с точки зрения отлова исключений. Она запишет значение в заданную переменную синхронизации даже в случае, если в процессе работы возникнет исключение. См. также описание модуля Exception в разделе 7.4..

Определение:

```
withMVar :: MVar a -> (a -> IO b) -> IO b
withMVar m io = block $ do a <- takeMVar m
                           b <- Exception.catch (unblock (io a))
                                                (\e -> do putMVar m a
                                                          throw e)
                           putMVar m a
                           return b
```

Функция: modifyMVar-

Описание: безопасная обёртка для замены значения в переменной синхронизации. Эта функция безопасна с точки зрения отлова исключений. Она заменит содержимое заданной переменной синхронизации даже в случае, если в процессе работы возникнет исключение. Её работа подобна работе функции withMVar.

Определение:

```
modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()
modifyMVar_ m io = block $ do a  <- takeMVar m
                              a' <- Exception.catch (unblock (io a))
                                                   (\e -> do putMVar m a
                                                             throw e)
                              putMVar m a'
```

Функция: modifyMVar

Описание: небольшая вариация функции modifyMVar_, которая позволяет не только записать значение в переменную синхронизации, но и вернуть значение в качестве результата своей работы.

Определение:

```
modifyMVar :: MVar a -> (a -> IO (a, b)) -> IO b
modifyMVar m io = block $ do a       <- takeMVar m
                             (a', b) <- Exception.catch (unblock (io a))
                                                        (\e -> do putMVar m a
                                                                  throw e)
                             putMVar m a'
                             return b
```

Функция: addMVarFinalizer

Описание: добавляет финализатор в заданную переменную синхронизации. Функция определена только в поставке компилятора GHC. Описание финализаторов приведено в разделе 10.2. (модуль ForeignPtr) и подразделе 11.9.2. (модуль Weak).

Определение:

```
addMVarFinalizer :: MVar a -> IO () -> IO ()
```

Функция определена в виде примитива.

7.3.3. Модуль QSem

Модуль QSem содержит определения программных сущностей, предназначенных для работы с простыми семафорами, которые используются для синхронизации управления между несколькими потоками. Данный модуль является «подчинённым» по отношению к модулю Concurrent, поэтому его импорт выглядит следующим образом:

```
import Control.Concurrent.QSem
```

Кроме того, если модуль Concurrent уже подключён, импортировать модуль QSem нет необходимости.

Главный алгебраический тип данных, который определяет понятие простого семафора, определён в рассматриваемом модуле под идентификатором QSem. Данный тип является абстрактным и определён внутри транслятора языка Haskell. Для него также определён экземпляр класса Typeable.

Далее описываются все прикладные функции, которые работают с простыми семафорами.

Функция: newQSem

Описание: строит новый простой семафор.

Определение:

```
newQSem :: Int -> IO QSem
```

Функция определена в виде примитива.

Функция: waitQSem

Описание: ожидает, когда заданный простой семафор освобождается.

Определение:

```
waitQSem :: QSem -> IO ()
```

Функция определена в виде примитива.

Функция: signalQSem

Описание: сигнализирует о том, что заданный простой семафор свободен.

Определение:

```
signalQSem :: QSem -> IO ()
```

Функция определена в виде примитива.

7.3.4. Модуль QSemN

Модуль QSemN содержит определения программных сущностей, предназначенных для работы с семафорами, в которых потоки могут ожидать произвольное количество данных. Данный модуль является «подчинённым» по отношению к модулю Concurrent, поэтому его импорт выглядит следующим образом:

```
import Control.Concurrent.QSemN
```

Кроме того, если модуль Concurrent уже подключён, импортировать модуль QSemN нет необходимости.

Главный алгебраический тип данных, который определяет понятие простого семафора, определён в рассматриваемом модуле под идентификатором QSemN. Данный тип является абстрактным и определён внутри транслятора языка Haskell. Для него также определён экземпляр класса Typeable.

Далее описываются все прикладные функции, которые работают с семафорами.

Функция: newQSemN

Описание: строит новый семафор с заданным количеством единиц ожидания.

Определение:

```
newQSemN :: Int -> IO QSem
```

Функция определена в виде примитива.

Функция: waitQSemN

Описание: ожидает, когда заданный семафор освобождается до определённого количества единиц.

Определение:

```
waitQSem :: QSem -> Int -> IO ()
```

Функция определена в виде примитива.

Функция: signalQSemN

Описание: сигнализирует о том, что заданный простой семафор свободен.

Определение:

```
signalQSem :: QSem -> Int -> IO ()
```

Функция определена в виде примитива.

7.3.5. Модуль SampleVar

Модуль SampleVar представляет описание программных сущностей, которые также работают с переменными синхронизации, но процесс управления немного отличается от переменных, объявленных как MVar (см. подраздел 7.3.2.). Данный

модуль является «подчинённым» по отношению к модулю Concurrent, поэтому его импорт выглядит следующим образом:

```
import Control.Concurrent.SampleVar
```

Кроме того, если модуль Concurrent уже подключён, импортировать модуль SampleVar нет необходимости.

Тип переменных синхронизации рассматриваемого вида является синонимом, который определён следующим образом:

```
type SampleVar a = MVar (Int, MVar a)
```

Первое значение в паре обозначает состояние переменной синхронизации. Если оно равно 1, переменная содержит некоторое значение. Если оно равно 0, переменная синхронизации пуста. Значение, меньшее 0, показывает (по модулю) количество заблокированных данной переменной синхронизации потоков.

Определённые таким образом переменные синхронизации работают иначе, чем обыкновенные MVar. Чтение значения из них, а также запись значения в пустую переменную синхронизации абсолютно тождественны подобным операциям над переменными синхронизации типа MVar. Однако запись нового значения в несвободную переменную синхронизации перезаписывает старое значение в ней, и именно этим такие переменные синхронизации отличаются от простых MVar.

Функции для работы с такими переменными синхронизации следующие.

Функция: newEmptySampleVar
Описание: создаёт новую переменную синхронизации вида SampleVar.
Определение:

```
newEmptySampleVar :: IO (SampleVar a)
newEmptySampleVar = do v <- newEmptyMVar
                       newMVar (0, v)
```

Функция: newSampleVar
Описание: создаёт новую переменную синхронизации вида SampleVar с заданным начальным значением.

Определение:

```
newSampleVar :: a -> IO (SampleVar a)
newSampleVar a = do v <- newEmptyMVar
                    putMVar v a
                    newMVar (1, v)
```

Функция: emptySampleVar

Описание: если заданная переменная синхронизации содержит некоторое значение, эта функция освобождает её. В противном случае не делает никаких действий.

Определение:

```
emptySampleVar :: SampleVar a -> IO ()
emptySampleVar v = do (readers, var) <- takeMVar v
                    if readers > 0
                      then do takeMVar var
                              putMVar v (0, var)
                      else putMVar v (readers,var)
```

Функция: readSampleVar

Описание: ожидает, когда заданная переменная синхронизации получает некоторое значение, считывает его и возвращает.

Определение:

```
readSampleVar :: SampleVar a -> IO a
readSampleVar svar = do (readers, val) <- takeMVar svar
                        putMVar svar (readers - 1, val)
                        takeMVar val
```

Функция: writeSampleVar

Описание: записывает значение в заданную переменную синхронизации, перезаписывая старое значение в ней.

Определение:

```
writeSampleVar :: SampleVar a -> a -> IO ()
writeSampleVar svar v = do (readers, val) <- takeMVar svar
                        case readers of
                          1 -> swapMVar val v >>
                               putMVar svar (1,val)
                          _ -> putMVar val v >>
                               putMVar svar (min 1 (readers + 1), val)
```

Функция: `isEmptySampleVar`
Описание: возвращает значение `True`, если заданная переменная синхронизации пуста. Необходимо отметить, что данная функция полезна только в случае, если никакие иные потоки не могут изменять состояние заданной переменной синхронизации. В противном случае её состояние может быть изменено мгновенно после получения результата этой функцией.
Определение:

```
isEmptySampleVar :: SampleVar a -> IO Bool
isEmptySampleVar svar = do (readers, val) <- readMVar svar
                           return (readers == 0)
```

7.4. Модуль Exception

Модуль `Exception` содержит описания программных сущностей, предназначенных для работы с исключениями. Отчасти определения в этом модуле дублируют такие же определение из стандартного модуля `Prelude`. Импорт же модуля осуществляется следующим образом:

```
import Control.Exception
```

В данном модуле определены некоторые алгебраические типы данных для представления различных вариантов исключений, а также разнообразные прикладные функции для работы с этими типами данных.
Тип: `Exception`
Описание: главный тип для представления исключений. Содержит в себе набор конструкторов, каждый из которых отвечает за определённый тип исключений. Каждое исключение, которое генерируется операционной системой, имеет этот тип, в то время как некоторые типы исключений в свою очередь могут иметь подтипы, расшифровываемые при помощи специализированного типа данных.
Определение:

```
data Exception
  = ArithException ArithException | ArrayException ArrayException | AssertionFailed String
  | AsyncException AsyncException | BlockedOnDeadMVar             | BlockedIndefinitely
  | NestedAtomically              | Deadlock                      | DynException Dynamic
  | ErrorCall String              | ExitException ExitCode        | IOException IOException
  | NoMethodError String          | NonTermination                | PatternMatchFail String
  | RecConError String            | RecSelError String            | RecUpdError String
```

Конструктор ArithException отвечает за разнообразные исключения, которые могут возникнуть во время выполнения арифметических операций. Необходимо отметить, что компилятор GHC на текущий момент может обрабатывать только исключение типа DivideByZero (деление на ноль).

Конструктор ArrayException отвечает за все исключения, которые связаны с обработкой массивов. Необходимо отметить, что на текущий момент компилятор GHC не поддерживает этот тип исключений.

Исключение типа AssertionFailed бросается функцией assert (см. стр. 209) в случаях, когда условие ложно. Аргумент этого конструктора типа String может использоваться в качестве ссылки на место в программе, где находится вызов функции assert.

Конструктор AsyncException используется для исключений, которые могут возникнуть в процессе работы многопоточных приложений при взаимодействии потоков между собой. Данный тип исключений детально описывается ниже.

Исключение типа BlockedOnDeadMVar возбуждается тогда, когда поток управления был заблокирован при помощи функции takeMVar (см. стр. 189) для переменной синхронизации, у которой более нет ссылок (таким образом заблокированный поток никогда не будет деблокирован).

Исключение типа BlockedIndefinitely выкидывается в случае, когда текущий поток управления заблокирован с невозможностью снять блокировку, поскольку на атомарную транзакцию, при помощи которой он заблокирован, нет больше ссылок.

Исключение типа NestedAtomically происходит тогда, когда во время исполнения программы было обнаружено, что внутрь атомарной STM-транзакции была произведена попытка внести ещё одну транзакцию. Обычно это происходит во время использования функции unsafePeformIO (см. стр. 421) над атомарными транзакциями.

Исключение типа Deadlock используется тогда, когда обнаруживается, что в программе не существует исполняющихся потоков управления (программа «зависла»). Это исключение возбуждается только в главном потоке управления.

Исключение типа DynException является динамически типизируемым исключением (подробно описывается ниже).

Исключение типа ErrorCall возбуждается во время вызова функции error (см. стр. 133). Аргумент конструктора типа String является описанием ошибки, которое выводится на экран.

Исключение типа ExitException выкидывается в случае вызова функций exitWith (см. стр. 484) и exitFailure (см. стр. 485). Аргумент конструктора является значением, которое передано этим функциям в качестве входного параметра. Необработанное исключение этого типа в главном потоке управления программы приводит к тому, что программа завершается с заданным кодом выхода.

Конструктор IOException используется для обработки исключений, возникающих в процессе работы функций ввода/вывода. Все такие исключения генерируются в монаде IO. См. также описание модуля System.IO.Error (подраздел 11.7.1.).

Исключение типа NoMethodError возбуждается в случае, если производится попытка вызвать метод класса, который не определён для типа, у которого вызывается этот метод, а также нет определения такого метода, используемого по умолчанию. Компилятор GHC в этом случае также генерирует предупреждение об отсутствующем определении метода.

Исключение типа NonTermination используется тогда, когда текущий поток управления завис в бесконечном цикле. Это исключение может и не быть сгенерировано в случаях, когда программа не подразумевает окончания своего исполнения.

Исключение типа PatternMatchFail выкидывается в случае неуспешного процесса сопоставления с образцами. Аргумент конструктора типа String содержит описание исключения, которое включает наименование функции, в которой произошло исключение, а также имя файла и номер строки в списке определений.

Исключение типа RecConError генерируется тогда, когда производится попытка вычислить значение поля структуры, для которого не было задано начального значения при создании. Аргумент конструктора типа String описывает местоположение кода для создания поля в исходных кодах программы.

Исключение типа RecSelError генерируется тогда, когда производится попытка обратиться к полю структуры, которого вообще не существует у структуры. Такая ситуация может происходить в тех случаях, когда в структуре имеется несколько конструкторов, внутри которых определяются разные наборы полей.

Аргумент конструктора типа String описывает местоположение кода для создания поля в исходных кодах программы.

Исключение типа RecUpdError генерируется тогда, когда производится попытка записать новое значение в поле структуры, которого вообще не существует в структуре. Такая ситуация может происходить в тех случаях, когда в структуре имеется несколько конструкторов, внутри которых определяются разные наборы полей. Аргумент конструктора типа String описывает местоположение кода для создания поля в исходных кодах программы.

Алгебраический тип данных Exception имеет определённые экземпляры для следующих классов: Eq, Show и Typeable.

Тип: IOException

Описание: данный тип исключений используется в рамках монады IO для обеспечения работы системы ввода/вывода. Данный тип определён в виде примитива и содержит внутри себя более специфичный тип исключения, строку с описанием исключения и дескриптор файла или канала, который использовался, когда было возбуждено исключение.

Определение:

Тип определён в виде примитива.

Тип IOException имеет экземпляры следующих классов: Eq, Show и Typeable.

Тип: ArithException

Описание: исключения, которые могут возникнуть в процессе выполнения арифметических операций.

Определение:

```
data ArithException
  = Overflow
  | Underflow
  | LossOfPrecision
  | DivideByZero
  | Denormal
```

Наименование конструкторов даёт понимание того, для чего каждый тип арифметического исключения используется. Тип Overflow используется при переполнении во время арифметических операций. Тип Undeflow также используется при переполнении при использовании действительных чисел, когда точность вычислений выходит за разумные рамки. Исключение LossOfPrecision возбуждается также во время вычислений над действительными числами, когда проис-

ходит потеря точности. Само собой разумеется, что тип DivideByZero используется при делении на ноль. Наконец, исключение типа Denormal возбуждается при денормализации числа.

Тип ArithException имеет определённые экземпляры для следующих классов: Eq, Ord, Show и Typeable.

Тип: ArrayException

Описание: данный тип исключений генерируется во время работы с массивами.

Определение:

```
data ArrayException
  = IndexOutOfBounds String
  | UndefinedElement String
```

Опять же наименование конструкторов намекает на способ использования конкретного исключения. Тип IndexOutOfBounds возбуждается тогда, когда происходит выход за пределы массива. Тип UndefinedElement генерируется, в свою очередь, при отсутствии элемента массива во время попытки обращения к нему.

Тип ArrayException имеет определённые экземпляры для следующих классов: Eq, Ord, Show и Typeable.

Тип: AsyncException

Описание: тип исключений, которые могут произойти во время работы многопоточных приложений (в аспекте работы потоков друг с другом).

Определение:

```
data AsyncException
  = StackOverflow
  | HeapOverflow
  | ThreadKilled
```

Также наименования конструкторов этого типа намекают на способ использования. Тип StackOverflow генерируется при переполнении стека. Тип HeapOverflow возбуждается при переполнении кучи. Наконец, исключение типа ThreadKilled используется, когда останавливается поток управления.

Тип AsyncException имеет определённые экземпляры для следующих классов: Eq, Ord, Show и Typeable.

Список функций для работы с перечисленными алгебраическими типами данных следующий.

Функция: `throwIO`

Описание: вариант функции `throw` (см. ниже), который может быть использован в рамках монады `IO`. Этот вариант должен быть использован в монаде `IO`, если имеется нужда в гарантировании того, что исключение не затронет порядок исполнения операций ввода/вывода (функция `throw` не гарантирует этого).

Определение:

```
throwIO :: Exception -> IO a
```

Функция определена в виде примитива.

Функция: `throw`

Описание: генерирует заданное исключение. Исключения могут быть сгенерированы в чистом функциональном коде, но отлавливаться они должны только внутри монады `IO`.

Определение:

```
throw :: Exception -> a
```

Функция определена в виде примитива.

Функция: `ioError`

Описание: генерирует исключение типа `IOError` в рамках монады `IO`.

Определение:

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

Функция определена в виде примитива.

Функция: `throwTo`

Описание: возбуждает заданное исключение в требуемом потоке управления. Функция определена только в поставке компилятора GHC. Данная функция не возвращает результата и останавливает поток управления до тех пор, пока в целевом потоке не возбудится исключение. В этом случае в потоке-источник имеется уверенность в том, что целевой поток получил исключение. Это — очень важное свойство, поскольку позволяет двум потокам безопасно закрывать друг друга. Также эта функция не возвращает результата, пока в целевом потоке происходит вызов внешней функции.

Определение:

```
throwTo :: ThreadId -> Exception -> IO ()
```

Функция определена в виде примитива.

Функция: `catch`

Описание: основная функция для отлова исключений. Она получает действие, которое необходимо запустить (и в котором может возникнуть исключение), и запускает его. Если в процессе работы действия происходит исключение, для его обработки вызывается обработчик, переданный в функцию `catch` вторым аргументом. Результат его работы возвращается в качестве результата функции. Если же исключений не произошло, результатом работы функции будет результат исходного действия.

Определение:

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

Функция определена в виде примитива.

В качестве примера можно привести такой код:

```
catch (openFile f ReadMode)
      (\e -> hPutStr stderr ("Couldn't open " ++
                             f ++
                             ": " ++
                             show e))
```

Для отлова исключений в чистых функциях необходимо пользоваться функцией `evaluate` (см. стр. 205).

Надо отметить, что из-за того, что в языке Haskell не определён порядок выполнения вычислений (в силу нестрогости языка), невозможно сказать, в каком порядке и какие исключения будут сгенерированы и отловлены в случае, если существует несколько вариантов развития таких событий. Например, в выражении

```
error "Error!" + 1 `div` 0
```

неизвестно, какое исключение будет возбуждено первым — `ErrorCall` или `ArithException`. Функция `catch` выполняет недетерминированный отлов исключений, поэтому при повторном вызове того же самого выражения результат может быть иной. Но это не идёт в разрез с принципом детерминизма функций, поскольку результат находится внутри монады `IO`.

Необходимо отметить, что в стандартном модуле `Prelude` имеется одноимённая функция `catch`, которая работает с исключениями ввода/вывода.

При использовании модуля Exception желательно скрывать функцию из модуля Prelude, поскольку описываемая функция является более общей.

Функция: catchJust

Описание: вариант функции catch, который отличается от исходного варианта тем, что первым параметром принимает на вход предикат, который используется для выбора тех исключений, которые интересны именно в этом вызове обработчика. Более того, имеется набор предопределённых предикатов: ioErrors, arithExceptions и т. д. Любые исключения, которые не удовлетворяют предикат, пропускаются этой функцией и могут быть отловлены в более ранних вызовах функций catch или catchJust.

Определение:

```
catchJust :: (Exception -> Maybe b) -> IO a -> (b -> IO a) -> IO a
catchJust p a handler = catch a handler'
  where
    handler' e = case p e of
                   Nothing -> throw e
                   Just b  -> handler b
```

Функция: handle

Описание: версия функции catch, в которой аргументы поменяны местами. Может использоваться в случаях, когда код обработчика исключений короче, чем код действия, в котором могут произойти исключения.

Определение:

```
handle :: (Exception -> IO a) -> IO a -> IO a
handle =  flip catch
```

Функция: handleJust

Описание: такая же версия функции catchJust, как и вариант handle. Используется в тех же целях.

Определение:

```
handleJust :: (Exception -> Maybe b) -> (b -> IO a) -> IO a -> IO a
handleJust p =  flip (catchJust p)
```

Функция: try

Описание: ещё один вариант функции catch, который возвращает значение типа Either: конструктор Right используется в случае, если исключений не было,

а конструктор Left используется, как должно быть понятно, в случаях, если возникло исключение.
Определение:

```
try :: IO a -> IO (Either Exception a)
try a = catch (a >>= \v -> return (Right v))
              (\e -> return (Left e))
```

Необходимо отметить, что в стандартном модуле Prelude имеется одноимённая функция try, которая работает с исключениями ввода/вывода. При использовании модуля Exception желательно скрывать функцию из модуля Prelude, поскольку описываемая функция является более общей.

Функция: tryJust
Описание: такой же вариант функции catchJust, как и вариант try для функции catch. Используется для тех же самых целей.
Определение:

```
tryJust :: (Exception -> Maybe b) -> IO a -> IO (Either b a)
tryJust p a = do
  r <- try a
  case r of
    Right v -> return (Right v)
    Left  e -> case p e of
                 Nothing -> throw e
                 Just b  -> return (Left b)
```

Функция: evaluate
Описание: заставляет транслятор языка Haskell вычислить значение выражения, определённого первым аргументом (несмотря на возможные исключения). В качестве результата возвращает вычисленное значение, обёрнутое монадой IO.
Определение:

```
evaluate :: a -> IO a
```

Функция определена в виде примитива.

Функция: mapException
Описание: проецирует одно исключение на другое.

Определение:

```
mapException :: (Exception -> Exception) -> a -> a
mapException f v = unsafePerformIO (catch (evaluate v)
                                          (\x -> throw (f x)))
```

Функция: ioErrors

Описание: предопределённый предикат для использования в функциях catchJust, handleJust и tryJust. Определяет набор исключений, связанных с вводом/выводом.

Определение:

```
ioErrors :: Exception -> Maybe IOError
ioErrors (IOException e) = Just e
ioErrors _               = Nothing
```

Функция: arithExceptions

Описание: предопределённый предикат для использования в функциях catchJust, handleJust и tryJust. Определяет набор исключений, связанных с арифметическими операциями.

Определение:

```
arithExceptions :: Exception -> Maybe ArithException
arithExceptions (ArithException e) = Just e
arithExceptions _                  = Nothing
```

Функция: errorCalls

Описание: предопределённый предикат для использования в функциях catchJust, handleJust и tryJust. Определяет набор исключений, связанных с вызовом функции error.

Определение:

```
errorCalls :: Exception -> Maybe String
errorCalls (ErrorCall e) = Just e
errorCalls _             = Nothing
```

Функция: assertions

Описание: предопределённый предикат для использования в функциях catchJust, handleJust и tryJust. Определяет набор исключений, связанных с вызовом функции assert.

Определение:

```
assertions :: Exception -> Maybe String
assertions (AssertionFailed e) = Just e
assertions _                   = Nothing
```

Функция: dynExceptions

Описание: предопределённый предикат для использования в функциях `catchJust`, `handleJust` и `tryJust`. Определяет набор исключений, которые могут быть назначены динамически.

Определение:

```
dynExceptions :: Exception -> Maybe Dynamic
dynExceptions (DynException e) = Just e
dynExceptions _                = Nothing
```

Функция: asyncExceptions

Описание: предопределённый предикат для использования в функциях `catchJust`, `handleJust` и `tryJust`. Определяет набор исключений, связанных с многопоточностью и работой потоков друг с другом.

Определение:

```
asyncExceptions :: Exception -> Maybe AsyncException
asyncExceptions (AsyncException e) = Just e
asyncExceptions _                  = Nothing
```

Функция: userErrors

Описание: предопределённый предикат для использования в функциях `catchJust`, `handleJust` и `tryJust`. Определяет набор исключений, связанных с исключениями, которые определены разработчиком программного обеспечения самостоятельно.

Определение:

```
userErrors :: Exception -> Maybe String
userErrors (IOException e) | isUserError e = Just (ioeGetErrorString e)
userErrors _                               = Nothing
```

Функция: throwDyn

Описание: генерирует динамическое исключение, в качестве которого может быть «выкинуто» произвольное значение, для которого имеется экземпляр клас-

са Typeable. Эта функция определена для того, чтобы иметь возможность генерирования произвольных исключений при ограничении, которое накладывает тип Exception — он не может быть расширен.
Определение:

```
throwDyn :: Typeable exception => exception -> b
throwDyn exception = throw (DynException (toDyn exception))
```

Функция: throwDynTo
Описание: вариант функции throwDyn, который имеет те же свойства, что и функция throwTo (см. стр. 202). Исключение бросается в заданном потоке. Данная функция определена только в поставке компилятора GHC.
Определение:

```
throwDynTo :: Typeable exception => ThreadId -> exception -> IO ()
```

Функция определена в виде примитива.
Функция: catchDyn
Описание: отлавливает динамическое исключение заданного типа. Все остальные динамические исключения перегенерируются.
Определение:

```
catchDyn :: Typeable exception => IO a -> (exception -> IO a) -> IO a
catchDyn m k = catchException m handle
  where
    handle ex = case ex of
                (DynException dyn) -> case fromDynamic dyn of
                                        Just exception  -> k exception
                                        Nothing         -> throw ex
                _                  -> throw ex
```

При работе с динамическими исключениями рекомендуется определить для этих целей отдельный тип данных, чтобы не было коллизии с функциями из сторонних библиотек, в которых используется механизм динамических исключений.
Функция: block
Описание: эта функция позволяет заблокировать текущий поток таким образом, чтобы он не воспринимал исключений, генерируемых извне, до тех пор, пока текущий поток не будет деблокирован. Любое внешнее исключение не генерирует-

ся в заблокированном потоке, пока производятся действия, являющиеся первым аргументом функции. Понятно, что функция возвращает результат выполнения действий.

Определение:

```
block :: IO a -> IO a
```

Функция определена в виде примитива.

Функция: `unblock`

Описание: функция, обратная по действию функции `block`. Деблокирует текущий поток так, что он продолжает воспринимать исключения, генерируемые извне.

Определение:

```
unblock :: IO a -> IO a
```

Функция определена в виде примитива.

Функция: `assert`

Описание: аналог оператора **if**, который возвращает значение второго аргумента, если первый аргумент равен `True`. Если первый аргумент равен `FalsE`, генерируется исключение типа `AssertionFailed` с пустой строкой в качестве аргумента. Служебными средствами трансляторов языка Haskell вместо пустой строки подставляются наименование файла с исходными кодами и номер строки. Эта функция используется для отладки.

Определение:

```
assert :: Bool -> a -> a
assert True  x = x
assert False _ = throw (AssertionFailed "")
```

Функция: `bracket`

Описание: если имеется необходимость зарезервировать некоторый ресурс (файл или канал), проделать с ним некоторые действия, а после них разблокировать ресурс для других возможных обработчиков, необходимо пользоваться этой функцией. Она в обязательном порядке произведёт необходимые действия даже в случае, если в процессе работы возникло исключение. Если исключение возникло, функция `bracket` выполняет завершающие действия, после чего регенерирует выброшенное исключение.

Определение:

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket before after thing = block (do a <- before
                                       r <- catch
                                              (unblock (thing a))
                                              (\e -> do after a
                                                        throw e)
                                       after a
                                       return r)
```

Типовым примером использования этой функции является:

```
bracket
   (openFile "filename" ReadMode)
   (hClose)
   (\handle -> do { ... })
```

Функция: `bracket-`

Описание: вариант функции `bracket`, который можно использовать в тех случаях, когда нет необходимости в значении первого (начального) действия.

Определение:

```
bracket_ :: IO a -> IO b -> IO c -> IO c
bracket_ before after thing = bracket before (const after) (const thing)
```

Функция: `bracketOnError`

Описание: подобна функции `bracket`, но просто выполняет заключительные действия, не возвращая результата обработки исключения.

Определение:

```
bracketOnError :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracketOnError before after thing = block (do a <- before
                                              catch
                                                (unblock (thing a))
                                                (\e -> do after a
                                                          throw e))
```

Функция: `finally`

Описание: специальный вариант функции `bracket`, который просто выполняет заключительные действия.

Определение:

```
finally :: IO a -> IO b -> IO a
a `finally` sequel = block (do r <- catch
                                (unblock a)
                                (\e -> do sequel
                                          throw e)
                               sequel
                               return r)
```

7.5. Модуль Monad

Модуль Monad содержит дополнительные расширения функциональности языка Haskell, которая используется для работы с монадами. Использование модуля:

```
import Control.Monad
```

Данный модуль определяет три класса, которые описывают интерфейсы монадических типов.

Класс: Functor
Описание: интерфейс для типов, которые могут проецироваться друг на друга.
Определение:

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

Экземпляры класса Functor должны удовлетворять следующим законам:

```
fmap id      == id

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

Для этого класса определены экземпляры для следующих типов: Digit, Elem, FingerTree, IO, Id, IntMap, Maybe, Node, Queue, ReadP, ReadPrec, STM, Seq, Tree, ViewL, ViewR, ZipList, [], Array i, Const m, Either a, Map k, ST s, WrappedMonad m, (,) a, (->) r, WrappedArrow a b.

Класс: Monad

Описание: интерфейс для определения базовых операций над монадами, понятием из теории категорий. С точки зрения программиста на языке Haskell о монадах проще всего думать как об абстрактном типе данных, представляющим последовательность действий.

Определение:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
```

Для каждого экземпляра класса Monad достаточно объявить методы (>>=) и return, хотя другие методы также можно определить в целях оптимизации. Также любой экземпляр этого класса должен удовлетворять следующим законам:

```
return a >>= k           ==  k a

m >>= return             ==  m

m >>= (\x -> k x >>= h)  ==  (m >>= k) >>= h
```

Если тип одновременно является экземпляром классов Functor и Monad, то он дополнительно должен удовлетворять следующему правилу:

```
fmap f xs == xs >>= return . f
```

Операция (>>=) последовательно выполняет два действия, передавая во второе результат выполнения первого действия. Это операция связывания двух действий в последовательность. Операция (>>) является облегчённой версией операции (>>=), которая используется тогда, когда результат выполнения первого действия не требуется во втором действии.

Метод return втягивает некоторое значение в монаду. Метод fail останавливает процесс последовательного выполнения действий с некоторым сообщением о причине остановки. Последний метод не является частью математического определения монады и введён в класс Monad для того чтобы корректно обрабатывать образцы в выражениях **do**.

Экземпляром этого класса являются следующие типы данных: IO, Maybe, P, ReadP, ReadPrec, STM, Seq, [], ArrowMonad a, ST s, (->) r.

Класс: MonadPlus

Описание: интерфейс для монад, у которых есть единица и которые поддерживают ассоциативную операцию (полугруппы).

Определение:

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

Любой экземпляр этого класса должен удовлетворять следующим законам:

```
mzero >>= f == mzero
v >> mzero  == mzero
```

Константный метод mzero возвращает единицу ассоциативной операции mplus полугруппы.

Экземпляром этого класса являются следующие типы данных: Maybe, P, ReadP, ReadPrec, Sec, [].

Утилитарные функции в этом модуле подчиняются некоторым соглашениям о наименовании. Постфикс M обозначает, что функция всегда работает с категориями Клейсли. В таких функциях конструктор типа m всегда используется только в типах, возвращаемых функциями. Например:

```
filter  ::              (a ->   Bool) -> [a] ->   [a]
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

Постфикс (_) используется в функциях, которые используются только тогда, когда результат выполнения не важен, а важны только побочные эффекты, предоставляемые монадой. Это значит, что такие функции меняют тип (m a) на (m ()). Например:

```
sequence  :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
```

Префикс m обобщает обычную функцию в функцию, работающую с монадами. Так, к примеру:

```
sum  :: Num a       => [a]   -> a
msum :: MonadPlus m => [m a] -> m a
```

Для работы с перечисленными классами определены следующие утилитарные функции.

Функция: mapM

Описание: расширение функции map для работы с категориями Клейсли.

Определение:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f = sequence . map f
```

Функция: mapM-

Описание: вариант функции mapM, используемый тогда, когда результат функции не важен, но важны побочные эффекты монады.

Определение:

```
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f = sequence_ . map f
```

Функция: forM

Описание: вариант функции mapM, у которого входные аргументы следуют в обратном порядке. Используется в эстетических целях.

Определение:

```
forM :: Monad m => [a] -> (a -> m b) -> m [b]
forM = flip mapM
```

Функция: forM-

Описание: вариант функции mapM_, у которого входные аргументы следуют в обратном порядке. Используется в эстетических целях.

Определение:

```
forM_ :: Monad m => [a] -> (a -> m b) -> m ()
forM_ = flip mapM_
```

Функция: sequence

Описание: последовательно исполняет каждое действие слева направо, собирая результат выполнения этих действия и возвращая его.

Определение:

```
sequence :: Monad m => [m a] -> m [a]
sequence []     = return []
sequence (c:cs) = do x  <- c
                     xs <- sequence cs
                     return (x:xs)
```

Функция: sequence-
Описание: вариант функции sequence, используемый тогда, когда результат функции не важен, но важны побочные эффекты монады.
Определение:

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

Функция: (=<<)
Описание: вариант операции (>>=), у которого входные аргументы следуют в обратном порядке. Используется в эстетических целях.
Определение:

```
(=<<) :: Monad m => (a -> m b) -> m a -> m b
f =<< x = x >>= f
```

Функция: join
Описание: функция для развёртывания одного монадического уровня и слияния значений на внутреннем уровне монадической структуры. Разворачивает одну монаду (результат работы функции должен быть понятен из её сигнатуры).
Определение:

```
join :: Monad m => m (m a) -> m a
join x = x >>= id
```

Функция: msum
Описание: обобщение функции sum (см. стр. 311) для списков на произвольную монаду, у которой имеется единица и операция сложения (ассоциативная операция полугруппы).

Определение:

```
msum :: MonadPlus m => [m a] -> m a
msum = foldr mplus mzero
```

Функция: filterM

Описание: обобщение функции filter (см. стр. 354) для списков на произвольную монаду.

Определение:

```
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM _ []     = return []
filterM p (x:xs) = do flg <- p x
                      ys  <- filterM p xs
                      return (if flg
                                then x:ys
                                else ys)
```

Функция: mapAndUnzipM

Описание: функция, которая сочетает в себе эффекты функций map (см. стр. 356) и unzip (см. стр. 270) для произвольной монады. Функция mapAndUnzip применяет заданную функцию над каждым элементом заданного списка, возвращает пару списков по результату действия входной функции.

Определение:

```
mapAndUnzipM :: Monad m => (a -> m (b, c)) -> [a] -> m ([b], [c])
mapAndUnzipM f xs = sequence (map f xs) >>= return . unzip
```

Функция: zipWithM

Описание: обобщает функцию zipWith (см. стр. 269) для списков на произвольную монаду.

Определение:

```
zipWithM :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM f xs ys = sequence (zipWith f xs ys)
```

Функция: zipWith-

Описание: вариант функции zipWithM, используемый тогда, когда результат функции не важен, но важны побочные эффекты монады.

Определение:

```
zipWithM_ :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m ()
zipWithM_ f xs ys = sequence_ (zipWith f xs ys)
```

Функция: `foldM`
Описание: обобщает функцию `foldl` (см. стр. 253) для списков на произвольную монаду.
Определение:

```
foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
foldM _ a []     = return a
foldM f a (x:xs) = f a x >>= \fax -> foldM f fax xs
```

Необходимо помнить, что эта функция оперирует своими аргументами слева направо. Так, к примеру

```
foldM f a1 [x1, x2, ..., xm]
```

тождественно

```
do a2 <- f a1 x1
   a3 <- f a2 x2
   ...
   f am xm
```

Если необходима свёртка монады справа налево, то входной список следует обратить.

Функция: `foldM-`
Описание: вариант функции `foldM`, используемый тогда, когда результат функции не важен, но важны побочные эффекты монады.
Определение:

```
foldM_ :: Monad m => (a -> b -> m a) -> a -> [b] -> m ()
foldM_ f a xs = foldM f a xs >> return ()
```

Функция: `replicateM`
Описание: обобщает функцию `replicate` (см. стр. 259) для списков на произвольную монаду.

Определение:

```
replicateM :: Monad m => Int -> m a -> m [a]
replicateM n x = sequence (replicate n x)
```

Функция: replicateM-
Описание: вариант функции replicateM, используемый тогда, когда результат функции не важен, но важны побочные эффекты монады.
Определение:

```
replicateM_ :: Monad m => Int -> m a -> m ()
replicateM_ n x = sequence_ (replicate n x)
```

Функция: guard
Описание: аналог охраны для произвольной монады, для которой определена единица и ассоциативная операция полугруппы.
Определение:

```
guard :: MonadPlus m => Bool -> m ()
guard True  = return ()
guard False = mzero
```

Функция: when
Описание: исполняет монадическое действие, если первый аргумент равен True.
Определение:

```
when :: (Monad m) => Bool -> m () -> m ()
when p s = if p
             then s
             else return ()
```

Функция: unless
Описание: зеркальный вариант функции when, исполняющий монадическое действие, если первый аргумент равен False.
Определение:

```
unless :: (Monad m) => Bool -> m () -> m ()
unless p s = if p
               then return ()
               else s
```

Функция: `liftM`

Описание: втягивает одноаргументную функцию в монаду таким образом, что результат выполнения функции становится обёрнутым монадой.

Определение:

```
liftM :: (Monad m) => (a1 -> r) -> m a1 -> m r
liftM f m1 = do x1 <- m1
                return (f x1)
```

Функция: `liftM2`

Описание: втягивает двухаргументную функцию в монаду таким образом, что результат выполнения функции становится обёрнутым монадой.

Определение:

```
liftM2 :: (Monad m) => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
liftM2 f m1 m2 = do x1 <- m1
                    x2 <- m2
                    return (f x1 x2)
```

Функция: `liftM3`

Описание: втягивает трёхаргументную функцию в монаду таким образом, что результат выполнения функции становится обёрнутым монадой.

Определение:

```
liftM3 :: (Monad m) => (a1 -> a2 -> a3 -> r) -> m a1 -> m a2 -> m a3 -> m r
liftM3 f m1 m2 m3 = do x1 <- m1
                       x2 <- m2
                       x3 <- m3
                       return (f x1 x2 x3)
```

Функция: `liftM4`

Описание: втягивает четырёхаргументную функцию в монаду таким образом, что результат выполнения функции становится обёрнутым монадой.

Определение:

```
liftM4 :: (Monad m) => (a1 -> a2 -> a3 -> a4 -> r) -> m a1 -> m a2 -> m a3 -> m a4 -> m r
liftM4 f m1 m2 m3 m4 = do x1 <- m1
                          x2 <- m2
                          x3 <- m3
                          x4 <- m4
                          return (f x1 x2 x3 x4)
```

Функция: liftM5

Описание: втягивает пятиаргументную функцию в монаду таким образом, что результат выполнения функции становится обёрнутым монадой.

Определение:

```
liftM5 :: (Monad m) => (a1 -> a2 -> a3 -> a4 -> a5 -> r)
                    -> m a1 -> m a2 -> m a3 -> m a4 -> m a5 -> m r
liftM5 f m1 m2 m3 m4 m5 = do x1 <- m1
                             x2 <- m2
                             x3 <- m3
                             x4 <- m4
                             x5 <- m5
                             return (f x1 x2 x3 x4 x5)
```

Функция: ap

Описание: во многих случаях функции семейства liftM для функций разного числа аргументов можно заменить последовательным применением этой функции.

Определение:

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap = liftM2 id
```

Например, следующий код

```
return f `ap` x1 `ap` ... `ap` xn
```

эквивалентен коду

```
liftMn f x1 x2 ... xn
```

7.5.1. Модуль Fix

Модуль Fix представляет описание программных сущностей, которые описывают монадические неподвижные точки. Детальное описание этого понятия доступно в работе [5]. Данный модуль является «подчинённым» по отношению к модулю Monad, поэтому его импорт выглядит следующим образом:

```
import Control.Monad.Fix
```

Кроме того, если модуль Monad уже подключён, импортировать модуль Fix нет необходимости.

В этом модуле описан один главный класс, который и является интерфейсом тех типов, которые имеют монадическую неподвижную точку. Определение этого класса выглядит следующим образом:

```
class Monad m => MonadFix m where
  mfix :: (a -> m a) -> m a
```

Любой экземпляр этого класса должен удовлетворять следующим законам:

1) *Чистота*:

```
mfix (return . h) = return (fix h)
```

2) *Сокращение влево*:

```
mfix (\x -> a >>= \y -> f x y) = a >>= \y -> mfix (\x -> f x y)
```

3) *Скольжение* (для строгих функций `h`):

```
mfix (\x -> mfix (\y -> f x y)) = mfix (\x -> f x x)
```

4) *Вложенность*:

```
u <*> pure y = pure ($ y) <*> u
```

Единственный метод класса `mfix` возвращает неподвижную точку заданной функции `f`. Метод выполняет функцию `f` только один раз с подачей на вход функции её результата, полученного на предыдущем шаге. Следовательно, если функция `f` не является строгой, метод `mfix` будет расходиться.

Экземплярами этого класса являются типы: `IO`, `Maybe`, `[]`, `ST s`, `(->) r`.

В рассматриваемом модуле описана также одна утилитарная функция:

Функция: `fix`

Описание: возвращает наименьшую неподвижную точку заданной функции `f`, то есть такое наименьшее значение `x`, что `f x = x`.

Определение:

```
fix :: (a -> a) -> a
fix f = let x = f x in x
```

7.5.2. Модуль Instances

В модуле Instances определены экземпляры классов Functor (см. стр. 211) и Monad (см. стр. 211) для некоторых дополнительных типов данных. Также в этом классе заново определены сами классы для исключения импорта модулей, содержащих первоначальные определения. Само собой разумеется, что определения классов тождественны тем, что даны в модуле Monad.

Данный модуль является «подчинённым» модулю Monad, поэтому его импорт выглядит следующим образом:

```
import Control.Monad.Instances
```

Наконец, в этом модуле определены экземпляры следующих типов. Для класса Functor определены экземпляры для типов (->), (,) и Either a. Для класса Monad определён экземпляр для типа типов (->).

7.5.3. Модуль ST

Модуль ST содержит описания программных сущностей, предназначенных для работы со строгими преобразователями монады State. Теоретическое обоснование этого процесса приведено в работе [13]. Использование модуля:

```
import Control.Monad.ST
```

В данном модуле описано два алгебраических типа данных, которые необходимы для работы с преобразователями монады State. Оба этих алгебраических типа данных реализованы в виде примитивов, поэтому их определения недоступны в исходных кодах модулей.

Тип: ST

Описание: строгий тип для представления преобразователя монады State. Вычисления в рамках этого типа преобразовывают состояние типа s и возвращают некоторое значение типа a. При этом тип s (тип состояния) может быть одним из двух: либо это неинстанциированный тип (в рамках вызова функции runST), либо это тип RealWorld (в рамках вызова функции stToIO).

Определение:

Тип определён в виде примитива.

Этот алгебраический тип данных имеет экземпляры для следующих классов: Typeable2, Functor, Monad, MonadFix, MArray (в различных вариантах) и Show.

Тип: `RealWorld`
Описание: служебный абстрактный и примитивный тип данных, используемый для внутренних целей системы типизации при работе с преобразователем монады `State`. Нет никакой необходимости использовать этот тип в своих программах.
Определение:

Тип определён в виде примитива.

Тип `RealWorld` имеет экземпляр для класса `Typeable`.

Функция: `runST`
Описание: возвращает значение, вычисленное в рамках преобразователя монады `State`. Директива **forall** в объявлении типа гарантирует, что внутреннее состояние недоступно вне преобразователя монады.
Определение:

```
runST :: (forall s . ST s a) -> a
```

Функция определена в виде примитива.

Функция: `fixST`
Описание: позволяет использовать результат вычислений в рамках преобразователя монады `State` в ленивых вычислениях. Необходимо отметить, что если первый аргумент этой функции является строгим, то результатом будет ($\perp$).
Определение:

```
fixST :: (a -> ST s a) -> ST s a
```

Функция определена в виде примитива.

Функция: `stToIO`
Описание: переводит значение из преобразователя монады `State` в монаду `IO`. Для этого использует служебный тип `RealWorld`.
Определение:

```
stToIO :: ST RealWorld a -> IO a
```

Функция определена в виде примитива.

Модуль `Lazy`

Модуль `Lazy` содержит те же самые определения, что и модуль `ST`, за исключением того, что он поддерживает отложенные вычисления. Поскольку он является «зависимым» модулем от `ST`, его импорт должен выглядеть следующим образом:

```
import Control.Monad.ST.Lazy
```

Дополнительно к алгебраическим типам данных и функциям из модуля ST рассматриваемый модуль содержит две функции для конвертации преобразователя монады State из строгого в ленивый и обратно.

Функция: strictToLazyST

Описание: преобразует строгий преобразователь монады State в ленивый. Строгий преобразователь, переданный в эту функцию, не вычисляется, пока его значение не понадобится.

Определение:

```
strictToLazyST :: ST s a -> ST s a
```

Функция определена в виде примитива.

Функция: lazyToStrictST

Описание: конвертирует нестрогий преобразователь монады State в строгий.

Определение:

```
lazyToStrictST :: ST s a -> ST s a
```

Функция определена в виде примитива.

Модуль Strict

В целях единообразия (по отношению к модулю Lazy) в стандартной поставке библиотек языка Haskell имеется модуль Strict, который является «подчинённым» модулю ST. Этот модуль всего лишь реимпортирует сам модуль ST, не добавляя никаких новых определений.

7.6. Модуль Parallel

Модуль Parallel является экспериментальным и содержит описание пары функций для организации параллельных вычислений. В отличие от модуля Concurrent этот модуль может использоваться на архитектурах, поддерживающих настоящее распараллеливание вычислительных процессов. Подключение модуля производится следующим образом:

```
import Control.Parallel
```

Функция: `par`
Описание: указывает, что было бы неплохо произвести процесс вычислений, заданный первым аргументом, параллельно с вычислением второго аргумента. Возвращает вычисленное значение второго аргумента.
Определение:

```
par :: a -> b -> b
```

Функция определена в виде примитива.

Семантически выражение a ‘par‘ b эквивалентно `b`.

Обычно функция `par` используется тогда, когда значение `a` необходимо получить позже. Также хорошей идеей является гарантирование того, что `a` является нетривиальным вычислением, чтобы затраты на организацию параллельного вычислительного процесса «окупились».

Необходимо отметить, что настоящий параллелизм поддерживается только на некоторых архитектурах с определёнными трансляторами языка Haskell. Так, к примеру, компилятор GHC необходимо запускать с ключом `-threaded`.

Функция: `seq`
Описание: возвращает значение ($\perp$), если первый аргумент равен ($\perp$). В противном случае возвращает значение второго аргумента. Эта функция обычно используется для оптимизации вычислений посредством отказа от ненужной (в некотором конкретном случае) ленивости.
Определение:

```
seq :: a -> b -> b
```

Функция определена в виде примитива.

Глава 8.

Пакет модулей `Data`

Пакет модулей `Data` содержит набор модулей, использующихся для работы с данными различной природы: начиная от обычных целых чисел и строк и заканчивая разнообразными массивами, хеш-таблицами, деревьями и т. д. Все модули пакета расширяют стандартные определения типов данных из модуля `Prelude` и предоставляют разработчику программного обеспечения массу дополнительных возможностей для обработки значений различных типов данных.

8.1. Модуль `Array`

Модуль `Array` содержит описание программных сущностей, позволяющих работать со строгими массивами. Подключение этого модуля производится следующим образом:

```
import Data.Array
```

Необходимо отметить, что в модуле `IArray` (описывается ниже, см. подраздел 8.1.3.) описываются программные сущности, которые позволяют работать с неизменяемыми массивами (к тому же в этом модуле предоставлены более общие интерфейсы к массивам). Все такие программные сущности имеют те же самые наименования. Если есть необходимость работы с более общим интерфейсом, необходимо подключить модуль `IArray`.

Рассматриваемый модуль импортирует, использует и реимпортирует модуль Ix (см. раздел 8.18.), который описывает типы, предназначенные для индексации массивов. Если подключён модуль Array, то дополнительно подключать модуль Ix не нужно.

Язык Haskell предоставляет в распоряжение программиста неизменяемые строгие и нестрогие массивы, которые можно рассматривать в виде функций, чьи области определения изоморфны множествам последовательно расположенных целых чисел. Ограниченные таким образом функции можно достаточно эффективно реализовать (в отличие, например, от списков произвольной длины), и программист может рассчитывать на быстрый доступ к произвольному элементу массива. Для гарантирования возможности быстрого доступа массивы реализованы в виде алгебраического типа данных, а не в виде обобщённых функций.

Тип: Array

Описание: тип неизменяемого нестрогого массива.

Определение:

```
data Array i e = ...
```

Сам тип реализован в виде примитива. Компонент i является типом индексов массива, а компонент e является типом элементов массива. Тип Array имеет экземпляры для следующих классов: Typeable2, IArray, Foldable, Functor, FunctorM, Traversable, Data, Eq, NFData, Ord, Read и Show.

Функции для работы с массивами разделяются на функции создания, доступа, модернизации значений и изменения индексов.

Функция: array

Описание: создаёт массив в заданном диапазоне индексов с заданными значениями, соответствующими индексам.

Определение:

```
array :: Ix i  => (i, i) -> [(i, e)] -> Array i e
```

Функция определена в виде примитива.

Первый аргумент функции определяет пару индексов — наименьший и наибольший. Все остальные индексы расположены между ними (поэтому на тип индексов накладывается ограничение в наличии экземпляра класса Ix). Второй аргумент этой функции является списком пар вида (*индекс*, *элемент*). Обычно этот список строится при помощи генератора.

Значение массива не определено ($\perp$), если индекс выходит за заданные границы. Кроме того, стандартом языка Haskell определяется, что если во входном списке несколько значений сопоставлены с одним и тем же индексом, то значение массива для этого индекса также не определено. Однако компилятор GHC возвращает в данном случае последнее ассоциированное значение. Кроме того, массивы всегда строги по отношению к индексам (в виду перечисленных ограничений), но являются нестрогими по отношению к элементам. Ну и, наконец, во входном списке с соответствиями может быть представлен не каждый индекс в указанных границах. Для несуществующих индексов значение массива также не определено.

Если массив был создан таким образом, что его нижняя граница больше верхней, то массив считается пустым, но существующим. Любое обращение к элементам такого массива вызовет ошибку, однако с самим пустым массивом производить операции возможно.

Функция: `listArray`

Описание: создаёт массив из пары индексов (нижний и верхний) и из списка, в котором значения полагаются идущими по порядку от нижнего индекса к верхнему.

Определение:

```
listArray :: Ix i => (i, i) -> [e] -> Array i e
```

Функция определена в виде примитива.

Функция: `accumArray`

Описание: также создаёт массив, но работает с повторяющимися индексами в списке соответствий, используя накопляющую функцию, которая комбинирует значения из списка соответствий с одинаковыми индексами.

Определение:

```
accumArray :: Ix i => (e -> a -> e) -> e -> (i, i) -> [(i, a)] -> Array i e
```

Функция определена в виде примитива.

Первым аргументом является аккумулирующая функция, вторым — начальное значение для накопления результата. Остальные аргументы такие же, как и у функции `array` (см. стр. 227).

Например, для заданного списка соответствий значений некоторым индексам следующая функция строит гистограмму:

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

Если накапливающая функция является строгой, то получающийся на её основе массив также строг по своим значениям (равно как и по индексам, по которым массивы строги всегда).

Функция: `(!)`
Описание: возвращает значение массива для заданного индекса.
Определение:

```
(!) :: Ix i => Array i e -> i -> e
```

Функция определена в виде примитива.

Функция: `bounds`
Описание: возвращает пару, состоящую из нижней и верхней границ заданного массива.
Определение:

```
bounds :: Ix i => Array i e -> (i, i)
```

Функция определена в виде примитива.

Функция: `indices`
Описание: возвращает список индексов заданного массива.
Определение:

```
indices :: Ix i => Array i e -> [i]
```

Функция определена в виде примитива.

Функция: `elems`
Описание: возвращает список элементов заданного массива.
Определение:

```
elems :: Ix i => Array i e -> [e]
```

Функция определена в виде примитива.

Функция: `assocs`

Описание: возвращает список соответствий (пар вида (*индекс*, *значение*)) для заданного массива, причём список отсортирован в соответствии с увеличением индексов исходного массива.

Определение:

```
assocs :: Ix i => Array i e -> [(i, e)]
```

Функция определена в виде примитива.

Функция: `(//)`

Описание: создаёт такой же массив, как и первый аргумент, за исключением того, что элементы из списка соответствий (второй аргумент) заменяют элементы старого массива.

Определение:

```
(//) :: Ix i => Array i e -> [(i, e)] -> Array i e
```

Функция определена в виде примитива.

Повторяющиеся индексы в списке соответствий обрабатываются также, как и для функции `array` (см. стр. 227). Стандарт языка говорит, что значения массива на таких индексах не определено ($\perp$), но компилятор GHC, тем не менее, возвращает значение, которое стояло последним в списке соответствий для данного индекса.

Функция: `accum`

Описание: получает на вход аккумулирующую функцию, массив и список соответствий, а возвращает массив, в котором значения переработаны аккумулирующей функцией.

Определение:

```
accum :: Ix i => (e -> a -> e) -> Array i e -> [(i, a)] -> Array i e
```

Функция определена в виде примитива.

Функция: `ixmap`

Описание: позволяет производить изменения в индексации массивов. создаёт массив, в котором индексы элементов заменены в соответствии с заданной функцией, а элементы такие же, как и в исходном массиве.

Определение:

```
ixmap :: (Ix i, Ix j) => (i, i) -> (i -> j) -> Array j e -> Array i e
```

Функция определена в виде примитива.

Подобная трансформация массива может быть проделана при помощи метода `fmap` (см. стр. 211) из экземпляра класса `Functor` для типа `Array`.

8.1.1. Модуль Base

Модуль `Base` содержит описания примитивов, которые используются для определения программных сущностей в модулях `IArray` (см. подраздел 8.1.3.) и `MArray` (см. подраздел 8.1.5.). Этот модуль входит в стандартную поставку, однако он не должен напрямую использоваться при разработке программного обеспечения. Вместо него необходимо подключать либо модуль `IArray`, либо модуль `MArray`.

8.1.2. Модуль Diff

Модуль `Diff` позволяет создавать функциональные массивы с константным временам обновления элементов (функция (`//`) — см. стр. 369). Данный модуль необходимо подключать следующим образом:

```
import Data.Array.Diff
```

Дифференциальные массивы имеют неизменяемый интерфейс, однако используют примитивные функции для ускорения доступа к своим элементам. Если к такому массиву применяется функция (`//`), то в нём происходит физическая замена элементов в старой памяти (деструктивное присваивание). Старый массив просто заменяет свои внутренние поля, без изменения внешнего поведения.

Такая технология позволяет производить выборку элементов массива (функция (`!`) — см. стр. 325) за время $O(1)$, а замену элементов массива за время $O(n)$, где n — количество изменяемых элементов.

Однако дифференциальные массивы позволяют получать доступ и к старым элементам, которые были заменены в процессе работы. Это значит, что все эти элементы сохраняются, однако доступ к ним постепенно становится всё медленнее и медленнее (по мере замены новыми значениями).

Тип: `IOToDiffArray`
Описание: главный алгебраический тип данных для описания дифференциальных массивов.
Определение:

```
data IOToDiffArray a i e = ...
```

Тип определён в виде примитива.

Произвольное значение типа `MArray` (см. подраздел 8.1.5.) может быть преобразовано к дифференциальному массиву. Этот алгебраический тип данных имеет около двадцати различных экземпляров класса `IArray` для многих примитивных типов данных (в том числе и для типов, представляющих значения из внешних языков программирования).

Для облегчения работы с дифференциальными массивами определено два дополнительных синонима. Первый является типом для полностью полиморфных ленивых ограниченных дифференциальных массивов:

```
type DiffArray = IOToDiffArray IOArray
```

Второй синоним определяет тип для представления строгих неограниченных дифференциальных массивов, в которых могут содержаться элементы только примитивных типов:

```
type DiffUArray = IOToDiffArray IOUArray
```

В рассматриваемый модуль импортируется модуль `IArray` для предоставления общих интерфейсов к массивам.

Также в этом модуле описывается три примитивных функции, которые, однако, могут понадобиться при определении дополнительных экземпляров класса `IArray` для типа `IOToDiffArray` при использовании произвольных типов элементов. К этим функциям относятся следующие:

Функция: `newDiffArray`
Описание: создаёт новый дифференциальный массив на основе пары индексов и списка соответствий.
Определение:

```
newDiffArray :: (MArray a e IO, Ix i) => (i, i) ->
                                         [(Int, e)] ->
                                         IO (IOToDiffArray a i e)
```

Функция определена в виде примитива.

Функция: `readDiffArray`
Описание: возвращает значение из дифференциального массива по индексу.
Определение:

```
readDiffArray :: (MArray a e IO, Ix i) => IOToDiffArray a i e -> Int -> IO e
```

Функция определена в виде примитива.

Функция: `replaceDiffArray`
Описание: заменяет значения в дифференциальном массиве в соответствии с заданным списком соответствий.
Определение:

```
replaceDiffArray :: (MArray a e IO, Ix i) => IOToDiffArray a i e ->
                                             [(Int, e)] ->
                                             IO (IOToDiffArray a i e)
```

Функция определена в виде примитива.

8.1.3. Модуль IArray

В модуле `IArray` описывается общий интерфейс к произвольным массивам, а потому к этому интерфейсу добавлен набор утилитарных функций для работы с произвольным массивом. Эти функции обобщают одноимённые функции из головного модуля `Array` (см. раздел 8.1.). Также в этом модуле обобщается и сам алгебраический тип данных, представляющий массивы. Поэтому при подключении модуля `IArray`

```
import Data.Array.IArray
```

подключать модуль `Array` нельзя (и наоборот).

Класс: `IArray`
Описание: интерфейс к произвольному массиву.

Определение:

```
class IArray a e where
  bounds :: Ix i => a i e -> (i, i)
```

Описывает единственный метод, который возвращает границы области индексации массивов в виде пары. Все остальные утилитарные функции построены с использованием этого метода.

Для этого класса определено 35 различных экземпляров для разнообразных массивов, хранящих элементы самых разных типов. Все основные примитивные типы данных охвачены экземплярами этого класса для соответствующих массивов.

Кроме уже описанных в разделе 8.1. функций в этом модуле определена ещё одна функция.

Функция: `amap`

Описание: создаёт новый массив на основе заданного при помощи применения к каждому элементу исходного массива заданной функции. Аналог функции `map` (см. стр. 356) для списков.

Определение:

```
amap :: (IArray a e', IArray a e, Ix i) => (e' -> e) -> a i e' -> a i e
```

Функция определена в виде примитива.

8.1.4. Модуль `IO`

Модуль `IO` (из пакета `Data`) содержит описания примитивных типов для представления ограниченных и неограниченных массивов в монаде `IO`. Использование модуля:

```
import Data.Array.IO
```

Тип: `IOArray`

Описание: изменяемый ограниченный нестрогий массив в рамках монады `IO`. Первое поле типа отвечает за индексы массива (тип поля должен обязательно иметь экземпляр класса `Ix`), второе поле отвечает за элементы массива произвольного типа.

Определение:

```
data IOArray i e = ...
```

Тип определён в виде примитива.

Данный тип имеет определённые экземпляры для следующих классов: Typeable2, MArray, IArray, Eq.

Тип: IOUArray

Описание: изменяемый неограниченный строгий массив в рамках монады IO. Первое поле типа отвечает за индексы массива (тип поля должен обязательно иметь экземпляр класса Ix), второе поле отвечает за элементы массива. Элементы массива могут быть только определённых типов: см. подраздел 8.1.5. со списком экземпляров.

Определение:

```
data IOUArray i e = ...
```

Тип определён в виде примитива.

Для этого типа определено более 30 различных экземпляров для классов IArray (см. стр. 233) и MArray (см. стр. 237), хранящих элементы самых разных типов. Все основные примитивные типы данных охвачены экземплярами этого класса для соответствующих массивов.

Функция: castIOUArray

Описание: преобразует входной массив типа IOUArray в такой же, но содержащий элементы иного типа. При этом все элементы в результате становятся неопределёнными.

Определение:

```
castIOUArray :: IOUArray ix a -> IO (IOUArray ix b)
```

Функция определена в виде примитива.

Функция: hGetArray

Описание: считывает заданное количество элементов из входного файла (потока) непосредственно в массив. Первый аргумент определяет входной файл, второй — массив, в который производится чтение файла (элементы типа Word8), третий — требуемое количество элементов. Функция возвращает количество считанных элементов, которое может быть меньше заказанного.

Определение:

```
hGetArray :: Handle -> IOUArray Int Word8 -> Int -> IO Int
hGetArray handle arr count = do bds <- getBounds arr
                                if (count < 0 || count > rangeSize bds)
                                  then illegalBufferSize handle "hGetArray" count
                                  else get 0
 where
   get i | i == count = return i
         | otherwise  = do error_or_c <- try (hGetChar handle)
                           case error_or_c of
                             Left ex | isEOFError ex -> return i
                                     | otherwise     -> ioError ex
                             Right c -> do unsafeWrite arr i (fromIntegral (ord c))
                                           get (i + 1)
```

Функция: `hPutArray`

Описание: записывает заданное количество элементов из массива в выходной файл (поток). Первый аргумент определяет выходной файл, второй — массив-источник с элементами типа `Word8`, третий — требуемое количество элементов, которое должно быть записано в файл.

Определение:

```
hPutArray :: Handle -> IOUArray Int Word8 -> Int -> IO ()
hPutArray handle arr count = do bds <- getBounds arr
                                if (count < 0 || count > rangeSize bds)
                                  then illegalBufferSize handle "hPutArray" count
                                  else put 0
 where
   put i | i == count = return ()
         | otherwise  = do w <- unsafeRead arr i
                           hPutChar handle (chr (fromIntegral w))
                           put (i + 1)
```

Модуль `Internals`

Служебный модуль `Internals`, в котором в виде примитивов определяются экземпляры классов `IArray` и `MArray` для типа `IOUArray`. Не должен использоваться напрямую, поскольку он сам всегда включён в модуль `IO` из пакета `Data`.

8.1.5. Модуль MArray

Модуль MArray содержит описания класса и утилитарных функций, которые обеспечивают работу с изменяемыми массивами. Использование модуля:

```
import Data.Array.MArray
```

Типы, которые поддерживают описываемый в этом модуле интерфейс (класс), в свою очередь описаны в модулях IO (см. подраздел 8.1.4.), ST (см. подраздел 8.1.6.) и Storable (см. подраздел 8.1.7.).

Класс: MArray

Описание: описывает интерфейс к изменяемым массивам.

Определение:

```
class Monad m => MArray a e m where
  getBounds :: Ix i => a i e -> m (i, i)
  newArray  :: Ix i => (i, i) -> e -> m (a i e)
  newArray_ :: Ix i => (i, i) -> m (a i e)
```

Массив, которые может быть экземпляром этого класса, должен иметь форму (a i e), где a — конструктор массива, i — тип индексов массива (этот тип должен в обязательном порядке быть экземпляром класса Ix), e — тип элементов, хранящихся в массиве.

Класс MArray параметризуется и типом массива, и типом элементов, которые в нём хранятся (поэтому экземпляры этого класса могут определяться в том же стиле, что и для класса IArray — см. стр. 233). Кроме того, класс также параметризуется монадой m, в которой происходит работа с изменяемым массивом.

Метод getBounds возвращает пару индексов массива — самый нижний и самый верхний. Метод newArray создаёт новый массив, в котором каждый элемент в заданном интервале проинициализирован заданным значением. Метод newArray_ создаёт новый массив, в котором каждое значение не определено (равно ($\perp$)).

Для класса MArray определено множество экземпляров типов IOUArray (см. стр. 235), StorableArray (см. стр. 244) и STArray (см. стр. 242), которые охватывают все примитивные типы элементов.

В этом модуле также подключён модуль Ix (см. раздел 8.18.), который используется для индексации массивов. При подключении модуля MArray можно уже не подключать модуль Ix.

Для манипуляции с изменяемыми монадическими массивами используется набор утилитарных функций.

Функция: `newArray`

Описание: реализация одноимённого метода класса `MArray`, используемая по умолчанию.

Определение:

```
newArray :: (MArray a e m, Ix i) => (i, i) -> e -> m (a i e)
newArray (l, u) init = do marr <- newArray_ (l, u)
                          sequence_ [unsafeWrite marr i init |
                                     i <- [0..rangeSize (l, u) - 1]]
                          return marr
```

Функция: `newArray-`

Описание: реализация одноимённого метода класса `MArray`, используемая по умолчанию.

Определение:

```
newArray_ :: (MArray a e m, Ix i) => (i, i) -> m (a i e)
newArray_ (l, u) = newArray (l, u) arrEleBottom
```

Функция: `newListArray`

Описание: создаёт изменяемый массив, наполняя его значениями из заданного списка. Считается, что значения в списке идут по порядку от наименьшего к наибольшему.

Определение:

```
newListArray :: (MArray a e m, Ix i) => (i, i) -> [e] -> m (a i e)
newListArray (l, u) es
  = do marr <- newArray_ (l, u)
       let n = rangeSize (l, u)
       let fillFromList i xs | i == n    = return ()
                             | otherwise = case xs of
                                             []   -> return ()
                                             y:ys -> unsafeWrite marr i y >>
                                                     fillFromList (i + 1) ys
       fillFromList 0 es
       return marr
```

Функция: `readArray`

Описание: возвращает значение элемента (по индексу) из изменяемого массива.

Определение:

```
readArray :: (MArray a e m, Ix i) => a i e -> i -> m e
readArray marr i = do (l, u) <- getBounds marr
                      unsafeRead marr (index (l, u) i)
```

Функция: writeArray

Описание: записывает значение в изменяемый массив по заданному индексу.

Определение:

```
writeArray :: (MArray a e m, Ix i) => a i e -> i -> e -> m ()
writeArray marr i e = do (l, u) <- getBounds marr
                         unsafeWrite marr (index (l, u) i) e
```

Функция: mapArray

Описание: создаёт новый массив, полученный из исходного при помощи применения заданной функции к каждому элементу исходного массива.

Определение:

```
mapArray :: (MArray a e' m, MArray a e m, Ix i) => (e' -> e) -> a i e' -> m (a i e)
mapArray f marr = do (l, u) <- getBounds marr
                     marr' <- newArray_ (l, u)
                     sequence_ [do e <- unsafeRead marr i
                                   unsafeWrite marr' i (f e)
                                | i <- [0..rangeSize (l, u) - 1]]
                     return marr'
```

Функция: mapIndices

Описание: создаёт новый массив, полученный из исходного при помощи применения заданной функции к индексам исходного массива.

Определение:

```
mapIndices :: (MArray a e m, Ix i, Ix j) => (i,i) -> (i -> j) -> a j e -> m (a i e)
mapIndices (l, u) f marr = do marr' <- newArray_ (l, u)
                              sequence_ [do e <- readArray marr (f i)
                                            unsafeWrite marr' (unsafeIndex (l, u) i) e
                                         | i <- range (l, u)]
                              return marr'
```

Функция: getBounds

Описание: возвращает в виде пары нижнюю и верхнюю границу заданного изменяемого массива.

Определение:

```
getBounds :: (MArray a e m, Ix i) => a i e -> m (i, i)
```

Функция определена в виде примитива.

Функция: `getElems`

Описание: возвращает список элементов массива.

Определение:

```
getElems :: (MArray a e m, Ix i) => a i e -> m [e]
getElems marr = do (l, u) <- getBounds marr
                   sequence [unsafeRead marr i | i <- [0..rangeSize (l, u) - 1]]
```

Функция: `getAssocs`

Описание: возвращает список соответствий вида (*индекс*, *значение*).

Определение:

```
getAssocs :: (MArray a e m, Ix i) => a i e -> m [(i, e)]
getAssocs marr = do (l, u) <- getBounds marr
                    sequence [do e <- unsafeRead marr (index (l, u) i)
                                 return (i, e)
                              | i <- range (l, u)]
```

Функция: `freeze`

Описание: конвертирует изменяемый массив (произвольный экземпляр класса `MArray`) в неизменяемый массив (соответствующий экземпляр класса `IArray` — см. стр. 233.

Определение:

```
freeze :: (Ix i, MArray a e m, IArray b e) => a i e -> m (b i e)
```

Функция определена в виде примитива.

Функция: `unsafeFreeze`

Описание: деструктивный вариант функции `freeze`, который не создаёт копию, а замещает в памяти изменяемый массив на неизменяемый. Тем не менее ссылки на изменяемый массив остаются рабочими, поэтому функция является небезопасной — любое обращение по ссылкам на изменяемый массив приведёт к ошибке.

Определение:

```
unsafeFreeze :: (Ix i, MArray a e m, IArray b e) => a i e -> m (b i e)
```

Функция определена в виде примитива.

Функция: `thaw`

Описание: функция, обратная к функции `freeze`. Конвертирует неизменяемый массив в соответствующий изменяемый массив.

Определение:

```
thaw :: (Ix i, IArray a e, MArray b e m) => a i e -> m (b i e)
```

Функция определена в виде примитива.

Функция: `unsafeThaw`

Описание: деструктивный вариант функции `thaw`, который не создаёт копию, а замещает в памяти неизменяемый массив на изменяемый. Тем не менее ссылки на неизменяемый массив остаются рабочими, поэтому функция является небезопасной — любое обращение по ссылкам на неизменяемый массив приведёт к ошибке.

Определение:

```
unsafeThaw :: (Ix i, IArray a e, MArray b e m) => a i e -> m (b i e)
```

Функция определена в виде примитива.

Небезопасные деструктивные функции `unsafeFreeze` и `unsafeThaw` должны использоваться с осторожностью. После их применения старые ссылки на заменённые массивы использоваться не должны. Эти функции полезны при работе с массивами типов `IOArray` (см. стр. 234), `IOUArray` (см. стр. 235), `STArray` (см. стр. 242) и `STUArray` (см. стр. 242), для которых эти функции выполняются за время $O(1)$.

8.1.6. Модуль ST

Модуль `ST` содержит описание реализации изменяемых массивов (ограниченных и неограниченных) для монады `ST` (преобразователь монады `State` — см. подраздел 7.5.3.). Использование этого модуля выглядит так:

```
import Data.Array.ST
```

Этот модуль также импортирует для работы своих функций модуль `MArray` (см. подраздел 8.1.5.), поэтому при использовании рассматриваемого модуля импортировать модуль `MArray` нет необходимости.

В этом модуле описаны два алгебраических типа данных, которые используются для представления ограниченных и неограниченных массивов.

Тип: `STArray`

Описание: Изменяемый ограниченный нестрогий массив в монаде `ST`.

Определение:

```
data STArray s i e = ...
```

Тип определён в виде примитива.

В этом определении тип `s` является типом состояния в монаде `ST`, тип `i` — как обычно, тип индекса (должен быть экземпляром класса `Ix`), а `e` — тип элементов в массиве.

Тип `STArray` имеет экземпляры для следующих классов: `Typeable3`, `MArray` и `Eq`.

Тип: `STUArray`

Описание: Изменяемый неограниченный строгий массив в монаде `ST`.

Определение:

```
data STUArray s i e = ...
```

Тип определён в виде примитива.

В этом определении типы соответствуют тому же, что и в определении типа `STArray`. Этот тип обычно более эффективен (и с точки зрения времени исполнения, и с точки зрения занимаемого пространства), чем тип `STArray`, но он строгий, поэтому если имеется необходимость ленивых вычислений с массивами, надо использовать тип `STArray`.

Тип `STUArray` имеет экземпляр для класса `Typeable3`, а также порядка двадцати различных экземпляров класса `MArray` для различных типов элементов (охватываются все примитивные типы данных).

С перечисленными типами данных работают три прикладные функции.

Функция: `runSTArray`

Описание: безопасный способ создания и работы с изменяемым ограниченным массивом перед возвращением неизменяемого массива для дальнейшего использования. Эта функция не копирует массив перед его возвращением, она использует небезопасную примитивную функцию `unsafeFreezeSTArray`, однако сама функция является безопасной оболочкой к опасной.

Определение:

```
runSTArray :: Ix i => (forall s . ST s (STArray s i e)) -> Array i e
runSTArray st = runST (st >>= unsafeFreezeSTArray)
```

Функция: runSTUArray
Описание: безопасный способ создания и работы с изменяемым неограниченным массивом перед возвращением неизменяемого массива для дальнейшего использования. Эта функция не копирует массив перед его возвращением, она использует небезопасную примитивную функцию unsafeFreezeSTUArray, однако сама функция является безопасной оболочкой к опасной.
Определение:

```
runSTUArray :: Ix i => (forall s . ST s (STUArray s i e)) -> UArray i e
runSTUArray st = runST (st >>= unsafeFreezeSTUArray)
```

Функция: castSTUArray
Описание: преобразует один массив типа STUArray в другой (изменяя при необходимости тип элементов). Все элементы в получаемом массиве не определены.
Определение:

```
castSTUArray :: STUArray s i a -> ST s (STUArray s i b)
```

Функция определена в виде примитива.

8.1.7. Модуль Storable

Модуль Storable содержит описание специального типа изменяемых массивов, которые используются при работе с внешними программами, написанными на языках программирования типа C (с аналогичной методикой работы с кучей). Использование модуля:

```
import Data.Array.Storable
```

Массив данного типа хранится в монаде IO, которая располагает его в последовательных блоках памяти в куче. Элементы массива хранятся в соответствии с правилами, описываемыми классом Storable (см. стр. 468). Такими массивами можно манипулировать из внешних программ, написанных на языках типа C.

Массивы описываемого типа являются более медленными, чем массивы типа `IOUArray` (см. стр. 235), однако они используются именно для связи с внешними программами.

Тип: `StorableArray`

Описание: массив описываемого типа.

Определение:

```
data StorableArray i e = ...
```

Тип определён в виде примитива.

Для этого типа данных определён экземпляр класса `Storeable`.

Функция: `withStorableArray`

Описание: возвращает указатель на массив. Этот указатель должен использоваться только внутри действия монады `IO`, которое передаётся вторым аргументом в эту функцию.

Определение:

```
withStorableArray :: StorableArray i e -> (Ptr e -> IO a) -> IO a
```

Функция определена в виде примитива.

Функция: `touchStorableArray`

Описание: используется для того, чтобы гарантировать наличие указателя на массив до осуществления последней операции с ним. Эта функция должна вызываться последней в наборе операций.

Определение:

```
touchStorableArray :: StorableArray i e -> IO ()
```

Функция определена в виде примитива.

Функция: `unsafeForeignPtrToStorableArray`

Описание: создаёт массив описываемого типа из произвольного указателя на внешнюю память. Ответственность за логическую целостность выполняемой операции полностью ложится на разработчика программного обеспечения, который использует эту небезопасную функцию.

Определение:

```
unsafeForeignPtrToStorableArray :: ForeignPtr e -> (i, i) -> IO (StorableArray i e)
```

Функция определена в виде примитива.

8.1.8. Модуль Unboxed

Модуль `Unboxed` описывает примитивный алгебраический тип данных для представления неизменяемых неограниченных массивов. Использование модуля:

```
import Data.Array.Unboxed
```

В этом модуле описывается единственный алгебраический тип данных и экземпляры необходимых классов для создания интерфейсов к массивам.

Тип: `UArray`
Описание: алгебраический тип данных для представления неизменяемых неограниченных строгих массивов.
Определение:

```
data UArray i e = ...
```

Тип определён в виде примитива.

Этот тип массивов обычно более эффективен (с точки зрения времени исполнения и с точки зрения используемой памяти), чем массивы типа `Array` (см. стр. 227). Это связано с тем, что такие массивы строги.

Для этого типа массивов определено более пятнадцати экземпляров класса `IArray`, которые покрывают собой все основные типы (примитивные в том числе). Кроме того, определены экземпляры для следующих классов: `Typeable2`, `Eq`, `Ord` и `Show`.

8.2. Модуль Bits

Модуль `Bits` описывает класс (и несколько экземпляров для него), предоставляющий интерфейс для применения битовых операций для нумералов. Применение модуля:

```
import Data.Bits
```

В этом модуле описывается единственный класс, чьё определение выглядит следующим образом.

Класс: `Bits`

Описание: интерфейс к значениям, над которыми имеют смысл побитовые операции.

Определение:

```
class Num a => Bits a where
  (.&.)         :: a -> a -> a
  (.|.)         :: a -> a -> a
  xor           :: a -> a -> a
  complement    :: a -> a
  shift         :: a -> Int -> a
  rotate        :: a -> Int -> a
  bit           :: Int -> a
  setBit        :: a -> Int -> a
  clearBit      :: a -> Int -> a
  complementBit :: a -> Int -> a
  testBit       :: a -> Int -> Bool
  bitSize       :: a -> Int
  isSigned      :: a -> Bool
  shiftL        :: a -> Int -> a
  shiftR        :: a -> Int -> a
  rotateL       :: a -> Int -> a
  rotateR       :: a -> Int -> a
```

Считается, что биты нумеруются с нулевого бита, причём нулевой бит является младшим. Для данного класса в рассматриваемом модуле определены экземпляры типов `Int` и `Integer`. Остальные экземпляры определяются в модулях `Int` (пакет `Data`, см. раздел 8.14.) и `Word` (см. раздел 8.34.).

Метод `(.&.)` определяет побитовую операцию «И». Соответственно, метод `(.|.)` определяет побитовую операцию «ИЛИ». Метод `xor` определяет побитовую операцию «Исключающее ИЛИ».

Метод `complement` возвращает значение, в котором все биты обращены (0 в 1 и наоборот), нежели во входном аргументе. Метод `shift` сдвигает аргумент влево на заданное количество позиций. Правый сдвиг для чисел со знаком достигается при помощи этого же метода, применённого к числу с обратным знаком. Метод `rotate` циклически сдвигает влево свой аргумент на заданное количество позиций. Правый циклический сдвиг осуществляется так же, как и для метода `shift`. Для неограниченных типов (например, `Integer`) циклический сдвиг эквивалентен простому сдвигу.

Метод `bit` возвращает значение с установленным `i`-ым битом (`i` задаётся в качестве аргумента). Два метода `setBit` и `clearBit` устанавливают и очищают заданный бит в заданном числе соответственно. Метод `complementBit` обращает заданный бит в заданном числе. Для трёх последних методов можно предусмотреть определения по умолчанию:

```
setBit x i = x .|. bit i

clearBit x i = x .&. complement (bit i)

complementBit x i = x `xor` bit i
```

Метод `textBit` возвращает значение `True`, если заданный бит в заданном числе установлен (равен 1). Метод `bitSize` возвращает количество бит, которое занимает входное значение. Само значение не используется в методе. Для значений неопределённого размера (например, тип `Integer`) метод возвращает неопределённое же значение ($\perp$). Метод `isSigned` возвращает `True`, если входной аргумент является числом со знаком, и `False` в противном случае.

Наконец, группа методов `shiftL`, `shiftR`, `rotateL`, `rotateR` осуществляют сдвиг или циклический сдвиг влево или вправо соответственно на заданное количество позиций. Аргументы всех этих функций должны быть неотрицательными.

8.3. Модуль `Bool`

В модуле `Bool` дублируются описания типа `Bool` (булевские значения истинности) и функции для их обработки. Данный модуль создан в экспериментальном порядке в целях постепенной разгрузки стандартного модуля `Prelude`. Все определённые в модуле `Bool` программные сущности определены и в модуле `Prelude`. Использование:

```
import Data.Bool
```

Соответственно, в рассматриваемый модуль вынесены определения: алгебраического типа данных `Bool` (см. стр. 108), операции (`&&`) (логическое «И» — см. раздел 6.4.), операции (`||`) (логическое «ИЛИ» — см. раздел 6.4.), операции `not` (логическое отрицание — см. стр. 147) и синонима `otherwise` (см. стр. 150),

который является константной функцией, возвращающей значение True для использования в выражениях охраны.

Также в модуле Bool определены экземпляры типа Bool для следующих классов: Bounded, Data, Enum, Eq, Ix, NFData, Ord, Random, Read, Show, Storable, Typeable, IArray, MArray.

8.4. Модуль ByteString

Модуль ByteString содержит определения программных сущностей для работы с байтовыми векторами, основанными на использовании массивов элементов типа Word8. Эти определения являются достаточно эффективными с точки зрения используемого для вычислений времени и занимаемой памяти. Кроме того, байтовые массивы являются строгими, для них существует указатель для использования во внешних программных модулях и библиотеках (написанных на других языках программирования).

В рассматриваемом модуле используется очень много функций, которые имеют те же наименования, что и в стандартном модуле Prelude. Поэтому импортирование модуля должно производиться с квалификацией имени. Например:

```
import qualified Data.ByteString as B
```

Модуль ByteString необходимо использовать вместо модуля PackedString, который объявлен устаревшим и постепенно будет выводиться из пакета стандартных модулей языка Haskell.

Основной и единственный алгебраический тип данных, описанный в этом модуле, — ByteString.

Тип: ByteString

Описание: эффективная реализация строгого вектора, состоящего из значений типа Word8. Этот тип данных содержит только восьмибитовые значения.

Определение:

Тип определён в виде примитива.

Для типа ByteString реализованы экземпляры следующих классов: Eq, Ord, Data, Monoid, Read, Show, Typeable.

Функция: empty

Описание: возвращает пустую восьмибитовую строку. Эффективность — $O(1)$.

Определение:

```
empty :: ByteString
```

Функция определена в виде примитива.

Функция: `singleton`

Описание: конвертирует заданное значение типа `Word8` в восьмибитовую строку с эффективностью $O(1)$.

Определение:

```
singleton :: Word8 -> ByteString
singleton c = unsafeCreate 1 $ \p -> poke p c
```

Функция: `pack`

Описание: конвертирует список значений типа `Word8` в восьмибитовую строку. Работает с эффективностью $O(n)$, где n — количество элементов в исходном списке. Для программ с большим количеством строковых литералов использование этой функции может стать неэффективным. Для компилятора GHC в данном случае рекомендуется использование функции `packAddress`.

Определение:

```
pack :: [Word8] -> ByteString
pack str = unsafeCreate (P.length str) $ \p -> go p str
  where
    go _ []     = return ()
    go p (x:xs) = poke p x >> go (p `plusPtr` 1) xs
```

Функция: `unpack`

Описание: функция, обратная к функции `pack`. Конвертирует заданную восьмибитовую строку в список значений типа `Word8`. Преобразование совершает с эффективностью $O(n)$, где n — количество символов в исходной строке.

Определение:

```
unpack :: ByteString -> [Word8]
unpack (PS _  _ 0) = []
unpack (PS ps s l) = inlinePerformIO $ withForeignPtr ps $ \p ->
                     go (p `plusPtr` s) (l - 1) []
```

```
  where
    go a b c | a `seq` b `seq` c `seq` False = undefined
    go p 0 acc = peek p          >>= \e -> return (e : acc)
    go p n acc = peekByteOff p n >>= \e -> go p (n - 1) (e : acc)
```

Функция: cons

Описание: добавляет заданный символ типа Word8 в начало строки. Аналог конструктора (:) для списков, однако сложность исполнения — $O(n)$.

Определение:

```
cons :: Word8 -> ByteString -> ByteString
cons c (PS x s l) = unsafeCreate (l + 1) $ \p ->
                    withForeignPtr x $ \f ->
                    do poke p c
                       memcpy (p `plusPtr` 1) (f `plusPtr` s) (fromIntegral l)
```

Функция: snoc

Описание: добавляет заданный символ типа Word8 в конец строки. Функция по действию аналогична функции cons. Сложность — $O(n)$, где n — длина строки.

Определение:

```
snoc :: ByteString -> Word8 -> ByteString
snoc (PS x s l) c = unsafeCreate (l+1) $ \p ->
                    withForeignPtr x $ \f ->
                    do memcpy p (f `plusPtr` s) (fromIntegral l)
                       poke (p `plusPtr` l) c
```

Функция: append

Описание: конкатенирует две восьмибитовые строки. Сложность — $O(n)$, где n — сумма длин двух заданных строк.

Определение:

```
append :: ByteString -> ByteString -> ByteString
append xs ys | null xs   = ys
             | null ys   = xs
             | otherwise = concat [xs, ys]
```

Функция: `head`

Описание: возвращает первый символ заданной восьмибитовой строки. Сложность — $O(1)$.

Определение:

```
head :: ByteString -> Word8
head (PS x s l) | l <= 0    = errorEmptyList "head"
                | otherwise = inlinePerformIO $
                              withForeignPtr x $
                              \p -> peekByteOff p s
```

Функция: `last`

Описание: возвращает последний символ заданной восьмибитовой строки. Сложность — $O(1)$.

Определение:

```
last :: ByteString -> Word8
last ps@(PS x s l) | null ps   = errorEmptyList "last"
                   | otherwise = inlinePerformIO $
                                 withForeignPtr x $
                                 \p -> peekByteOff p (s + l - 1)
```

Функция: `tail`

Описание: возвращает заданную восьмибитовую строку без первого символа. Сложность — $O(1)$.

Определение:

```
tail :: ByteString -> ByteString
tail (PS p s l) | l <= 0    = errorEmptyList "tail"
                | otherwise = PS p (s + 1) (l - 1)
```

Функция: `last`

Описание: возвращает заданную восьмибитовую строку без последнего символа. Сложность — $O(1)$.

Определение:

```
last :: ByteString -> Word8
last ps@(PS x s l) | null ps   = errorEmptyList "last"
                   | otherwise = inlinePerformIO $
                                 withForeignPtr x $
                                 \p -> peekByteOff p (s + l - 1)
```

Функция: `null`

Описание: возвращает значение `True`, если заданная восьмибитовая строка пуста. Сложность — $O(1)$.

Определение:

```
null :: ByteString -> Bool
null (PS _ _ l) = assert (l >= 0) $ l <= 0
```

Функция: `length`

Описание: возвращает длину заданной восьмибитовой строки.

Определение:

```
length :: ByteString -> Int
length (PS _ _ l) = assert (l >= 0) $ l
```

Функция: `map`

Описание: применяет заданную функцию к каждому символу исходной восьмибитовой строки. Сложность — $O(n)$, где n — длина строки.

Определение:

```
map :: (Word8 -> Word8) -> ByteString -> ByteString
map f = loopArr . loopMap f
```

Функция: `reverse`

Описание: обращает заданную восьмибитовую строку со сложностью работы $O(n)$, где n — длина строки.

Определение:

```
reverse :: ByteString -> ByteString
reverse (PS x s l) = unsafeCreate l $
                     \p -> withForeignPtr x $
                     \f -> c_reverse p (f `plusPtr` s) (fromIntegral l)
```

Функция: `intersperse`

Описание: возвращает восьмибитовую строку, между каждыми символами которой вставляется заданный первым аргументом символ типа `Word8`. Сложность — $O(n)$, где n — длина строки.

Определение:

```
intersperse :: Word8 -> ByteString -> ByteString
intersperse c ps@(PS x s l) | length ps < 2  = ps
                            | otherwise      = unsafeCreate (2 * l - 1) $
                                               \p -> withForeignPtr x $
                                               \f -> c_intersperse p (f `plusPtr` s)
                                                                     (fromIntegral l) c
```

Функция: `transpose`

Описание: транспонирует матрицу, составленную из восьмибитовых строк.

Определение:

```
transpose :: [ByteString] -> [ByteString]
transpose ps = P.map pack (List.transpose (P.map unpack ps))
```

Функция: `foldl`

Описание: сворачивает заданную восьмибитовую строку при помощи заданного бинарного оператора и начального значения. Функция работает со строкой слева направо.

Определение:

```
foldl :: (a -> Word8 -> a) -> a -> ByteString -> a
foldl f z = loopAcc . loopUp (foldEFL f) z
```

Функция: `foldl'`

Описание: вариант функции `foldl`, являющийся строгим относительно аргумента, выполняющего роль аккумулятора. Для восьмибитовых строк функция `foldl` также является строгой относительно аккумулятора, поэтому данная функция создана исключительно ради совместимости.

Определение:

```
foldl' :: (a -> Word8 -> a) -> a -> ByteString -> a
foldl' = foldl
```

Функция: `foldl1`

Описание: вариант функции `foldl`, берущий в качестве начального значения первый символ восьмибитовой строки.

Определение:

```
foldl1 :: (Word8 -> Word8 -> Word8) -> ByteString -> Word8
foldl1 f ps | null ps   = errorEmptyList "foldl1"
            | otherwise = foldl f (unsafeHead ps) (unsafeTail ps)
```

Функция: `foldl1'`

Описание: вариант функции `foldl1`, являющийся строгим относительно аргумента, выполняющего роль аккумулятора.

Определение:

```
foldl1' :: (Word8 -> Word8 -> Word8) -> ByteString -> Word8
foldl1' f ps | null ps   = errorEmptyList "foldl1'"
             | otherwise = foldl' f (unsafeHead ps) (unsafeTail ps)
```

Функция: `foldr`

Описание: сворачивает заданную восьмибитовую строку при помощи заданного бинарного оператора и начального значения. Функция работает со строкой справа налево.

Определение:

```
foldr :: (Word8 -> a -> a) -> a -> ByteString -> a
foldr k z = loopAcc . loopDown (foldEFL (flip k)) z
```

Функция: `foldr'`

Описание: вариант функции `foldr`, являющийся строгим относительно аргумента, выполняющего роль аккумулятора.

Определение:

```
foldr' :: (Word8 -> a -> a) -> a -> ByteString -> a
foldr' k v (PS x s l) = inlinePerformIO $
                        withForeignPtr x $
                        \ptr -> go v (ptr `plusPtr` (s + l - 1)) (ptr `plusPtr` (s - 1))
  where
    go a b c | a `seq` b `seq` c `seq` False = undefined
    go z p q | p == q    = return z
             | otherwise = do c <- peek p
                              go (c `k` z) (p `plusPtr` (-1)) q
```

Функция: `foldr1`

Описание: вариант функции `foldr`, берущий в качестве начального значения первый символ восьмибитовой строки.

Определение:

```
foldr1 :: (Word8 -> Word8 -> Word8) -> ByteString -> Word8
foldr1 f ps | null ps   = errorEmptyList "foldr1"
            | otherwise = foldr f (last ps) (init ps)
```

Функция: `foldr1'`

Описание: вариант функции `foldr1`, являющийся строгим относительно аргумента, выполняющего роль аккумулятора.

Определение:

```
foldr1' :: (Word8 -> Word8 -> Word8) -> ByteString -> Word8
foldr1' f ps | null ps   = errorEmptyList "foldr1"
             | otherwise = foldr' f (last ps) (init ps)
```

Функция: `concat`

Описание: конкатенирует строки из заданного списка восьмибитовых строк. Сложность — $O(n)$, где n — сумма длин всех строк в списке.

Определение:

```
concat :: [ByteString] -> ByteString
concat []     = empty
concat [ps]   = ps
concat xs     = unsafeCreate len $ \ptr -> go xs ptr
  where
    len = P.sum . P.map length $ xs
    go a b | a `seq` b `seq` False = undefined
    go []            _   = return ()
    go (PS p s l:ps) ptr = do withForeignPtr p $ \fp -> memcpy ptr (fp `plusPtr` s)
                                                                   (fromIntegral l)
                              go ps (ptr `plusPtr` l)
```

Функция: `concatMap`

Описание: применяет функцию к каждому символу заданной восьмибитовой строки, после чего сращивает результаты применения в одну строку.

Определение:

```
concatMap :: (Word8 -> ByteString) -> ByteString -> ByteString
concatMap f = concat . foldr ((:) . f) []
```

Функция: any

Описание: возвращает значение True, если хотя бы один символ восьмибитовой строки удовлетворяет заданному предикату. Сложность — $O(n)$, где n — длина заданной строки.

Определение:

```
any :: (Word8 -> Bool) -> ByteString -> Bool
any _ (PS _ _ 0) = False
any f (PS x s l) = inlinePerformIO $
                   withForeignPtr x $
                   \ptr -> go (ptr `plusPtr` s) (ptr `plusPtr` (s+l))
  where
    go a b | a `seq` b `seq` False = undefined
    go p q | p == q    = return False
           | otherwise = do c <- peek p
                            if f c
                              then return True
                              else go (p `plusPtr` 1) q
```

Функция: all

Описание: возвращает значение True, если все символы восьмибитовой строки удовлетворяют заданному предикату. Сложность — $O(n)$, где n — длина заданной строки.

Определение:

```
all :: (Word8 -> Bool) -> ByteString -> Bool
all _ (PS _ _ 0) = True
all f (PS x s l) = inlinePerformIO $
                   withForeignPtr x $
                   \ptr -> go (ptr `plusPtr` s) (ptr `plusPtr` (s+l))
```

```
  where
    go a b | a `seq` b `seq` False = undefined
    go p q | p == q     = return True  -- end of list
           | otherwise  = do c <- peek p
                             if f c
                               then go (p `plusPtr` 1) q
                               else return False
```

Функция: `maximum`

Описание: возвращает значение наибольшего (по коду) символа в заданной восьмибитовой строке. Сложность — $O(n)$, где n — длина заданной строки.

Определение:

```
maximum :: ByteString -> Word8
maximum xs@(PS x s l) | null xs   = errorEmptyList "maximum"
                      | otherwise = inlinePerformIO $
                                    withForeignPtr x $
                                    \p -> c_maximum (p `plusPtr` s) (fromIntegral l)
```

Функция: `minimum`

Описание: возвращает значение наименьшего (по коду) символа в заданной восьмибитовой строке. Сложность — $O(n)$, где n — длина заданной строки.

Определение:

```
minimum :: ByteString -> Word8
minimum xs@(PS x s l) | null xs   = errorEmptyList "minimum"
                      | otherwise = inlinePerformIO $
                                    withForeignPtr x $
                                    \p -> c_minimum (p `plusPtr` s) (fromIntegral l)
```

Функция: `scanl`

Описание: функция, аналогичная функции `foldl`, однако возвращающая список всех промежуточных значений. Последним значением в таком списке будет как раз значение функции `foldl`, которое она вернула бы на тех же входных аргументах.

Определение:

```
scanl :: (Word8 -> Word8 -> Word8) -> Word8 -> ByteString -> ByteString
scanl f z ps = loopArr . loopUp (scanEFL f) z $ (ps `snoc` 0)
```

Функция: scanl1

Описание: вариант функции scanl, работающий без начального значения. В качестве начального значения принимается первый символ восьмибитовой строки.
Определение:

```
scanl1 :: (Word8 -> Word8 -> Word8) -> ByteString -> ByteString
scanl1 f ps | null ps   = empty
            | otherwise = scanl f (unsafeHead ps) (unsafeTail ps)
```

Функция: scanr

Описание: функция, аналогичная функции foldr, однако возвращающая список всех промежуточных значений. Последним значением в таком списке будет как раз значение функции foldr, которое она вернула бы на тех же входных аргументах.
Определение:

```
scanr :: (Word8 -> Word8 -> Word8) -> Word8 -> ByteString -> ByteString
scanr f z ps = loopArr . loopDown (scanEFL (flip f)) z $ (0 `cons` ps)
```

Функция: scanr1

Описание: вариант функции scanr, работающий без начального значения. В качестве начального значения принимается первый символ восьмибитовой строки.
Определение:

```
scanr1 :: (Word8 -> Word8 -> Word8) -> ByteString -> ByteString
scanr1 f ps | null ps   = empty
            | otherwise = scanr f (last ps) (init ps)
```

Функция: mapAccumL

Описание: данная функция ведёт себя как комбинация функций map и foldl. Она применяет заданную функцию к каждому символу восьмибитовой строки, собирает значения слева направо, после чего возвращает пару, состоящую из аккумулирующего параметра и результирующей строки.
Определение:

```
mapAccumL :: (acc -> Word8 -> (acc, Word8)) -> acc -> ByteString -> (acc, ByteString)
mapAccumL f z = unSP . loopUp (mapAccumEFL f) z
```

Функция: mapAccumR

Описание: данная функция ведёт себя как комбинация функций `map` и `foldk`. Она применяет заданную функцию к каждому символу восьмибитовой строки, собирает значения справа налево, после чего возвращает пару, состоящую из аккумулирующего параметра и результирующей строки.
Определение:

```
mapAccumR :: (acc -> Word8 -> (acc, Word8)) -> acc -> ByteString -> (acc, ByteString)
mapAccumR f z = unSP . loopDown (mapAccumEFL f) z
```

Функция: `mapIndexed`
Описание: применяет заданную индексирующую функцию к восьмибитовой строке. Сложность — $O(n)$, где n — длина заданной строки.
Определение:

```
mapIndexed :: (Int -> Word8 -> Word8) -> ByteString -> ByteString
mapIndexed f = loopArr . loopUp (mapIndexEFL f) 0
```

Функция: `replicate`
Описание: возвращает восьмибитовую строку, состоящую из заданного количества заданного символа. Сложность — $O(n)$, где n — длина результата.
Определение:

```
replicate :: Int -> Word8 -> ByteString
replicate w c | w <= 0    = empty
              | otherwise = unsafeCreate w $
                            \ptr -> memset ptr c (fromIntegral w) >> return ()
```

Функция: `unfoldr`
Описание: разворачивает начальное значение в восьмибитовую строку. Является обратной по отношению к функции `foldr`. Сложность — $O(n)$, где n — длина результата.
Определение:

```
unfoldr :: (a -> Maybe (Word8, a)) -> a -> ByteString
unfoldr f = concat . unfoldChunk 32 64
  where
    unfoldChunk n n' x = case unfoldrN n f x of
                           (s, Nothing) -> s : []
                           (s, Just x') -> s : unfoldChunk n' (n + n') x'
```

Функция: `unfolrdN`

Описание: вариант функции `unfoldr`, который в любом случае ограничивает длину результата значением, переданным в качестве первого параметра. Сложность — $O(n)$, где n — длина результата.

Определение:

```
unfoldrN :: Int -> (a -> Maybe (Word8, a)) -> a -> (ByteString, Maybe a)
unfoldrN i f x0 | i < 0     = (empty, Just x0)
                | otherwise = unsafePerformIO $ createAndTrim' i $ \p -> go p x0 0
  where
    go a b c | a `seq` b `seq` c `seq` False = undefined
    go p x n = case f x of
                 Nothing      -> return (0, n, Nothing)
                 Just (w, x') | n == i    -> return (0, n, Just x)
                              | otherwise -> do poke p w
                                                go (p `plusPtr` 1) x' (n + 1)
```

Функция: `take`

Описание: возвращает восьмибитовую строку, представляющую собой первые n символов строки-источника. Сложность — $O(1)$.

Определение:

```
take :: Int -> ByteString -> ByteString
take n ps@(PS x s l) | n <= 0    = empty
                     | n >= l    = ps
                     | otherwise = PS x s n
```

Функция: `drop`

Описание: возвращает восьмибитовую строку, представляющую собой исходную строку без первых n символов строки-источника. Сложность — $O(1)$.

Определение:

```
drop  :: Int -> ByteString -> ByteString
drop n ps@(PS x s l) | n <= 0    = ps
                     | n >= l    = empty
                     | otherwise = PS x (s + n) (l - n)
```

Функция: `splitAt`

Описание: возвращает пару восьмибитовых строк, первым элементом которой является результат функции `take` на тех же входных параметрах, а вторым — результат функции `drop` соответственно. Сложность — $O(1)$.

Определение:

```
splitAt :: Int -> ByteString -> (ByteString, ByteString)
splitAt n ps@(PS x s l) | n <= 0    = (empty, ps)
                        | n >= l    = (ps, empty)
                        | otherwise = (PS x s n, PS x (s + n) (l - n))
```

Функция: `takeWhile`

Описание: возвращает подстроку заданной восьмибитовой строки, состоящую из тех символов в начале исходной строки, которые удовлетворяют входному предикату. Как только функция находит символ, который не удовлетворяет предикату, процесс выборки подстроки заканчивается.

Определение:

```
takeWhile :: (Word8 -> Bool) -> ByteString -> ByteString
takeWhile f ps = unsafeTake (findIndexOrEnd (not . f) ps) ps
```

Функция: `dropWhile`

Описание: возвращает подстроку заданной восьмибитовой строки, состоящую из тех символов в конце исходной строки, которые не удовлетворяют входному предикату. Как только функция находит символ, который не удовлетворяет предикату, процесс выборки подстроки заканчивается, результатом возвращается остаток входной строки.

Определение:

```
dropWhile :: (Word8 -> Bool) -> ByteString -> ByteString
dropWhile f ps = unsafeDrop (findIndexOrEnd (not . f) ps) ps
```

Функция: `span`

Описание: возвращает пару восьмибитовых строк, первым элементом которой является результат функции `takewhile` на тех же входных параметрах, а вторым — результат функции `dropWhile` соответственно.

Определение:

```
span :: (Word8 -> Bool) -> ByteString -> (ByteString, ByteString)
span p ps = break (not . p) ps
```

Функция: `spanEnd`

Описание: вариант функции `span`, который действует с конца заданной строки-источника.

Определение:

```
spanEnd :: (Word8 -> Bool) -> ByteString -> (ByteString, ByteString)
spanEnd  p ps = splitAt (findFromEndUntil (not . p) ps) ps
```

Функция: `break`
Описание: вариант функции `span`, в котором входной предикат обращён.
Определение:

```
break :: (Word8 -> Bool) -> ByteString -> (ByteString, ByteString)
break p ps = case findIndexOrEnd p ps of n -> (unsafeTake n ps, unsafeDrop n ps)
```

Функция: `breakEnd`
Описание: вариант функции `break`, который действует с конца заданной строки-источника.
Определение:

```
breakEnd :: (Word8 -> Bool) -> ByteString -> (ByteString, ByteString)
breakEnd  p ps = splitAt (findFromEndUntil p ps) ps
```

Функция: `group`
Описание: возвращает список восьмибитовых строк, являющихся подстроками входной строки, состоящих из последовательных одинаковых символов.
Определение:

```
group :: ByteString -> [ByteString]
group xs | null xs   = []
         | otherwise = ys : group zs
  where (ys, zs) = spanByte (unsafeHead xs) xs
```

Например, вызов `group "Mississippi"` вернёт список `["M", "i", "ss", "i", "ss", "i", "pp", "i"]`.

Функция: `groupBy`
Описание: более общий вариант функции `group`, в который передаётся функция для вычисления равенства (первым аргументом).
Определение:

```
groupBy :: (Word8 -> Word8 -> Bool) -> ByteString -> [ByteString]
groupBy k xs | null xs   = []
             | otherwise = unsafeTake n xs : groupBy k (unsafeDrop n xs)
```

```
  where
    n = 1 + findIndexOrEnd (not . k (unsafeHead xs)) (unsafeTail xs)
```

Функция: `inits`

Описание: возвращает все начальные подстроки заданной восьмибитовой строки. Сложность — $O(n)$, где n — длина строки.

Определение:

```
inits :: ByteString -> [ByteString]
inits (PS x s l) = [PS x s n | n <- [0..l]]
```

Функция: `tails`

Описание: возвращает все конечные подстроки заданной восьмибитовой строки. Сложность — $O(n)$, где n — длина строки.

Определение:

```
tails :: ByteString -> [ByteString]
tails p | null p    = [empty]
        | otherwise = p : tails (unsafeTail p)
```

Функция: `split`

Описание: разделяет входную восьмибитовую строку на подстроки при помощи заданного символа (сам символ никогда не входит в финальные строки). Сложность — $O(n)$, где n — длина строки.

Определение:

```
split :: Word8 -> ByteString -> [ByteString]
split _ (PS _ _ 0) = []
split w (PS x s l) = inlinePerformIO $
                     withForeignPtr x $
                     \p -> do let ptr = p `plusPtr` s
                              loop a | a `seq` False = undefined
                              loop n = let q = inlinePerformIO $
                                               memchr (ptr `plusPtr` n)
                                                      w (fromIntegral (l - n))
                                       in  if q == nullPtr
                                             then [PS x (s + n) (l - n)]
                                             else let i = q `minusPtr` ptr
                                                  in  PS x (s + n) (i - n):loop (i + 1)
                              return (loop 0)
```

Функция: `splitWith`

Описание: вариант функции `split`, в котором предикат для разбиения входной восьмибитовой строки на подстроки передаётся первым аргументом. Сложность — $O(n)$, где n — длина строки.

Определение:

```
splitWith :: (Word8 -> Bool) -> ByteString -> [ByteString]
splitWith _ (PS _ _ 0) = []
splitWith p ps = loop p ps
  where
    loop a b | a `seq` b `seq` False = undefined
    loop q qs = if null rest
                  then [chunk]
                  else chunk : loop q (unsafeTail rest)
      where
        (chunk, rest) = break q qs
```

Функция: `join`

Описание: принимает на вход строку и список строк, возвращает восьмибитовую строку, составленную из строк входного списка, перемеженных первым аргументом. Функция является обратной к функции `split`. Сложность — $O(n)$, где n — длина строки.

Определение:

```
join :: ByteString -> [ByteString] -> ByteString
join s = concat . (List.intersperse s)
```

Функция: `isPrefixOf`

Описание: возвращает значение `True`, если первый аргумент является началом второго. Сложность — $O(n)$, где n — длина меньшей строки.

Определение:

```
isPrefixOf :: ByteString -> ByteString -> Bool
isPrefixOf (PS x1 s1 l1) (PS x2 s2 l2)
  | l1 == 0   = True
  | l2 < l1   = False
  | otherwise = inlinePerformIO $ withForeignPtr x1 $ \p1 -> withForeignPtr x2 $
                \p2 -> do i <- memcmp (p1 `plusPtr` s1) (p2 `plusPtr` s2)
                                                        (fromIntegral l1)
                return $! i == 0
```

Функция: isSuffixOf

Описание: возвращает значение True, если первый аргумент является окончанием второго. Сложность — $O(n)$, где n — длина меньшей строки.

Определение:

```
isSuffixOf :: ByteString -> ByteString -> Bool
isSuffixOf (PS x1 s1 l1) (PS x2 s2 l2)
  | l1 == 0   = True
  | l2 < l1   = False
  | otherwise = inlinePerformIO $
                withForeignPtr x1 $
                \p1 -> withForeignPtr x2 $
                \p2 -> do i <- memcmp (p1 `plusPtr` s1)
                                      (p2 `plusPtr` s2 `plusPtr` (l2 - l1))
                                      (fromIntegral l1)
                          return $! i == 0
```

Функция: isSubstringOf

Описание: возвращает значение True, если первая строка является подстрокой второго аргумента.

Определение:

```
isSubstringOf :: ByteString -> ByteString -> Bool
isSubstringOf p s = not $ P.null $ findSubstrings p s
```

Функция: findSubstring

Описание: возвращает позицию первого вхождения первого аргумента во второй.

Определение:

```
findSubstring :: ByteString -> ByteString -> Maybe Int
findSubstring = (listToMaybe .) . findSubstrings
```

Функция: findSubstrings

Описание: возвращает список позиций всех вхождений первого аргумента в качестве подстроки во второй.

Определение:

```
findSubstrings :: ByteString -> ByteString -> [Int]
findSubstrings pat@(PS _ _ m) str@(PS _ _ n) = search 0 0

  where
    patc x = pat `unsafeIndex` x
    strc x = str `unsafeIndex` x
```

```
    kmpNext = listArray (0,m) (-1:kmpNextL pat (-1))
    kmpNextL p _ | null p = []
    kmpNextL p j = let j' = next (unsafeHead p) j + 1
                       ps = unsafeTail p
                       x = if not (null ps) && unsafeHead ps == patc j'
                             then kmpNext Array.! j'
                             else j'
                   in  x:kmpNextL ps j'
    search i j = match ++ rest
      where
        match = if j == m
                  then [(i - j)]
                  else []
        rest = if i == n
                 then []
                 else search (i+1) (next (strc i) j + 1)
    next c j | j >= 0 && (j == m || c /= patc j) = next c (kmpNext Array.! j)
             | otherwise = j
```

Функция: `elem`

Описание: возвращает значение `True`, если заданный символ входит в восьмибитовую строку. Сложность — $O(n)$, где n — длина строки.

Определение:

```
elem :: Word8 -> ByteString -> Bool
elem c ps = case elemIndex c ps of
              Nothing -> False
              _       -> True
```

Функция: `notElem`

Описание: возвращает значение `False`, если заданный символ не входит в восьмибитовую строку. Сложность — $O(n)$, где n — длина строки.

Определение:

```
notElem :: Word8 -> ByteString -> Bool
notElem c ps = not (elem c ps)
```

Функция: `find`

Описание: возвращает первый элемент восьмибитовой строки, который удовлетворяет заданному предикату. Сложность — $O(n)$, где n — длина строки.

Определение:

```
find :: (Word8 -> Bool) -> ByteString -> Maybe Word8
find f p = case findIndex f p of
             Just n -> Just (p `unsafeIndex` n)
             _      -> Nothing
```

Функция: `filter`

Описание: фильтрует входную восьмибитовую строку при помощи заданного предиката, оставляя в ней только те символы, которые этому предикату удовлетворяют. Сложность — $O(n)$, где n — длина строки.

Определение:

```
filter :: (Word8 -> Bool) -> ByteString -> ByteString
filter f = loopArr . loopFilter f
```

Функция: `index`

Описание: возвращает символ на заданной позиции в восьмибитовой строке (индексация символов начинается с 0. Сложность — $O(1)$.

Определение:

```
index :: ByteString -> Int -> Word8
index ps n
    | n < 0          = moduleError "index" ("negative index: " ++ show n)
    | n >= length ps = moduleError "index" ("index too large: " ++ show n ++ ",
                                            length = " ++ show (length ps))
    | otherwise      = ps `unsafeIndex` n
```

Функция: `elemIndex`

Описание: возвращает позицию первого элемента в восьмибитовой строке, который равен заданному. Сложность — $O(n)$, где n — длина строки.

Определение:

```
elemIndex :: Word8 -> ByteString -> Maybe Int
elemIndex c (PS x s l) = inlinePerformIO $
                         withForeignPtr x $
                         \p -> do let p' = p `plusPtr` s
                                  q <- memchr p' c (fromIntegral l)
                                  return $! if q == nullPtr
                                              then Nothing
                                              else Just $! q `minusPtr` p'
```

Функция: `elemIndices`

Описание: возвращает список позиций заданного символа в восьмибитовой строке. Сложность — $O(n)$, где n — длина строки.

Определение:

```
elemIndices :: Word8 -> ByteString -> [Int]
elemIndices w (PS x s l) = inlinePerformIO $
                           withForeignPtr x $
                           \p -> do let ptr = p `plusPtr` s
                                        loop a | a `seq` False = undefined
                                        loop n = let q = inlinePerformIO $
                                                         memchr (ptr `plusPtr` n)
                                                                w (fromIntegral (l - n))
                                                 in  if q == nullPtr
                                                       then []
                                                       else let i = q `minusPtr` ptr
                                                            in  i : loop (i + 1)
                                    return $! loop 0
```

Функция: `findIndex`

Описание: обобщённый вариант функции `elemIndex`, в котором поиск позиции осуществляется при помощи заданного предиката.

Определение:

```
findIndex :: (Word8 -> Bool) -> ByteString -> Maybe Int
findIndex k (PS x s l) = inlinePerformIO $
                         withForeignPtr x $
                         \f -> go (f `plusPtr` s) 0
  where
    go a b | a `seq` b `seq` False = undefined
    go ptr n | n >= l    = return Nothing
             | otherwise = do w <- peek ptr
                              if k w
                                then return (Just n)
                                else go (ptr `plusPtr` 1) (n + 1)
```

Функция: `findIndices`

Описание: обобщённый вариант функции `elemIndices`, в котором поиск позиции осуществляется при помощи заданного предиката.

Определение:

```
findIndices :: (Word8 -> Bool) -> ByteString -> [Int]
findIndices p ps = loop 0 ps
  where
    loop a b  | a `seq` b `seq` False = undefined
    loop n qs | null qs           = []
              | p (unsafeHead qs) = n : loop (n + 1) (unsafeTail qs)
              | otherwise         =     loop (n + 1) (unsafeTail qs)
```

Функция: `count`

Описание: возвращает количество вхождений заданного символа в восьмибитовую строку.

Определение:

```
count :: Word8 -> ByteString -> Int
count w (PS x s m) = inlinePerformIO $
                     withForeignPtr x $
                     \p -> fmap fromIntegral $
                     c_count (p `plusPtr` s) (fromIntegral m) w
```

Функция: `zip`

Описание: принимает на вход две восьмибитовые строки и возвращает список пар символов, находящихся на одинаковых позициях в каждой строке. Сложность — $O(n)$, где n — длина меньшей строки.

Определение:

```
zip :: ByteString -> ByteString -> [(Word8,Word8)]
zip ps qs | null ps || null qs = []
          | otherwise          = (unsafeHead ps, unsafeHead qs) :
                                   zip (unsafeTail ps) (unsafeTail qs)
```

Функция: `zipWith`

Описание: обобщение функции `zip`, которое преобразует пару восьмибитовых строк в список при помощи заданной функции, получающей на вход по символу из каждой строки. Сложность — $O(n)$, где n — длина меньшей строки.

Определение:

```
zipWith :: (Word8 -> Word8 -> a) -> ByteString -> ByteString -> [a]
zipWith f ps qs | null ps || null qs = []
                | otherwise          = f (unsafeHead ps) (unsafeHead qs) :
                                       zipWith f (unsafeTail ps) (unsafeTail qs)
```

Функция: `unzip`

Описание: преобразует список пар символов в пару восьмибитовых строк. По эффекту обратна функции `zip`. Сложность — $O(n)$, где n — длина большей строки.

Определение:

```
unzip :: [(Word8,Word8)] -> (ByteString,ByteString)
unzip ls = (pack (P.map fst ls), pack (P.map snd ls))
```

Функция: `sort`

Описание: осуществляет сортировку символов в восьмибитовой строке. Сложность — $O(n)$, где n — длина строки.

Определение:

```
sort :: ByteString -> ByteString
sort (PS input s l)
  = unsafeCreate l $
    \p -> allocaArray 256 $
    \arr -> do memset (castPtr arr) 0 (256 * fromIntegral (sizeOf (undefined :: CSize)))
               withForeignPtr input (\x -> countOccurrences arr (x `plusPtr` s) l)
               let go a b | a `seq` b `seq` False = undefined
                   go 256 _   = return ()
                   go i   ptr = do n <- peekElemOff arr i
                                   when (n /= 0) $
                                        memset ptr (fromIntegral i) n >> return ()
                   go (i + 1) (ptr `plusPtr` (fromIntegral n))
                   go 0 p
```

Функция: `sortBy`

Описание: обобщение функции `sort`, которое сортирует символы в восьмибайтовой строке при помощи заданного предиката сравнения. В текущей поставке модулей не реализовано.

Определение:

```
sortBy :: (Word8 -> Word8 -> Ordering) -> ByteString -> ByteString
sortBy f ps = undefined
```

Функция: packCString

Описание: создаёт восьмибитовую строку из строки типа `CString`. Сложность — $O(n)$, где n — длина строки.

Определение:

```
packCString :: CString -> ByteString
packCString cstr = unsafePerformIO $
                   do fp <- newForeignPtr_ (castPtr cstr)
                      l <- c_strlen cstr
                      return $! PS fp 0 (fromIntegral l)
```

Функция: packCStringLen

Описание: создаёт восьмибитовую строку из строки типа `CString` без использования функции `strlen`, а потому сложность — $O(1)$.

Определение:

```
packCStringLen :: CStringLen -> ByteString
packCStringLen (ptr, len) = unsafePerformIO $
                            do fp <- newForeignPtr_ (castPtr ptr)
                               return $! PS fp 0 (fromIntegral len)
```

Функция: packMallocCString

Описание: создаёт восьмибитовую строку из строки типа `CString`, которая была создана при помощи функции `malloc` (в языке C). Сложность — $O(n)$, где n — длина строки.

Определение:

```
packMallocCString :: CString -> ByteString
packMallocCString cstr = unsafePerformIO $
                         do fp <- newForeignFreePtr (castPtr cstr)
                            len <- c_strlen cstr
                            return $! PS fp 0 (fromIntegral len)
```

Функция: useAsCString

Описание: использование восьмибитовой строки в тех функциях, которые используют строки в стиле языка C (заканчивающиеся нулевым символом). Па-

мять от созданной строки после использования освобождается автоматически. Сложность — $O(n)$, где n — длина строки.
Определение:

```
useAsCString :: ByteString -> (CString -> IO a) -> IO a
useAsCString (PS ps s l) = bracket alloc (c_free.castPtr)
  where
    alloc = withForeignPtr ps $
            \p -> do buf <- c_malloc (fromIntegral l+1)
                     memcpy (castPtr buf) (castPtr p `plusPtr` s) (fromIntegral l)
                     poke (buf `plusPtr` l) (0 :: Word8)
                     return (castPtr buf)
```

Функция: `useAsCStringLen`
Описание: вариант функции `useAsCString`, которая имеет сложность $O(1)$ и используется тогда, когда известна длина строки.
Определение:

```
useAsCStringLen :: ByteString -> (CStringLen -> IO a) -> IO a
```

Функция определена в виде примитива.

Функция: `copy`
Описание: копирует восьмибитовую строку. Эта функция может использоваться в тех случаях, когда старая копия строки может быть подвергнута уборке сборщиком мусора. Например, если для работы получена какая-либо большая строка, из которой используется только часть. Сложность — $O(n)$, где n — длина строки.
Определение:

```
copy :: ByteString -> ByteString
copy (PS x s l) = unsafeCreate l $
                  \p -> withForeignPtr x $
                  \f -> memcpy p (f `plusPtr` s) (fromIntegral l)
```

Функция: `copyCString`
Описание: копирует строку типа `CString` в виде восьмибитовой строки. Функция используется тогда, когда известно, что исходная строка будет уничтожена во внешнем модуле. Сложность — $O(n)$, где n — длина строки.

Определение:

```
copyCString :: CString -> IO ByteString
copyCString cstr = do len <- c_strlen cstr
                      copyCStringLen (cstr, fromIntegral len)
```

Функция: copyCStringLen

Описание: вариант функции copyCString, который используется тогда, когда известна длина строки.

Определение:

```
copyCStringLen :: CStringLen -> IO ByteString
copyCStringLen (cstr, len) = create len $
                             \p -> memcpy p (castPtr cstr) (fromIntegral len)
```

Функция: getLine

Описание: считывает восьмибитовую строку из стандартного потока ввода.

Определение:

```
getLine :: IO ByteString
getLine = hGetLine stdin
```

Функция: getContents

Описание: считывает всё содержимое стандартного потока ввода в восьмибитовую строку.

Определение:

```
getContents :: IO ByteString
getContents = hGetContents stdin
```

Функция: putStr

Описание: записывает заданную восьмибитовую строку в стандартный поток вывода.

Определение:

```
putStr :: ByteString -> IO ()
putStr = hPut stdout
```

Функция: putStrLn

Описание: вариант функции putStr, записывающий в стандартный поток вывода после строки символ перевода строки.

Определение:

```
putStrLn :: ByteString -> IO ()
putStrLn = hPutStrLn stdout
```

Функция: `interact`

Описание: принимает на вход функцию, которой подаётся на вход всё содержимое стандартного потока ввода и чей результат записывается непосредственно в стандартный поток вывода.

Определение:

```
interact :: (ByteString -> ByteString) -> IO ()
interact transformer = putStr . transformer =<< getContents
```

Функция: `readFile`

Описание: считывает всё содержимое заданного файла в восьмибитовую строку. Эта функция более эффективна, нежели считывание символов файла в список символов `String`, а потом использование функции `pack`.

Определение:

```
readFile :: FilePath -> IO ByteString
readFile f = bracket (openBinaryFile f ReadMode)
                     hClose
                     (\h -> hFileSize h >>= hGet h . fromIntegral)
```

Функция: `writeFile`

Описание: записывает восьмибитовую строку в заданный файл, перезаписывая его содержимое.

Определение:

```
writeFile :: FilePath -> ByteString -> IO ()
writeFile f txt = bracket (openBinaryFile f WriteMode)
                          hClose
                          (\h -> hPut h txt)
```

Функция: `appendFile`

Описание: дописывает восьмибитовую строку в конец заданного файла.

Определение:

```
appendFile :: FilePath -> ByteString -> IO ()
appendFile f txt = bracket (openBinaryFile f AppendMode)
                           hClose
                           (\h -> hPut h txt)
```

Функция: hGetLine

Описание: считывает восьмибитовую строку из заданного источника.

Определение:

```
hGetLine :: Handle -> IO ByteString
hGetLine h = System.IO.hGetLine h >>= return . pack . P.map c2w
```

Функция: hGetContents

Описание: считывает всё содержимое заданного источника в восьмибитовую строку.

Определение:

```
hGetContents :: Handle -> IO ByteString
hGetContents h = do let start_size = 1024
                    p <- mallocArray start_size
                    i <- hGetBuf h p start_size
                    if i < start_size
                      then do p' <- reallocArray p i
                              fp <- newForeignFreePtr p'
                              return $! PS fp 0 i
                      else f p start_size
  where
    f p s = do let s' = 2 * s
               p' <- reallocArray p s'
               i  <- hGetBuf h (p' `plusPtr` s) s
               if i < s
                 then do let i' = s + i
                         p'' <- reallocArray p' i'
                         fp  <- newForeignFreePtr p''
                         return $! PS fp 0 i'
                 else f p' s'
```

Функция: hGet

Описание: считывает восьмибитовую строку непосредственно из заданного источника, не выделяя для этого буфер, как это делает функция hGetcontents.

Определение:

```
hGet :: Handle -> Int -> IO ByteString
hGet _ 0 = return empty
hGet h i = createAndTrim i $ \p -> hGetBuf h p i
```

Функция: hGetNonBlocking

Описание: вариант функции hGet, который не ожидает доступности данных, а возвращает только те данные, которые доступны на момент вызова функции.

Определение:

```
hGetNonBlocking :: Handle -> Int -> IO ByteString
hGetNonBlocking = hGet
```

Функция: hPut

Описание: записывает восьмибитовую строку в заданный источник.

Определение:

```
hPut :: Handle -> ByteString -> IO ()
hPut _ (PS _  _ 0) = return ()
hPut h (PS ps s l) = withForeignPtr ps $ \p-> hPutBuf h (p `plusPtr` s) l
```

Функция: hPutStr

Описание: синоним функции hPut для совместимости.

Определение:

```
hPutStr :: Handle -> ByteString -> IO ()
hPutStr = hPut
```

Функция: hPutStrLn

Описание: вариант функции hPut, который добавляет после записанной в заданный источник восьмибитовой строки символ перевода строки.

Определение:

```
hPutStrLn :: Handle -> ByteString -> IO ()
hPutStrLn h ps | length ps < 1024 = hPut h (ps `snoc` 0x0a)
               | otherwise        = hPut h ps >> hPut h (singleton (0x0a))
```

8.4.1. Модуль Base

Этот модуль является служебным и вряд ли должен использоваться программистом самостоятельно (хотя это возможно). В нём приводятся описания примитивов для работы с восьмибитовыми строками (алгебраические типы данных и функции). Использование этого модуля выглядит так:

```
import Data.ByteString.Base
```

При этом надо учесть, что модуль ByteString сам по себе уже импортирует рассматриваемый модуль, реимпортируя часть его функций (которая может быть полезна стороннему разработчику программного обеспечения), поэтому если модуль ByteString уже использован, импортировать модуль Base не надо.

В этом модуле практически все программные сущности описаны в виде примитивов, поэтому в нижеследующем описании этот факт отмечаться не будет. Если для функции указана только сигнатура, то такая функция примитивна. Описание этого модуля в данном справочнике приводится для того, чтобы у читателя было понимание, какие функции используются для работы с восьмибитовыми строками. Кроме того, многие из функций, которые описаны ниже, используются в функциях модуля ByteString.

Однако в первую очередь в модуле описывается два алгебраических типа данных, которые позволяют работать с восьмибитовыми строками.

Тип: ByteString

Описание: эффективное представление восьмибитовых строк, содержащих символы типа Word8.

Определение:

```
data ByteString = PS !(ForeignPtr Word8) !Int !Int
```

Для данного типа определены экземпляры следующих классов: Data, Eq, Monoid, Ord, Read, Show, Typeable.

Тип: LazyByteString

Описание: эффективное представление ленивых восьмибитовых строк, содержащих символы типа Word8.

Определение:

```
newtype LazyByteString = LPS [ByteString]
```

Для данного типа определены экземпляры следующих классов: Data, Read, Show, Typeable.

Функция: unsafeHead

Описание: вариант функции head для непустых строк, который пропускает проверку на пустоту, поэтому обязанностью программиста становится передача в эту функцию только непустых строк.

Определение:

```
unsafeHead :: ByteString -> Word8
unsafeHead (PS x s l) = assert (l > 0) $
                        inlinePerformIO $
                        withForeignPtr x $
                        \p -> peekByteOff p s
```

Функция: unsafeTail

Описание: вариант функции tail для непустых строк, который пропускает проверку на пустоту, поэтому обязанностью программиста становится передача в эту функцию только непустых строк.

Определение:

```
unsafeTail :: ByteString -> ByteString
unsafeTail (PS ps s l) = assert (l > 0) $ PS ps (s + 1) (l - 1)
```

Функция: unsafeIndex

Описание: вариант функции index, который пропускает проверку вхождения заданного индекса в диапазон индексов строки (начиная с 0). Программист должен сам следить за тем, чтобы передаваемый индекс был в правильном диапазоне.

Определение:

```
unsafeIndex :: ByteString -> Int -> Word8
unsafeIndex (PS x s l) i = assert (i >= 0 && i < l) $
                           inlinePerformIO $
                           withForeignPtr x $
                           \p -> peekByteOff p (s + i)
```

Функция: unsafeTake

Описание: вариант функции take, который пропускает проверку вхождения заданного числа в диапазон индексов строки. Программист должен сам следить за тем, чтобы передаваемое число символов, которое необходимо взять, лежало в интервале от 0 до длины строки.

Определение:

```
unsafeTake :: Int -> ByteString -> ByteString
unsafeTake n (PS x s l) = assert (0 <= n && n <= l) $ PS x s n
```

Функция: `unsafeDrop`

Описание: вариант функции `drop`, который пропускает проверку вхождения заданного числа в диапазон индексов строки. Программист должен сам следить за тем, чтобы передаваемое число символов, которое необходимо взять, лежало в интервале от 0 до длины строки.

Определение:

```
unsafeDrop  :: Int -> ByteString -> ByteString
unsafeDrop n (PS x s l) = assert (0 <= n && n <= l) $ PS x (s + n) (l - n)
```

Функция: `empty`

Описание: создаёт пустую восьмибитовую строку.

Определение:

```
empty :: ByteString
```

Функция: `create`

Описание: создаёт восьмибитовую строку заданного размера и заполняет её при помощи заданной функции.

Определение:

```
create :: Int -> (Ptr Word8 -> IO ()) -> IO ByteString
create l f = do fp <- mallocByteString l
                withForeignPtr fp $ \p -> f p
                return $! PS fp 0 l
```

Функция: `createAndTrim`

Описание: создаёт восьмибитовую строку размера, не более указанного, и заполняет её содержимым при помощи заданной функции. Функция `createAndTrim` является основным эффективным механизмом создания восьмибитовых строк в языке Haskell при работе со строками из языка C.

Определение:

```
createAndTrim :: Int -> (Ptr Word8 -> IO Int) -> IO ByteString
createAndTrim l f = do fp <- mallocByteString l
                       withForeignPtr fp $
                         \p -> do l' <- f p
                                  if assert (l' <= l) $ l' >= l
                                    then return $! PS fp 0 l
                                    else create l' $
                                         \p' -> memcpy p' p (fromIntegral l')
```

Функция: `createAndTrim'`

Описание: вариант функции `createAndTrim`, используемый в случаях, когда заданная функция для заполнения строки возвращает специальный результат.

Определение:

```
createAndTrim' :: Int -> (Ptr Word8 -> IO (Int, Int, a)) -> IO (ByteString, a)
createAndTrim' l f = do fp <- mallocByteString l
                        withForeignPtr fp $
                          \p -> do (off, l', res) <- f p
                                   if assert (l' <= l) $ l' >= l
                                     then return $! (PS fp 0 l, res)
                                     else do ps <- create l' $
                                                   \p' -> memcpy p' (p `plusPtr` off)
                                                                    (fromIntegral l')
                                             return $! (ps, res)
```

Функция: `mallocByteString`

Описание: обёртка над примитивной функцией для выделения памяти под восьмибитовую строку.

Определение:

```
mallocByteString :: Int -> IO (ForeignPtr a)
```

Функция: `unsafeCreate`

Описание: создаёт восьмибитовую строку внутри монады `IO`.

Определение:

```
unsafeCreate :: Int -> (Ptr Word8 -> IO ()) -> ByteString
unsafeCreate l f = unsafePerformIO (create l f)
```

Функция: unsafeUseAsCString

Описание: функция для использования восьмибитовой строки внутри функций, которые работают со строками в формате языка C. Возвращает результат работы такой функции в монаде IO.

Определение:

```
unsafeUseAsCString :: ByteString -> (CString -> IO a) -> IO a
unsafeUseAsCString (PS ps s _) ac = withForeignPtr ps $
                                    \p -> ac (castPtr p `plusPtr` s)
```

Функция: unsafeUseAsCStringLen

Описание: функция для использования восьмибитовой строки внутри функций, которые работают со строками в формате языка C (и принимают на вход данные типа CStringLen). Возвращает результат работы такой функции в монаде IO.

Определение:

```
unsafeUseAsCStringLen :: ByteString -> (CStringLen -> IO a) -> IO a
unsafeUseAsCStringLen (PS ps s l) f = withForeignPtr ps $
                                      \p -> f (castPtr p `plusPtr` s, l)
```

Функция: fromForeignPtr

Описание: конструирует восьмибитовую строку из указателя на внешнюю строку в формате языка C.

Определение:

```
fromForeignPtr :: ForeignPtr Word8 -> Int -> ByteString
fromForeignPtr fp l = PS fp 0 l
```

Функция: toForeignPtr

Описание: создаёт указатель на строку в формате языка C на основе заданной восьмибитовой строки.

Определение:

```
toForeignPtr :: ByteString -> (ForeignPtr Word8, Int, Int)
toForeignPtr (PS ps s l) = (ps, s, l)
```

Функция: packCStringFinalizer

Описание: создаёт строку при помощи заданных указателя на память в формате языка C, длины строки и действия в монаде IO, которое произведёт финаль-

ную обработку созданной строки. Данная функция доступна только в поставке компилятора GHC.
Определение:

```
packCStringFinalizer :: Ptr Word8 -> Int -> IO () -> IO ByteString
```

Функция: `packAddress`
Описание: упаковывает произвольную последовательность байт, оканчивающуюся нулевым символом, в восьмибитовую строку. Данная функция доступна только в поставке компилятора GHC.
Определение:

```
packAddress :: Addr# -> ByteString
```

Функция: `unsafePackAddress`
Описание: упаковывает заданное количество байт по требуемому адресу в восьмибитовую строку. Данная функция доступна только в поставке компилятора GHC.
Определение:

```
unsafePackAddress :: Int -> Addr# -> ByteString
```

Функция: `unsafeFinalize`
Описание: явно запускает функцию-финализатор, определённую для заданной восьмибитовой строки. Вызвать эту функцию можно только один раз, поскольку последующие вызовы могут вернуть некорректный адрес. Данная функция доступна только в поставке компилятора GHC.
Определение:

```
unsafeFinalize :: ByteString -> IO ()
```

Функция: `inlinePerformIO`
Описание: инлайновое выполнение функции `unsafePerformIO`.
Определение:

```
inlinePerformIO :: IO a -> a
inlinePerformIO = unsafePerformIO
```

Функция: `nullForeignPtr`
Описание: возвращает пустой указатель на внешнюю память в формате языка C.

Определение:

```
nullForeignPtr :: ForeignPtr Word8
```

Функция: `countOccurrences`

Описание: подсчитывает количество проявлений определённого байта по заданному адресу.

Определение:

```
countOccurrences :: (Storable a, Num a) => Ptr a -> Ptr Word8 -> Int -> IO ()
countOccurrences :: (Storable a, Num a) => Ptr a -> Ptr Word8 -> Int -> IO ()
countOccurrences a b c | a `seq` b `seq` c `seq` False = undefined
countOccurrences counts str l = go 0
 where
   go a | a `seq` False = undefined
   go i | i == l    = return ()
        | otherwise = do k <- fromIntegral `fmap` peekElemOff str i
                         x <- peekElemOff counts k
                         pokeElemOff counts k (x + 1)
                         go (i + 1)
```

Функция: `c-strlen`

Описание: обёртка над функцией `strlen` из языка C.

Определение:

```
c_strlen :: CString -> IO CSize
```

Функция: `c-malloc`

Описание: обёртка над функцией `malloc` из языка C.

Определение:

```
c_malloc :: CSize -> IO (Ptr Word8)
```

Функция: `c-free`

Описание: обёртка над функцией `free` из языка C.

Определение:

```
c_free :: Ptr Word8 -> IO ()
```

Функция: `c-free-finalizer`

Описание: обёртка над функцией `free_finalizer` из языка C.

Определение:

```
c_free_finalizer :: FunPtr (Ptr Word8 -> IO ())
```

Функция: memchr
Описание: обёртка над функцией memchr из языка C.
Определение:

```
memchr :: Ptr Word8 -> Word8 -> CSize -> IO (Ptr Word8)
```

Функция: memcmp
Описание: обёртка над функцией memcmp из языка C.
Определение:

```
memcmp :: Ptr Word8 -> Ptr Word8 -> CSize -> IO CInt
```

Функция: memcpy
Описание: обёртка над функцией memcpy из языка C.
Определение:

```
memcpy :: Ptr Word8 -> Ptr Word8 -> CSize -> IO ()
```

Функция: memmove
Описание: обёртка над функцией memmove из языка C.
Определение:

```
memmove :: Ptr Word8 -> Ptr Word8 -> CSize -> IO ()
```

Функция: memset
Описание: обёртка над функцией memset из языка C.
Определение:

```
memset :: Ptr Word8 -> Word8 -> CSize -> IO (Ptr Word8)
```

Функция: c-reverse
Описание: обёртка над функцией reverse из языка C.
Определение:

```
c_reverse :: Ptr Word8 -> Ptr Word8 -> CULong -> IO ()
```

Функция: `c-intersperse`
Описание: обёртка над функцией `intersperse` из языка C.
Определение:

```
c_intersperse :: Ptr Word8 -> Ptr Word8 -> CULong -> Word8 -> IO ()
```

Функция: `c-maximum`
Описание: обёртка над функцией `maximum` из языка C.
Определение:

```
c_maximum :: Ptr Word8 -> CULong -> IO Word8
```

Функция: `c-minimum`
Описание: обёртка над функцией `minimum` из языка C.
Определение:

```
c_minimum :: Ptr Word8 -> CULong -> IO Word8
```

Функция: `c-count`
Описание: обёртка над функцией `count` из языка C.
Определение:

```
c_count :: Ptr Word8 -> CULong -> Word8 -> IO CULong
```

Функция: `w2c`
Описание: конвертирует восьмибитовую строку, содержащую значения типа `Word8`, в строку, содержащую значения типа `Char`.
Определение:

```
w2c :: Word8 -> Char
w2c = chr . fromIntegral
```

Функция: `c2w`
Описание: конвертирует восьмибитовую строку, содержащую значения типа `Char`, в строку, содержащую значения типа `Word8`. Функция небезопасна, поскольку просто отсекает символы, до восьми битов. Является «квази»-обратной к функции `w2c`.

Определение:

```
c2w :: Char -> Word8
c2w = fromIntegral . ord
```

Функция: `isSpaceWord8`

Описание: возвращает значение `True`, если заданный символ типа `Word8` является «пробельным».

Определение:

```
isSpaceWord8 :: Word8 -> Bool
isSpaceWord8 w = case w of
                   0x20 -> True
                   0x0A -> True
                   0x09 -> True
                   0x0C -> True
                   0x0D -> True
                   0x0B -> True
                   0xA0 -> True
                   _    -> False
```

8.4.2. Модуль `Char8`

В модуле `Char8` определены все те же самые программные сущности, что и в модуле `ByteString`. Все типы данных, все функции имеют абсолютно те же самые наименования, их количество такое же (за исключением двух дополнительных функций, которые описываются ниже). Этот модуль полностью дублирует модуль `ByteString`, однако его предназначение несколько иное. В то время как в модуле `ByteString` для представления символов строки используется тип `Word8`, в рассматриваемом модуле `Char8` для тех же целей используется одноимённый тип `Char8`.

Так же, как и с модулем `ByteString`, при использовании рассматриваемого модуля его необходимо импортировать квалифицированно для избежания коллизии с уже определёнными программными сущностями из стандартного модуля `Prelude`. Поэтому импорт модуля выглядит следующим образом (как пример):

```
import qualified Data.ByteString.Char8 as C
```

В этом модуле также определены две дополнительные функции, используемые для создания восьмибитовых строк на низком уровне (на уровне физических адресов в памяти).

Функция: `packAddress`

Описание: создаёт восьмибитовую строку при помощи заданной адресом последовательности байт, оканчивающейся нулевым символом. Сложность — $O(n)$, где n — длина строки.

Определение:

```
packAddress :: Addr# -> ByteString
```

Функция определена в виде примитива.

Функция: `unsafePackAddress`

Описание: небезопасный способ создания восьмибитовой строки при помощи указания адреса в памяти, откуда брать информацию для заполнения строки, а также количества символов, которые необходимо использовать. Сложность — $O(1)$.

Определение:

```
unsafePackAddress :: Int -> Addr# -> ByteString
```

Функция определена в виде примитива.

8.4.3. Модуль `Lazy`

Модуль `Lazy` предоставляет эффективную реализацию ленивых восьмибитовых строк при помощи массивов, содержащих символы типа `Word8`. Такие ленивые восьмибитовые строки помогают создавать высокопроизводительные приложения, которые эффективны как с точки зрения занимаемой памяти (большие строки хранятся и используются по мере необходимости), так и с точки зрения скорости (в силу той же ленивости).

Некоторые функции работают несколько более эффективно, чем их строгие варианты (например, функции `concat`, `append`, `reverse` и `cons`). И другие функции из этого модуля работают на несколько процентов эффективнее, чем их строгие варианты. Но когда дело доходит до огромных массивов информации, либо при разработке приложения имеются жёсткие ограничения по используемой памяти, функции рассматриваемого модуля дают несравненный результат.

Так же, как и с другими модулями семейства ByteString, при использовании рассматриваемого модуля его необходимо импортировать квалифицированно для избежания коллизии с уже определёнными программными сущностями из стандартного модуля Prelude. Поэтому импорт модуля выглядит следующим образом (как пример):

```
import qualified Data.ByteString.Lazy as B
```

Модуль Char8

Наконец, последний модуль из семейства модулей для работы с восьмибитовыми строками — это модуль Char8, который является подчинённым модулю Lazy из пакета Data. По его наименованию и месту в иерархии модулей можно понять, что этот модуль предоставляет определения программных сущностей для работы с ленивыми восьмибитовыми строками, символы которых имеют тип Char8, а не Word8, как было в предыдущем модуле.

Естественно, что в этом модуле определены все те же самые программные сущности, что и в одноимённом модуле Char8, описанном в подразделе 8.4.2. (см. стр. 286). Это значит, что имена в этом модуле конфликтуют как с именами программных сущностей из стандартного модуля Prelude, так и из других модулей для работы с восьмибитовыми строками. Поэтому использование модуля всегда должно быть квалифицировано:

```
import qualified Data.ByteString.Lazy.Char8 as C
```

Для понимания того, какие типы данных и функции описаны в рассматриваемом модуле, необходимо ознакомиться с соответствующими программными сущностями из модулей Char8 и Lazy.

8.5. Модуль Char

Модуль Char отчасти дублирует описания программных сущностей для представления и работы с символами. Данный модуль создан в экспериментальном порядке в целях постепенной разгрузки стандартного модуля Prelude. Большинство определённых в модуле Char программных сущностей определены и в мо-

дуле Prelude. Однако есть и новые функции, поэтому ниже все программные сущности скрупулёзно перечисляются. Использование:

```
import Data.Char
```

Главный тип данных, описываемый в этом модуле, — Char. Этот тип используется для представления строковых символов.

Тип: Char

Описание: перечисление, чьи значения представляют собой символы в кодировке Unicode (стандарт ISO/IEC 10646).

Определение:

Тип определён в виде примитива.

Для того чтобы получить символ из соответствующего целого числа, представляющего собой код символа в таблице кодов, необходимо воспользоваться методом toEnum из класса Ord (см. стр. 121). Обратное преобразование производится при помощи метода fromEnum того же класса.

Для этого типа данных определены экземпляры следующих классов: Bounded, Data, Enum, Eq, IsChar, Ix, NFData, Ord, PrintfArg, Random, Read, Show, Storable, Typeable, IArray и MArray.

Тип: String

Описание: список символов, использующийся для представления строк.

Определение:

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

Все символы из кодировки Unicode разделены на буквы, цифры, знаки пунктуации, специальные символы, пробельные символы и т. д. Многие функции этого модуля предназначены для проверки и разделения символов на различные классы.

Функция: isControl

Описание: возвращает значение True, если заданный символ является управляющим. В противном случае возвращается значение False.

Определение:

```
isControl :: Char -> Bool
isControl c = c < ' ' || c >= '\DEL' && c <= '\x9f'
```

Функция: isSpace

Описание: возвращает значение True, если заданный символ является пробельным. В противном случае возвращается значение False.

Определение:

```
isSpace :: Char -> Bool
isSpace c = c == ' '  ||
            c == '\t' ||
            c == '\n' ||
            c == '\r' ||
            c == '\f' ||
            c == '\v'
```

Функция: isLower

Описание: возвращает значение True, если заданный символ является символом в нижнем регистре (строчная буква). В противном случае возвращается значение False.

Определение:

```
isLower :: Char -> Bool
isLower c = c >= 'a' && c <= 'z'
```

Функция: isUpper

Описание: возвращает значение True, если заданный символ является символом в верхнем регистре (заглавная буква). В противном случае возвращается значение False.

Определение:

```
isUpper :: Char -> Bool
isUpper c = c >= 'A' && c <= 'Z'
```

Функция: isAlpha

Описание: возвращает значение True, если заданный символ является буквой. В противном случае возвращается значение False.

Определение:

```
isAlpha :: Char -> Bool
isAlpha c = isUpper c || isLower c
```

Функция: `isAlphaNum`
Описание: возвращает значение `True`, если заданный символ является буквой или цифрой. В противном случае возвращается значение `False`.
Определение:

```
isAlphaNum :: Char -> Bool
isAlphaNum c = isAlpha c || isDigit c
```

Функция: `isPrint`
Описание: возвращает значение `True`, если заданный символ может быть напечатан. В противном случае возвращается значение `False`.
Определение:

```
isPrint :: Char -> Bool
```

Функция определена в виде примитива.

Функция: `isDigit`
Описание: возвращает значение `True`, если заданный символ является цифрой. В противном случае возвращается значение `False`.
Определение:

```
isDigit :: Char -> Bool
isDigit c = c >= '0' && c <= '9'
```

Функция: `isOctDigit`
Описание: возвращает значение `True`, если заданный символ является цифрой для представления восьмиричных чисел. В противном случае возвращается значение `False`.
Определение:

```
isOctDigit :: Char -> Bool isOctDigit c = c >= '0' && c <= '7'
```

Функция: `isHexDigit`
Описание: возвращает значение `True`, если заданный символ является цифрой для представления шестнадцатиричных чисел. В противном случае возвращается значение `False`.

Определение:

```
isHexDigit :: Char -> Bool
isHexDigit c = isDigit c ||
               c >= 'A' && c <= 'F' ||
               c >= 'a' && c <= 'f'
```

Функция: `isLetter`

Описание: синоним функции `isAlpha` для использования с произвольным диапазоном символов Unicode.

Определение:

```
isLetter :: Char -> Bool
isLetter c = case generalCategory c of
               UppercaseLetter -> True
               LowercaseLetter -> True
               TitlecaseLetter -> True
               ModifierLetter  -> True
               OtherLetter     -> True
               _               -> False
```

Функция: `isMark`

Описание: возвращает значение `True`, если заданный символ является знаком для представления ударений и других символов, комбинируемых с предыдущим. В противном случае возвращается значение `False`.

Определение:

```
isMark :: Char -> Bool
isMark :: Char -> Bool
isMark c = case generalCategory c of
             NonSpacingMark       -> True
             SpacingCombiningMark -> True
             EnclosingMark        -> True
             _                    -> False
```

Функция: `isNumber`

Описание: синоним функции `isDigit` для использования с произвольным диапазоном символов Unicode (возвращает значение `True` для произвольных цифр — арабских, римских и т. п.).

Определение:

```
isNumber :: Char -> Bool
isNumber c = case generalCategory c of
               DecimalNumber -> True
               LetterNumber  -> True
               OtherNumber   -> True
               _             -> False
```

Функция: `isPunctuation`

Описание: возвращает значение `True`, если заданный символ является знаком препинания. В противном случае возвращается значение `False`.

Определение:

```
isPunctuation :: Char -> Bool
isPunctuation c = case generalCategory c of
                    ConnectorPunctuation -> True
                    DashPunctuation      -> True
                    OpenPunctuation      -> True
                    ClosePunctuation     -> True
                    InitialQuote         -> True
                    FinalQuote           -> True
                    OtherPunctuation     -> True
                    _                    -> False
```

Функция: `isSymbol`

Описание: возвращает значение `True`, если заданный символ является математическим символом, знаком валюты и т. д. В противном случае возвращается значение `False`.

Определение:

```
isSymbol :: Char -> Bool
isSymbol c = case generalCategory c of
               MathSymbol     -> True
               CurrencySymbol -> True
               ModifierSymbol -> True
               OtherSymbol    -> True
               _              -> False
```

Функция: `isSeparator`

Описание: возвращает значение `True`, если заданный символ является разделительным. В противном случае возвращается значение `False`.

Определение:

```
isSeparator :: Char -> Bool
isSeparator c = case generalCategory c of
                  Space              -> True
                  LineSeparator      -> True
                  ParagraphSeparator -> True
                  _                  -> False
```

Функция: `isAscii`

Описание: возвращает значение `True`, если заданный символ находится в диапазоне символов ASCII. В противном случае возвращается значение `False`.

Определение:

```
isAscii :: Char -> Bool
isAscii c = c < '\x80'
```

Функция: `isLatin1`

Описание: возвращает значение `True`, если заданный символ находится в диапазоне символов Latin1. В противном случае возвращается значение `False`.

Определение:

```
isLatin1 :: Char -> Bool
isLatin1 c = c <= '\xff'
```

Функция: `isAsciiUpper`

Описание: возвращает значение `True`, если заданный символ находится в диапазоне символов ASCII и является заглавным (верхний регистр). В противном случае возвращается значение `False`.

Определение:

```
isAsciiUpper :: Char -> Bool
isAsciiUpper c = c >= 'A' && c <= 'Z'
```

Функция: `isAsciiLower`

Описание: возвращает значение `True`, если заданный символ находится в диапазоне символов ASCII и является строчным (нижний регистр). В противном случае возвращается значение `False`.

Определение:

```
isAsciiLower :: Char -> Bool
isAsciiLower c = c >= 'a' && c <= 'z'
```

Для описания категории символов в кодировке Unicode имеется специальное перечисление. Каждый символ принадлежит ровно одной категории. В некоторых функциях, приведённых ранее, как раз и используется это перечисление.

Тип: `GeneralCategory`
Описание: представляет категории символов Unicode, как это определено стандартом.
Определение:

```
data GeneralCategory
  = UppercaseLetter      | LowercaseLetter
  | TitlecaseLetter      | ModifierLetter
  | OtherLetter          | NonSpacingMark
  | SpacingCombiningMark | EnclosingMark
  | DecimalNumber        | LetterNumber
  | OtherNumber          | ConnectorPunctuation
  | DashPunctuation      | OpenPunctuation
  | ClosePunctuation     | InitialQuote
  | FinalQuote           | OtherPunctuation
  | MathSymbol           | CurrencySymbol
  | ModifierSymbol       | OtherSymbol
  | Space                | LineSeparator
  | ParagraphSeparator   | Control
  | Format               | Surrogate
  | PrivateUse           | NotAssigned
  deriving (Eq, Ord, Enum, Read, Show, Bounded, Ix)
```

Как видно из определения, для этого перечисления автоматически определяются экземпляры следующих классов: `Bounded`, `Enum`, `Eq`, `Ix`, `Ord`, `Read` и `Show`.

Сами категории символов могут быть поняты из наименований конструкторов типа `GeneralCategory`. Например, конструктор `CurrencySymbol` отвечает за категорию символов, представляющих собой знаки валют.

Функция: `generalCategory`
Описание: возвращает категорию Unicode для заданного символа.

Определение:

```
generalCategory :: Char -> GeneralCategory
```

Функция определена в виде примитива.

Функция: `toUpper`

Описание: возвращает соответствующий заданному символу символ в верхнем регистре (например, для символа `'a'` возвращается символ `'A'`).

Определение:

```
toUpper :: Char -> Char
```

Функция определена в виде примитива.

Функция: `toLower`

Описание: возвращает соответствующий заданному символу символ в нижнем регистре (например, для символа `'Z'` возвращается символ `'z'`).

Определение:

```
toLower :: Char -> Char
```

Функция определена в виде примитива.

Функция: `toTitle`

Описание: возвращает соответствующий заданному символу символ в виде, используемом для заголовков. Эта функция возвращает новые символы для небольшого числа лигатур, для символов в нижнем регистре возвращаются соответствующие символы в верхнем регистре, остальные символы возвращаются неизменёнными.

Определение:

```
toTitle :: Char -> Char
```

Функция определена в виде примитива.

Функция: `digitToInt`

Описание: возвращает числовое представление заданного символа в восьмиричной, десятичной или шестнадцатиричной системах счисления. Для остальных символов функция возвращает ошибку.

Определение:

```
digitToInt :: Char -> Int
digitToInt c | isDigit c            = ord c - ord '0'
             | c >= 'a' && c <= 'f' = ord c - ord 'a' + 10
             | c >= 'A' && c <= 'F' = ord c - ord 'A' + 10
             | otherwise            = error ("Char.digitToInt: not a digit " ++ show c)
```

Функция: `intToDigit`

Описание: обратная функция к функции `digitToInt`. Возвращает символ для заданной восьмиричной, десятичной или шестнадцатиричной цифры. Если заданное значение не является цифрой, функция возвращает ошибку.

Определение:

```
intToDigit :: Int -> Char

intToDigit i | i >= 0  && i <=  9 = toEnum (fromEnum '0' + i)
             | i >= 10 && i <= 15 = toEnum (fromEnum 'a' + i - 10)
             | otherwise          = error "Char.intToDigit: not a digit"
```

Функция: `ord`

Описание: синоним метода `fromEnum` из класса `Ord` (см. стр. 121). Возвращает код заданного символа.

Определение:

```
ord :: Char -> Int
ord = fromEnum
```

Функция: `chr`

Описание: синоним метода `toEnum` из класса `Ord` (см. стр. 121). Возвращает символ для заданного кода.

Определение:

```
chr :: Int -> Char
chr = toEnum
```

Функции `showLitChar`, `lexLitChar`, `readLitChar` на текущий момент реимпортируются из модуля `Prelude`. См. соответственно стр. 161, стр. 143 и стр. 155.

8.6. Модуль Complex

Модуль Complex предлагает разработчику программного обеспечения алгебраический тип данных и небольшой набор функций для представления и работы с комплексными числами. Использование модуля:

```
import Data.Complex
```

Тип: Complex
Описание: тип для представления комплексных чисел в ортогональной форме. Для получения формы в полярной системе координат используются специальные преобразующие функции.
Определение:

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

Как видно из определения, для этого типа автоматически строятся экземпляры классов Eq, Read и Show. В модуле дополнительно определены экземпляры и таких типов: Typeable1, Floating, Fractional, NFData и Num.

Функция: realPart
Описание: возвращает действительную часть комплексного числа.
Определение:

```
realPart :: RealFloat a => Complex a -> a
realPart (x :+ _) =  x
```

Функция: imagPart
Описание: возвращает мнимую часть комплексного числа.
Определение:

```
imagPart :: RealFloat a => Complex a -> a
imagPart (_ :+ y) =  y
```

Функция: mkPolar
Описание: формирует ортогональное комплексное число из компонентов в полярной системе координат (модуля и фазы).

Определение:

```
mkPolar :: (RealFloat a) => a -> a -> Complex a
mkPolar r theta = r * cos theta :+ r * sin theta
```

Функция: `cis`

Описание: формирует ортогональное комплексное число из компонентов в полярной системе координат, где модуль равен единице, а фаза задана.

Определение:

```
cis :: (RealFloat a) => a -> Complex a
cis theta = cos theta :+ sin theta
```

Функция: `polar`

Описание: возвращает компоненты заданного комплексного числа в полярной системе координат в виде пары (*модуль*, *фаза*).

Определение:

```
polar :: RealFloat a => Complex a -> (a, a)
polar z = (magnitude z, phase z)
```

Функция: `magnitude`

Описание: возвращает модуль заданного комплексного числа.

Определение:

```
magnitude :: (RealFloat a) => Complex a -> a
magnitude (x :+y ) =  scaleFloat k (sqrt ((scaleFloat mk x)^(2::Int) +
                                          (scaleFloat mk y)^(2::Int)))
  where
    k  = max (exponent x) (exponent y)
    mk = - k
```

Функция: `phase`

Описание: возвращает фазу заданного комплексного числа.

Определение:

```
phase :: (RealFloat a) => Complex a -> a
phase (0 :+ 0) = 0
phase (x :+ y) = atan2 y x
```

Функция: `conjugate`

Описание: возвращает комплексное число, сопряжённое заданному.

Определение:

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

8.7. Модуль Dynamic

Модуль Dynamic описывает базовый интерфейс для представления и работы с динамическими типами. Также в этом модуле определены функции для преобразования значений произвольных типов в динамические значения и обратно. Использование:

```
import Data.Dynamic
```

Тип: Dynamic
Описание: значение динамического типа представляет собой объект, к которому приписан его тип.
Определение:

Тип определён в виде примитива.

Этот тип данных может представлять только мономорфное значение. Попытка создать динамический тип их полиморфного выражения приведёт к неопределённости (см. ниже описание функции toDyn).

Для этого типа данных определены экземпляры классов Show и Typeable. Вывод на экран величин с динамическими типами очень полезен и удобен для отладки программ.

Функция: toDyn
Описание: преобразует значение произвольного типа в динамическое значение. Данная функция преобразует только мономорфные значения (и это гарантирует ограничение Typeable a =>). Чтобы сконвертировать полиморфное значение, его необходимо «мономорфировать» перед передачей в эту функцию (например, вызов toDyn (id :: Int -> Int)).
Определение:

```
toDyn :: Typeable a => a -> Dynamic
toDyn v = Dynamic (typeOf v) (unsafeCoerce v)
```

Функция: `fromDyn`

Описание: преобразует значение динамического типа назад в значение произвольного типа языка Haskell. Если такое преобразование возможно, то функция возвращает преобразованное значение первого аргумента. Если нет, возвращает второй аргумент.

Определение:

```
fromDyn :: Typeable a => Dynamic -> a -> a
fromDyn (Dynamic t v) def | typeOf def == t = unsafeCoerce v
                          | otherwise       = def
```

Функция: `fromDynamic`

Описание: преобразует значение динамического типа назад в значение произвольного типа языка Haskell. Если такое преобразование возможно, то функция возвращает преобразованное значение, обёрнутое в конструктор `Just` типа `Maybe` (см. стр. 109). Если нет, возвращает значение `Nothing`.

Определение:

```
fromDynamic :: Typeable a => Dynamic -> Maybe a
fromDynamic (Dynamic t v) = case unsafeCoerce v of
                              r | t == typeOf r -> Just r
                                | otherwise     -> Nothing
```

Функция: `dynApply`

Описание: применяет функцию к значениям динамического типа. Если такое применение возможно, то возвращает результат функции также динамического типа, но обёрнутый в конструктор `Just` типа `Maybe` (см. стр. 109). Если применение невозможно, возвращает значение `Nothing`.

Определение:

```
dynApply :: Dynamic -> Dynamic -> Maybe Dynamic
dynApply (Dynamic t1 f) (Dynamic t2 x)
  = case funResultTy t1 t2 of
      Just t3 -> Just (Dynamic t3 ((unsafeCoerce f) x))
      Nothing -> Nothing
```

Функция: `dynApp`

Описание: применяет функцию к значениям динамического типа. Если такое применение возможно, то возвращает результат функции также динамического типа. Если применение невозможно, возвращает ошибку.

Определение:

```
dynApp :: Dynamic -> Dynamic -> Dynamic
dynApp f x = case dynApply f x of
               Just r -> r
               Nothing -> error ("Type error in dynamic application.\n" ++
                                 "Can't apply function " ++ show f ++
                                 " to argument " ++ show x)
```

Функция: dynTypeRep
Описание: возвращает описание типа из значения динамического типа.
Определение:

```
dynTypeRep :: Dynamic -> TypeRep
dynTypeRep (Dynamic tr _) = tr
```

8.8. Модуль Either

В модуле Either дублируются описания типа Either и функции для его обработки. Данный модуль создан в экспериментальном порядке в целях постепенной разгрузки стандартного модуля Prelude. Все определённые в модуле Either программные сущности определены и в модуле Prelude. Использование:

```
import Data.Either
```

Соответственно, в рассматриваемый модуль вынесены определения: алгебраического типа данных Either (см. стр. 109) и функции either (см. стр. 132).

Также в модуле Either определены экземпляры типа Either для следующих классов: Typeable1, Functor, Data, Eq, Ord, Read и Show.

Необходимо отметить, что по соглашению тип Either используется для представления величин, которые могут нести в себе описание ошибочной ситуации. Конструктор Left этого типа используется для представления ошибок, конструктор Right — для представления обычных значений соответственно (здесь имеется дополнительное мнемоническое правило: слово *right* с английского языка можно перевести как «правый» и как «правильный»).

8.9. Модуль Eq

Модуль Eq также предназначен для разгрузки стандартного модуля Prelude. В него вынесено определение класса Eq (см. стр. 117) и определения нескольких десятков экземпляров для этого класса. Использование:

```
import Data.Eq
```

Класс Eq определяет класс типов, в которых имеет смысл отношение сравнения. Соответственно, определены два метода: (==) (равенство значений) и (/=) (неравенство значений). Для экземпляра можно определять либо один метод, либо другой, либо оба вместе. Все базовые типы данных являются экземплярами этого класса. Более того, для произвольного алгебраического типа данных можно автоматически построить экземпляр этого класса, если каждый из компонентов типа является экземпляром класса Eq.

Соответственно, в рассматриваемом модуле определены экземпляры этого класса для следующих типов: All, Any, ArithException, Array, ArrayException, AsyncException, Bool, BufferMode, BufferState, ByteString, CCc, CChar, CClock, CDev, CDouble, CFloat, CGid, CIno, CInt, CIntMax, CIntPtr, CLDouble, CLLong, CLong, CMode, CNlink, COff, Complex, CPid, CPtrdiff, CRLim, CSChar, CShort, CSigAtomic, CSize, CSpeed, CSsize, CTcflag, CTime, CUChar, CUInt, CUIntMax, CUIntPtr, CULLong, CULong, CUShort, CUid, CWchar, CalendarTime, Char, ClockTime, Constr, ConstrRep, DataRep, Day, Double, Either, Errno, Exception, ExitCode, FDType, Fd, Fixed, Fixity, Float, ForeignPtr, FunPtr, GeneralCategory, Handle, HandlePosn, HashData, IOArray, IOErrorType, IOException, IOMode, IORef, Inserts, Int, Int16, Int32, Int64, Int8, IntMap, IntPtr, IntSet, Integer, Key, KeyPr, Lexeme, Map, Maybe, Month, MVar, Ordering, PackedString, Permissions, Product, Ptr, Ratio, SeekMode, Seq, Set, StableName, StablePtr, STArray STRef, Sum, ThreadId, TimeDiff, TimeLocale, Tree, TVar, TyCon, TypeRep, UArray, Unique, Version, ViewL, ViewR, Word, Word16, Word32, Word64, Word8, WordPtr и [].

Кроме того, в этом модуле также определены экземпляры класса Eq для кортежей размером от 0 до 14.

8.10. Модуль Fixed

Модуль Fixed определяет тип данных для выполнения арифметических действий над значениями фиксированной точности. В модуле описывается класс, тип данных и несколько синонимов типов для выполнения таких операций. Кроме того, имеется несколько обобщённых функций для выполнения арифметических действий. Использование:

```
import Data.Fixed
```

Тип: Fixed
Описание: тип для представления значений, над которыми можно производить арифметические операции с фиксированной точностью.
Определение:

```
newtype Fixed a = MkFixed Integer
  deriving (Eq, Ord)
```

Для этого типа определены экземпляры следующих классов: Enum, Eq, Fractional, Num, Ord, Real, RealFrac и Show.

Класс: HasResolution
Описание: интерфейс для тех типов, которые могут использоваться в операциях с фиксированной точностью. Единственный метод resolution используется для преобразования к типу Fixed, который является изоморфным типу Integer.
Определение:

```
class HasResolution a where
  resolution :: a -> Integer
```

Тип: E6
Описание: тип для проведения арифметических операций с точностью 10^6.
Определение:

```
data E6 = E6
```

Тип: E12
Описание: тип для проведения арифметических операций с точностью 10^{12}.
Определение:

```
data E12 = E12
```

Типы E6 и E12 имеют определённые экземпляры класса HasResolution и используются для проведения вычислений с точностью до 6 и до 12 знаков.

Функция: div'

Описание: обобщение функции div (см. стр. 130) для произвольного экземпляра класса Real.

Определение:

```
div' :: (Real a, Integral b) => a -> a -> b
div' n d = floor ((toRational n) / (toRational d))
```

Функция: mod'

Описание: обобщение функции mod (см. стр. 147) для произвольного экземпляра класса Real.

Определение:

```
mod' :: (Real a) => a -> a -> a
mod' n d = n - (fromInteger f) * d
  where
    f = div' n d
```

Функция: divMod'

Описание: обобщение функции divMod (см. стр. 119) для произвольного экземпляра класса Real.

Определение:

```
divMod' :: (Real a, Integral b) => a -> a -> (b, a)
divMod' n d = (f, n - (fromIntegral f) * d)
  where
    f = div' n d
```

Остаётся отметить, что за этот модуль в поставке языка Haskell отвечает Э. Якели, с которым можно связаться по адресу электронной почты ashley@semantic.org.

8.11. Модуль Foldable

В модуле Foldable собраны определения программных сущностей языка Haskell, которые определяют общий интерфейс и утилитарные функции для структур данных, которые могут быть «свёрнуты» в единственное значение.

Другими словами, в этом модуле определяется класс для представления типов, на которые можно обобщить функции foldl (см. стр. 253) и foldr (см. стр. 254). В теории категорий такой интерфейс называется катаморфизмом. Использование:

```
import Data.Foldable
```

Многие функции из этого модуля имеют те же самые наименования, что и функции из стандартного модуля Prelude, а также модулей Monad (см. раздел 7.5.) и List (см. раздел 8.19.), поэтому обычно этот модуль импортируется квалифицированно, либо одноимённые функции из указанных модулей скрываются при импорте.

Главная программная сущность, определённая в рассматриваемом модуле, — класс Foldable, который и определяет интерфейс к типам, чьи значения могут быть свёрнуты.

Класс: Foldable

Описание: интерфейс к типам данных, значения которых могут быть свёрнуты к атомарному значению. Из всего набора методов можно определить либо метод foldMap, либо метод foldr, остальные методы выражены через указанные.

Определение:

```
class Foldable t where
  fold :: Monoid m => t m -> m
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr :: (a -> b -> b) -> b -> t a -> b
  foldl :: (a -> b -> a) -> a -> t b -> a
  foldr1 :: (a -> a -> a) -> t a -> a
  foldl1 :: (a -> a -> a) -> t a -> a
```

Метод fold комбинирует элементы типа данных при помощи операций класса Monoid (см. стр. 385). С другой стороны, метод foldMap сначала преобразует тип данных в моноид, а лишь затем производит свёртку. В этом процессе главное, чтобы экземпляры класса Monoid в действительности удовлетворяли теоретико-категорным законам для моноидов.

Методы foldl и foldr представляют собой левоассоциативную и правоассоциативную свёртку типов соответственно. Также и методы foldl1 и foldr1 являются вариантами методов foldl и foldr, для которых нет надобности ука-

зывать начальное значение для свёртки. В качестве него используется начальный элемент сворачиваемого значения.

Для этого класса определены экземпляры следующих типов: Array, Digit, Elem, FingerTree, IntMap, Maybe, Map, Node, Seq, Set, Tree, ViewL, ViewR и [].

Функция: foldr'

Описание: строгий вариант правоассоциативной свёртки foldr.

Определение:

```
foldr' :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr' f z xs = foldl f' id xs z
  where
    f' k x z = k $! f x z
```

Функция: foldl'

Описание: строгий вариант левоассоциативной свёртки foldl.

Определение:

```
foldl' :: Foldable t => (a -> b -> a) -> a -> t b -> a
foldl' f z xs = foldr f' id xs z
  where
    f' x k z = k $! f z x
```

Функция: foldrM

Описание: монадический вариант правоассоциативной свёртки foldr.

Определение:

```
foldrM :: (Foldable t, Monad m) => (a -> b -> m b) -> b -> t a -> m b
foldrM f z xs = foldl f' return xs z
  where
    f' k x z = f x z >>= k
```

Функция: foldlM

Описание: монадический вариант левоассоциативной свёртки foldl.

Определение:

```
foldlM :: (Foldable t, Monad m) => (a -> b -> m a) -> a -> t b -> m a
foldlM f z xs = foldr f' return xs z
  where
    f' x k z = f z x >>= k
```

Функция: traverse_

Описание: проецирует каждый элемент сворачиваемого значения в действие, выполняет полученные действия слева направо и игнорирует результаты (важны побочные эффекты действий).

Определение:

```
traverse_ :: (Foldable t, Applicative f) => (a -> f b) -> t a -> f ()
traverse_ f = foldr ((*>) . f) (pure ())
```

Функция: for_

Описание: вариант функции traverse_, у которого порядок аргументов обратный.

Определение:

```
for_ :: (Foldable t, Applicative f) => t a -> (a -> f b) -> f ()
for_ = flip traverse_
```

Функция: sequenceA_

Описание: выполняет каждое действие в сворачиваемой структуре слева направо и игнорирует результаты (важны побочные эффекты, а не результат).

Определение:

```
sequenceA_ :: (Foldable t, Applicative f) => t (f a) -> f ()
sequenceA_ = foldr (*>) (pure ())
```

Функция: asum

Описание: обобщение функции concat (см. стр. 309) для набора произвольных действий.

Определение:

```
asum :: (Foldable t, Alternative f) => t (f a) -> f a
asum = foldr (<|>) empty
```

Функция: mapM_

Описание: проецирует каждый элемент сворачиваемой структуры данных на монадическое действие, выполняет действия слева направо и игнорирует конечный результат (важны лишь побочные эффекты монады).

Определение:

```
mapM_ :: (Foldable t, Monad m) => (a -> m b) -> t a -> m ()
mapM_ f = foldr ((>>) . f) (return ())
```

Функция: forM-

Описание: вариант функции mapM_, у которого аргументы идут в обратном порядке.

Определение:

```
forM_ :: (Foldable t, Monad m) => t a -> (a -> m b) -> m ()
forM_ = flip mapM_
```

Функция: sequence-

Описание: выполняет каждое действие в монадической структуре слева направо и игнорирует возвращаемые результаты (важны лишь побочные эффекты, предоставляемые монадой).

Определение:

```
sequence_ :: (Foldable t, Monad m) => t (m a) -> m ()
sequence_ = foldr (>>) (return ())
```

Функция: msum

Описание: обобщение функции concat (см. стр. 309) для набора произвольных монадических действий.

Определение:

```
msum :: (Foldable t, MonadPlus m) => t (m a) -> m a
msum = foldr mplus mzero
```

Функция: toList

Описание: преобразует произвольную структуру в список.

Определение:

```
toList :: Foldable t => t a -> [a]
toList = foldr (:) []
```

Функция: concat

Описание: конкатенация всех списков в сворачиваемой структуре в один список.

Определение:

```
concat :: Foldable t => t [a] -> [a]
concat = fold
```

Функция: `concatMap`

Описание: применяет заданную функцию к каждому элементу в контейнере, после чего конкатенирует полученные результаты в один список.

Определение:

```
concatMap :: Foldable t => (a -> [b]) -> t a -> [b]
concatMap = foldMap
```

Функция: `and`

Описание: возвращает логическое произведение (операция «И») для всех значений типа `Bool` в сворачиваемой структуре данных. Для того чтобы функция вернула результат `True`, структура должна быть конечной. Для результата `False` структура может быть бесконечной, но хотя бы одно значение `False` должно находиться на конечной позиции.

Определение:

```
and :: Foldable t => t Bool -> Bool
and = getAll . foldMap All
```

Функция: `or`

Описание: возвращает логическую сумму (операция «ИЛИ») для всех значений типа `Bool` в сворачиваемой структуре данных. Для того чтобы функция вернула результат `False`, структура должна быть конечной. Для результата `True` структура может быть бесконечной, но хотя бы одно значение `True` должно находиться на конечной позиции.

Определение:

```
or :: Foldable t => t Bool -> Bool
or = getAny . foldMap Any
```

Функция: `any`

Описание: определяет, существует ли в сворачиваемой структуре данных хотя бы одно значение, которое удовлетворяет заданному предикату.

Определение:

```
any :: Foldable t => (a -> Bool) -> t a -> Bool
any p = getAny . foldMap (Any . p)
```

Функция: `all`

Описание: определяет, все ли значения в сворачиваемой структуре данных удовлетворяют заданному предикату.

Определение:

```
all :: Foldable t => (a -> Bool) -> t a -> Bool
all p = getAll . foldMap (All . p)
```

Функция: `sum`

Описание: возвращает сумму элементов в сворачиваемой структуре.

Определение:

```
sum :: (Foldable t, Num a) => t a -> a
sum = getSum . foldMap Sum
```

Функция: `product`

Описание: возвращает произведение элементов в сворачиваемой структуре данных.

Определение:

```
product :: (Foldable t, Num a) => t a -> a
product = getProduct . foldMap Product
```

Функция: `maximum`

Описание: возвращает максимальный элемент в сворачиваемой непустой структуре данных.

Определение:

```
maximum :: (Foldable t, Ord a) => t a -> a
maximum = foldr1 max
```

Функция: `maximumBy`

Описание: обобщение функции `maximum`, в котором сравнение производится передаваемой в качестве первого аргумента функцией.

Определение:

```
maximumBy :: Foldable t => (a -> a -> Ordering) -> t a -> a
maximumBy cmp = foldr1 max'
  where
    max' x y = case cmp x y of
                 GT -> x
                 _  -> y
```

Функция: `minimum`

Описание: возвращает минимальный элемент в сворачиваемой непустой структуре данных.

Определение:

```
minimum :: (Foldable t, Ord a) => t a -> a
minimum = foldr1 min
```

Функция: `minimumBy`

Описание: обобщение функции `minimum`, в котором сравнение производится передаваемой в качестве первого аргумента функцией.

Определение:

```
minimumBy :: Foldable t => (a -> a -> Ordering) -> t a -> a
minimumBy cmp = foldr1 min'
  where
    min' x y = case cmp x y of
                 GT -> y
                 _  -> x
```

Функция: `elem`

Описание: возвращает значение `True`, если заданное значение содержится в сворачиваемой структуре данных.

Определение:

```
elem :: (Foldable t, Eq a) => a -> t a -> Bool
elem = any . (==)
```

Функция: `notElem`

Описание: обращение предиката `elem`.

Определение:

```
notElem :: (Foldable t, Eq a) => a -> t a -> Bool
notElem x = not . elem x
```

Функция: `find`
Описание: возвращает из сворачиваемой структуры данных первый элемент слева, который удовлетворяет заданному предикату. Если ничего не найдено, возвращается значение `Nothing`.
Определение:

```
find :: Foldable t => (a -> Bool) -> t a -> Maybe a
find p = listToMaybe . concatMap (\x -> if p x
                                        then [x]
                                        else [])
```

Данный модуль является обновлённой и более совершенной версией модуля `FunctorM`, который в свою очередь объявлен устаревшим и выводится из состава стандартных библиотек языка Haskell. Совместно с модулем `Traversable` (см. раздел 8.28.) этот модуль предоставляет всю функциональность (и даже больше), которую предоставлял модуль `FunctorM`.

Остаётся отметить, что за этот модуль в поставке языка Haskell отвечает P. Патерсон, с которым можно связаться по адресу электронной почты `ross@soi.city.ac.uk`.

8.12. Модуль `Graph`

В модуле `Graph` определены функции для алгоритмов работы с графами, которые описаны в работе [11]. Использование модуля:

```
import Data.Graph
```

Во внешнем интерфейсе этого модуля имеется несколько программных сущностей. Это алгебраический тип данных и пара функций для манипуляции им.
Тип: `SCC`
Описание: представляет сильно связанные компоненты графа (SCC — от английского выражения *strongly connected components*). Каждый граф является одним или несколькими значениями типа `SCC`.

Определение:

```
data SCC vertex
  = AcyclicSCC vertex
  | CyclicSCC  [vertex]
```

Первый конструктор `AcyclicSCC` представляет собой отдельные вершины графа, которые не связаны ни в какой цикл. Соответственно, второй конструктор `CyclicSCC` представляет компонент графа, в котором выделено наибольшее количество связанных друг с другом вершин (непосредственно или косвенно).

Функция: `stronglyConnComp`

Описание: создаёт граф на основе списка вершин, каждая из которых содержит некоторое значение, идентификатор вершины и список идентификаторов, с которыми эта вершина связана. Граф получается направленным, топологически отсортированным.

Определение:

```
stronglyConnComp :: Ord key => [(node, key, [key])] -> [SCC node]
stronglyConnComp edges0 = map get_node (stronglyConnCompR edges0)
  where
    get_node (AcyclicSCC (n, _, _)) = AcyclicSCC n
    get_node (CyclicSCC triples)    = CyclicSCC [n | (n, _, _) <- triples]
```

Функция: `stronglyConnCompR`

Описание: вариант функции `stronglyConnComp`, который сохраняет первоначальную информацию, записанную в каждой вершине (идентификатор самой вершины, а также список идентификаторов связанных вершин).

Определение:

```
stronglyConnCompR :: Ord key => [(node, key, [key])] -> [SCC (node, key, [key])]
stronglyConnCompR []     = []
stronglyConnCompR edges0 = map decode forest
  where (graph, vertex_fn, _) = graphFromEdges edges0
        forest                = scc graph
        decode (Node v []) | mentions_itself v = CyclicSCC [vertex_fn v]
                           | otherwise         = AcyclicSCC (vertex_fn v)
        decode other = CyclicSCC (dec other [])
          where dec (Node v ts) vs = vertex_fn v : foldr dec vs ts
        mentions_itself v = v `elem` (graph ! v)
```

Функция: `flattenSCC`
Описание: возвращает список вершин из заданного сильно связанного компонента.
Определение:

```
flattenSCC :: SCC vertex -> [vertex]
flattenSCC (AcyclicSCC v) = [v]
flattenSCC (CyclicSCC vs) = vs
```

Функция: `flattenSCCs`
Описание: возвращает список вершин из заданных сильно связанных компонентов.
Определение:

```
flattenSCCs :: [SCC a] -> [a]
flattenSCCs = concatMap flattenSCC
```

Собственно, для представления графов создано несколько синонимов типов, которые позволяют описывать графы с метками в вершинах в виде ограниченных целых чисел. Данные типы суть `Graph`, `Table`, `Bounds`, `Edge` и `Vertex`.
Тип: `Vertex`
Описание: абстрактное представление вершины графа.
Определение:

```
type Vertex = Int
```

Тип: `Edge`
Описание: Абстрактное представление ребра графа. Ребро считается направленным от первой вершины в паре ко второй.
Определение:

```
type Edge = (Vertex, Vertex)
```

Тип: `Bounds`
Описание: граничные значения в типе `Table`.
Определение:

```
type Bounds = (Vertex, Vertex)
```

Тип: `Table`

Описание: индексированная таблица вершин графа для представления одного элемента в матрице смежности.

Определение:

```
type Table a = Array Vertex a
```

Тип: `Graph`

Описание: матрица смежности графа, в которой каждой вершине сопоставлен список вершин, к которым от неё отходит ребро.

Определение:

```
type Graph = Table [Vertex]
```

Уже для работы с этими типами предназначены следующие утилитарные функции.

Функция: `graphFromEdges`

Описание: строит граф из списка вершин, каждой из которых приписан уникальный идентификатор (ключ). Результатом является граф, к которому приписано две функции: функция для обратного преобразования вершины графа в исходную вершину с идентификатором и списком идентификаторов смежных вершин, а также функция поиска вершины по ключу.

Определение:

```
graphFromEdges :: Ord key => [(node, key, [key])] ->
                             (Graph, Vertex -> (node, key, [key]), key -> Maybe Vertex)
graphFromEdges edges0 = (graph, \v -> vertex_map ! v, key_vertex)
  where
    max_v           = length edges0 - 1
    bounds0         = (0, max_v) :: (Vertex, Vertex)
    sorted_edges    = sortBy lt edges0
    edges1          = zipWith (,) [0..] sorted_edges
    graph           = array bounds0 [(,) v (mapMaybe key_vertex ks)|
                                     (,) v (_, _, ks) <- edges1]
    key_map         = array bounds0 [(,) v k | (,) v (_, k, _ ) <- edges1]
    vertex_map      = array bounds0 edges1
    (_, k1, _) `lt` (_, k2, _) = k1 `compare` k2
    key_vertex k    = findVertex 0 max_v
      where
        findVertex a b | a > b = Nothing
        findVertex a b = case compare k (key_map ! mid) of
```

```
                        LT -> findVertex a (mid - 1)
                        EQ -> Just mid
                        GT -> findVertex (mid + 1) b
        where
          mid = (a + b) `div` 2
```

Функция: graphFromEdges'

Описание: вариант функции graphFromEdges, который не возвращает функцию поиска вершины по ключу.

Определение:

```
graphFromEdges' :: Ord key => [(node, key, [key])] -> (Graph, Vertex -> (node, key, [key]))
graphFromEdges' x = (a, b)
  where
    (a, b, _) = graphFromEdges x
```

Функция: buildG

Описание: строит граф по списку рёбер.

Определение:

```
buildG :: Bounds -> [Edge] -> Graph
buildG bounds0 edges0 = accumArray (flip (:)) [] bounds0 edges0
```

Функция: transposeG

Описание: возвращает граф, в котором все рёбра имеют противоположную направленность, чем в исходном графе.

Определение:

```
transposeG :: Graph -> Graph
transposeG g = buildG (bounds g) (reverseE g)
```

Функция: vertices

Описание: возвращает список вершин графа.

Определение:

```
vertices :: Graph -> [Vertex]
vertices = indices
```

Функция: edges

Описание: возвращает список рёбер графа.

Определение:

```
edges :: Graph -> [Edge]
edges g = [(v, w) | v <- vertices g, w <- g ! v ]
```

Функция: `outdegree`

Описание: возвращает матрицу количества рёбер, выходящих из каждой вершины графа.

Определение:

```
outdegree :: Graph -> Table Int
outdegree = mapT numEdges
  where
    numEdges _ ws = length ws
```

Функция: `indegree`

Описание: возвращает матрицу количества рёбер, входящих в каждую вершину графа.

Определение:

```
indegree :: Graph -> Table Int
indegree = outdegree . transposeG
```

Функция: `dfs`

Описание: лес части графа, в которую можно попасть из заданного набора вершин. Поиск осуществляется по принципу «сначала в глубину» по всем вершинам списка, начиная с начала.

Определение:

```
dfs :: Graph -> [Vertex] -> Forest Vertex
dfs g vs = prune (bounds g) (map (generate g) vs)
```

Функция: `dff`

Описание: недетерминированный лес части графа, в которую можно попасть из заданного набора вершин. Поиск осуществляется по принципу «сначала в глубину» по всем вершинам списка в произвольном порядке.

Определение:

```
dff :: Graph -> Forest Vertex
dff g = dfs g (vertices g)
```

Функция: `topSort`

Описание: возвращает топологически отсортированный граф.

Определение:

```
topSort :: Graph -> [Vertex]
topSort = reverse . postOrd
```

Функция: `components`

Описание: возвращает список связных компонентов графа. Связной называется такая часть графа, в которой от любой вершины существует путь до любой другой (не принимая во внимание направленность вершины).

Определение:

```
components :: Graph -> Forest Vertex
components = dff . undirected
```

Функция: `scc`

Описание: возвращает список сильно связных компонентов графа. Сильно связной называется такая часть графа, в которой от любой вершины существует путь до любой другой (принимая во внимание направленность вершины).

Определение:

```
scc :: Graph -> Forest Vertex
scc g = dfs g (reverse (postOrd (transposeG g)))
```

Функция: `bcc`

Описание: возвращает список двусвязных компонентов графа. Двусвязной называется такая часть графа, в которой от любой вершины существует двунаправленный путь до любой другой.

Определение:

```
bcc :: Graph -> Forest [Vertex]
bcc g = (concat . map bicomps . map (do_label g dnum)) forest
 where
   forest = dff g
   dnum   = preArr (bounds g) forest
```

Функция: `reachable`

Описание: список вершин, до которых можно добраться из заданной вершины графа.

Определение:

```
reachable :: Graph -> Vertex -> [Vertex]
reachable g v = preorderF (dfs g [v])
```

Функция: `path`
Описание: возвращает значение `True`, если до второй вершины существует путь из первой.
Определение:

```
path :: Graph -> Vertex -> Vertex -> Bool
path g v w = w `elem` (reachable g v)
```

В данном модуле имеются ещё некоторые функции, которые реализуют все алгоритмы, описанные в указанной работе [11]. Изучить эти функции можно, просмотрев их определения в исходном коде модуля.

8.13. Модуль `HashTable`

Модуль `HashTable` содержит реализацию расширяемых хеш-таблиц, которые описаны в работе [12]. Использование:

```
import Data.HashTable
```

Главным алгебраическим типом данных для представления хеш-таблиц является тип `HashTable`.
Тип: `HashTable`
Описание: тип для представления хеш-таблиц, которые можно модифицировать.
Определение:

```
data HashTable key val
  = HashTable {
               cmp     :: !(key -> key -> Bool),
               hash_fn :: !(key -> Int32),
               tab     :: !(IORef (HT key val))
             }
```

Функция: `new`
Описание: создаёт новую хеш-таблицу. В качестве входных параметров передаются две функции, первая из которых определяет свойство равенства на ключах,

а вторая — хеш-функцию для ключей. Для этих двух функций безусловно должно выполняться правило: `eq A B => hash A == hash B`.

Определение:

```
new :: (key -> key -> Bool) -> (key -> Int32) -> IO (HashTable key val)
new cmpr hash = do recordNew
                   let mask = tABLE_MIN - 1
                   bkts' <- newMutArray (0,mask) []
                   bkts  <- freezeArray bkts'
                   let kcnt = 0
                       ht = HT {buckets = bkts, kcount = kcnt, bmask = mask}
                   table <- newIORef ht
                   return (HashTable {tab = table, hash_fn = hash, cmp = cmpr})
```

Функция: `insert`

Описание: заносит новую пару (*значение*, *ключ*) в заданную хеш-таблицу. Необходимо отметить, что эта функция не удаляет старое значение для вносимого ключа (если таковое имеется). Это сделано ради увеличения эффективности функции. Функция `lookup` возвращает значение по ключу, которое внесено в хеш-таблицу последним. Для замены значений необходимо пользоваться функцией `update`.

Определение:

```
insert :: HashTable key val -> key -> val -> IO ()
insert ht key val = updatingBucket CanInsert
                                   (\bucket -> ((key, val):bucket, 1, ())) ht key
```

Функция: `delete`

Описание: удаляет запись из хеш-таблицы по заданному ключу.

Определение:

```
delete :: HashTable key val -> key -> IO ()
delete ht@HashTable{cmp = eq} key = updatingBucket Can'tInsert
                                                   (deleteBucket (eq key)) ht key
```

Функция: `lookup`

Описание: по заданному ключу возвращает значение из хеш-таблицы.

Определение:

```
lookup :: HashTable key val -> key -> IO (Maybe val)
lookup ht@HashTable{cmp = eq} key = do recordLookup (_,_, bucket) <- findBucket ht key
```

```
                                    let firstHit (k, v) r | eq key k  = Just v
                                                          | otherwise = r
                                    return (foldr firstHit Nothing bucket)
```

Функция: `update`

Описание: записывает новую пару (*значение*, *ключ*) в заданную хеш-таблицу. Если записываемый ключ уже имеется в хеш-таблице, происходит перезаписывание значения. Эта функции менее эффективна, чем функция `insert`, однако, более эффективна, чем связка функций `delete` и `insert` для перезаписи значения.

Определение:

```
update :: HashTable key val -> key -> val -> IO Bool

update ht@HashTable{cmp = eq} key val = updatingBucket CanInsert
  (\bucket -> let (bucket', dels, _) = deleteBucket (eq key) bucket
              in  ((key,val):bucket', 1 + dels, dels /= 0))
  ht key
```

Функция: `fromList`

Описание: преобразует список пар вида (*значение*, *ключ*) в хеш-таблицу.

Определение:

```
fromList :: (Eq key) => (key -> Int32) -> [(key,val)] -> IO (HashTable key val)
fromList hash list = do table <- new (==) hash
                        sequence_ [ insert table k v | (k, v) <- list ]
                        return table
```

Функция: `toList`

Описание: преобразует заданную хеш-таблицу в список пар вида (*значение*, *ключ*).

Определение:

```
toList :: HashTable key val -> IO [(key,val)]
toList = mapReduce id concat
```

Функция: `longestChain`

Описание: возвращает самую длинную последовательность пар вида (*значение*, *ключ*) для заданной таблицы, при этом ключ в данной последовательности всегда одинаков. Данная функция может использоваться для определения того, хороша ли хеширующая функция для работы с данными. Если возвращённая последовательность сравнительно велика (например, 14

элементов или больше), то необходимо использовать другую хеширующую функцию.

Определение:

```
longestChain :: HashTable key val -> IO [(key,val)]
longestChain = mapReduce id (maximumBy lengthCmp)
  where
    lengthCmp (_:x)(_:y) = lengthCmp x y
    lengthCmp []   []    = EQ
    lengthCmp []   _     = LT
    lengthCmp _    []    = GT
```

8.14. Модуль Int

В модуле Int определены типы для представления ограниченных целых чисел. В отличие от примитивного типа Int, в этом модуле описываются целочисленные типы, занимающие различное количество байт в памяти, а потому использующиеся в различных целях. Использование:

```
import Data.Int
```

Первый тип данных, который описан в этом модуле, — Int — переопределяет примитивный тип.

Тип: Int

Описание: тип для представления целых чисел фиксированной точности. Все значения этого типа лежат в интервале $[-2^{29}, 2^{29} - 1]$. Точные значения нижней и верхней границы для конкретной реализации (в зависимости от транслятора языка) можно узнать при помощи методов minBound и maxBound класса Bounded (см. стр. 115).

Определение:

Тип определён в виде примитива.

Для данного типа определены экземпляры следующих классов: Bits, Bounded, Data, Enum, Eq, Integral, Ix, NFData, NFDataIntegral, NFDataOrd, Num, Ord, PrintfArg, Random, Read, Real, Show, Storable, Typeable, IArray, и MArray.

Тип: Int8

Описание: тип для представления целых чисел со знаком, занимающих в памяти 8 бит (интервал $[-2^7, 2^7 - 1]$).

Определение:

Тип определён в виде примитива.

Тип: `Int16`

Описание: тип для представления целых чисел со знаком, занимающих в памяти 16 бит (интервал $[-2^{15}, 2^{15} - 1]$).

Определение:

Тип определён в виде примитива.

Тип: `Int32`

Описание: тип для представления целых чисел со знаком, занимающих в памяти 32 бита (интервал $[-2^{31}, 2^{31} - 1]$).

Определение:

Тип определён в виде примитива.

Тип: `Int64`

Описание: тип для представления целых чисел со знаком, занимающих в памяти 64 бита (интервал $[-2^{63}, 2^{63} - 1]$).

Определение:

Тип определён в виде примитива.

Для типов `Int8`, `Int16`, `Int32` и `Int64` определены экземпляры следующих классов: `Bits`, `Bounded`, `Data`, `Enum`, `Eq`, `Integral`, `Ix`, `Num`, `Ord`, `Read`, `Real`, `Show`, `Storable`, `Typeable`, `IArray` и `MArray`.

Необходимо отметить, что для перечисленных типов все арифметические операции выполняются по модулю 2^n, где n — количество бит, используемых для представления числа. Преобразование из одного целочисленного типа в другой можно производить при помощи функции `fromIntegral`, которая для всех перечисленных типов работает достаточно быстро. Также правила класса `Enum` (см. стр. 115) для типа `Int` работают и с определёнными в этом модуле целочисленными типами. Наконец, правый и левый сдвиги (см. раздел 8.2.) на число битов, равное или большее количеству битов, используемому для представления типа, возвращает значение 0 или -1 в зависимости от знака числа, над которым производится сдвиг.

8.15. Модуль `IntMap`

В модуле `IntMap` содержится описание такой важной идиомы в программировании, как отображения. Реализация отображений достаточно эффективна и ос-

нована на использовании специального вида деревьев (вместо обычных сбалансированных деревьев), описанных в работах [19, 16]. В этом модуле используются имена функций, которые конфликтуют со многими функциями из стандартного модуля `Prelude`, поэтому использование его выглядит следующим образом:

```
import Data.IntMap (IntMap)
import qualified Data.IntMap as IntMap
```

Реализация отображений, предлагаемая в этом модуле, достаточно эффективна для логических операций (например, `union` или `intersection`). Кроме того, операции по вставке и удалению элементов из отображений также выполняются быстро. Ну и большинство функций из этого модуля имеют сложность $O(min(n, W))$, где W — количество битов для представления ключей.

Рассматриваемый модуль является весьма привлекательным с точки зрения обучения правильному программированию на языке Haskell в частности и в функциональном стиле в целом, поскольку модуль не содержит ни одной программной сущности, определённой в виде примитива. Все типы данных, экземпляры классов и функции определены непосредственно в модуле.

Главный алгебраический тип данных, описанный в этом модуле, — `IntMap`. Этот тип используется для представления отображений с целочисленными ключами.

Тип: `IntMap`
Описание: отображение с целочисленными ключами.
Определение:

```
data IntMap a
  = Nil
  | Tip !Key a
  | Bin !Prefix !Mask !(IntMap a) !(IntMap a)
```

Для удобства работы определён синоним `Key`, который равен типу `Int`.

Тип `IntMap` определён в качестве экземпляра для следующих классов: `Foldable`, `Functor`, `Typeable1`, `Data`, `Eq`, `Monoid`, `Ord`, `Read` и `Show`.

Функция: `(!)`
Описание: осуществляет поиск элемента в отображении. Вызывает функцию `error`, если элемент не найден.

Определение:

```
(!) :: IntMap a -> Key -> a
m ! k = find' k m
```

Функция: (\\)

Описание: синоним функции `difference` (см. стр. 353).

Определение:

```
(\\) :: IntMap a -> IntMap b -> IntMap a
m1 \\ m2 = difference m1 m2
```

Функция: `null`

Описание: возвращает значение `True`, если заданное отображение пустое.

Определение:

```
null :: IntMap a -> Bool
null Nil   = True
null other = False
```

Функция: `size`

Описание: возвращает количество хранимых в заданном отображении элементов.

Определение:

```
size :: IntMap a -> Int
size t = case t of
           Bin p m l r -> size l + size r
           Tip k x     -> 1
           Nil         -> 0
```

Функция: `member`

Описание: возвращает значение `True`, если заданный ключ входит в отображение.

Определение:

```
member :: Key -> IntMap a -> Bool
member k m = case lookup k m of
               Nothing -> False
               Just x  -> True
```

Функция: `nonMember`

Описание: обращение предиката `member`.

Определение:

```
notMember :: Key -> IntMap a -> Bool
notMember k m = not $ member k m
```

Функция: `lookup`

Описание: возвращает из отображения значение по заданному ключу.

Определение:

```
lookup :: (Monad m) => Key -> IntMap a -> m a
lookup k t = case lookup' k t of
               Just x  -> return x
               Nothing -> fail "Data.IntMap.lookup: Key not found"


lookup' :: Key -> IntMap a -> Maybe a
lookup' k t = let nk = natFromInt k
              in  seq nk (lookupN nk t)


lookupN :: Nat -> IntMap a -> Maybe a
lookupN k t = case t of
                Bin p m l r | zeroN k (natFromInt m) -> lookupN k l
                            | otherwise              -> lookupN k r
                Tip kx x    | (k == natFromInt kx)   -> Just x
                            | otherwise              -> Nothing
                Nil                                  -> Nothing
```

Функция: `findWithDefault`

Описание: возвращает значение по заданному ключу, либо заданное значение по умолчанию, если по ключу значения в отображении нет.

Определение:

```
findWithDefault :: a -> Key -> IntMap a -> a
findWithDefault def k m = case lookup k m of
                            Nothing -> def
                            Just x  -> x
```

Функция: `empty`

Описание: возвращает пустое отображение.

Определение:

```
empty :: IntMap a
empty = Nil
```

Функция: `singleton`

Описание: возвращает отображение с одним элементом.

Определение:

```
singleton :: Key -> a -> IntMap a
singleton k x = Tip k x
```

Функция: `insert`

Описание: добавляет новую пару (*значение*, *ключ*) в отображение. Если значение с заданным ключом уже существует в отображении, оно замещается новым.

Определение:

```
insert :: Key -> a -> IntMap a -> IntMap a
insert k x t = case t of
                Bin p m l r | nomatch k p m -> join k (Tip k x) p t
                            | zero k m      -> Bin p m (insert k x l) r
                            | otherwise     -> Bin p m l (insert k x r)
                Tip ky y    | k == ky       -> Tip k x
                            | otherwise     -> join k (Tip k x) ky t
                Nil                         -> Tip k x
```

Функция: `insertWith`

Описание: добавляет новую пару (*значение*, *ключ*) в отображение. Если значение с заданным ключом уже существует в отображении, к старому и новому значению применяется комбинирующая функция.

Определение:

```
insertWith :: (a -> a -> a) -> Key -> a -> IntMap a -> IntMap a
insertWith f k x t = insertWithKey (\k x y -> f x y) k x t
```

Функция: `insertWithKey`

Описание: добавляет новую пару (*значение*, *ключ*) в отображение. Если значение с заданным ключом уже существует в отображении, к старому значению, а также к новому значению и ключу применяется комбинирующая функция.

Определение:

```
insertWithKey :: (Key -> a -> a -> a) -> Key -> a -> IntMap a -> IntMap a
insertWithKey f k x t
  = case t of
      Bin p m l r | nomatch k p m -> join k (Tip k x) p t
                  | zero k m      -> Bin p m (insertWithKey f k x l) r
                  | otherwise     -> Bin p m l (insertWithKey f k x r)
      Tip ky y    | k == ky       -> Tip k (f k x y)
                  | otherwise     -> join k (Tip k x) ky t
      Nil                         -> Tip k x
```

Функция: `insertLookupWithKey`

Описание: возвращает пару вида (`(Maybe a, IntMap a)`), где первый элемент является результатом выполнения функции `lookup`, а второй — результатом функции `insertLookupWithKey`.

Определение:

```
insertLookupWithKey :: (Key -> a -> a -> a) -> Key -> a -> IntMap a -> (Maybe a, IntMap a)
insertLookupWithKey f k x t
  = case t of
      Bin p m l r | nomatch k p m -> (Nothing, join k (Tip k x) p t)
                  | zero k m      -> let (found, l') = insertLookupWithKey f k x l
                                     in  (found, Bin p m l' r)
                  | otherwise     -> let (found, r') = insertLookupWithKey f k x r
                                     in  (found, Bin p m l r')
      Tip ky y    | k == ky       -> (Just y, Tip k (f k x y))
                  | otherwise     -> (Nothing, join k (Tip k x) ky t)
      Nil                         -> (Nothing,Tip k x)
```

Функция: `delete`

Описание: удаляет из отображения запись по ключу. Если заданного ключа нет в отображении, ничего не происходит.

Определение:

```
delete :: Key -> IntMap a -> IntMap a
delete k t = case t of
               Bin p m l r | nomatch k p m -> t
                           | zero k m      -> bin p m (delete k l) r
                           | otherwise     -> bin p m l (delete k r)
               Tip ky y    | k == ky       -> Nil
                           | otherwise     -> t
               Nil                         -> Nil
```

Функция: `adjust`

Описание: применяет заданную функцию к значению по ключу. Если заданного ключа нет в отображении, ничего не происходит.

Определение:

```
adjust ::  (a -> a) -> Key -> IntMap a -> IntMap a
adjust f k m = adjustWithKey (\k x -> f x) k m
```

Функция: `adjustWithKey`

Описание: применяет заданную функцию к значению по ключу (функция принимает на вход не только значение, но и сам ключ). Если заданного ключа нет в отображении, ничего не происходит.

Определение:

```
adjustWithKey ::  (Key -> a -> a) -> Key -> IntMap a -> IntMap a
adjustWithKey f k m = updateWithKey (\k x -> Just (f k x)) k m
```

Функция: `update`

Описание: выполняет обновление элемента или его удаление из отображения в зависимости от результата выполнения функции. Если функция (первый аргумент вызова) возвращает значение `Nothing`, то элемент удаляется из отображения. В противном случае в отображение записывается новое значение.

Определение:

```
update ::  (a -> Maybe a) -> Key -> IntMap a -> IntMap a
update f k m = updateWithKey (\k x -> f x) k m
```

Функция: `updateWithKey`

Описание: вариант функции `update`, в котором принимается на вход функция, оперирующая не только значением, но и ключом.

Определение:

```
updateWithKey ::  (Key -> a -> Maybe a) -> Key -> IntMap a -> IntMap a
updateWithKey f k t = case t of
                      Bin p m l r | nomatch k p m -> t
                                  | zero k m      -> bin p m (updateWithKey f k l) r
                                  | otherwise     -> bin p m l (updateWithKey f k r)
                      Tip ky y    | k == ky       -> case (f k y) of
                                                          Just y' -> Tip ky y'
                                                          Nothing -> Nil
                                  | otherwise     -> t
```

```
                    Nil                          -> Nil
```

Функция: updateLookupWithKey

Описание: одновременно выполняет действия функций lookup и updateWithKey для заданных значений. Результат возвращается в виде пары.

Определение:

```
updateLookupWithKey ::  (Key -> a -> Maybe a) -> Key -> IntMap a -> (Maybe a,IntMap a)
updateLookupWithKey f k t
  = case t of
      Bin p m l r | nomatch k p m -> (Nothing, t)
                  | zero k m      -> let (found, l') = updateLookupWithKey f k l
                                     in  (found, bin p m l' r)
                  | otherwise     -> let (found, r') = updateLookupWithKey f k r
                                     in  (found, bin p m l r')
      Tip ky y    | k == ky       -> case (f k y) of
                                       Just y' -> (Just y, Tip ky y')
                                       Nothing -> (Just y, Nil)
                  | otherwise     -> (Nothing, t)
      Nil                         -> (Nothing, Nil)
```

Функция: union

Описание: левонаправленное объединение двух отображений. Если в двух отображениях обнаружены одинаковые ключи, выбирается значение из левого (первого) отображения.

Определение:

```
union :: IntMap a -> IntMap a -> IntMap a
union t1@(Bin p1 m1 l1 r1) t2@(Bin p2 m2 l2 r2)
    | shorter m1 m2 = union1
    | shorter m2 m1 = union2
    | p1 == p2      = Bin p1 m1 (union l1 l2) (union r1 r2)
    | otherwise     = join p1 t1 p2 t2
  where
    union1 | nomatch p2 p1 m1 = join p1 t1 p2 t2
           | zero p2 m1       = Bin p1 m1 (union l1 t2) r1
           | otherwise        = Bin p1 m1 l1 (union r1 t2)
    union2 | nomatch p1 p2 m2 = join p1 t1 p2 t2
           | zero p1 m2       = Bin p2 m2 (union t1 l2) r2
           | otherwise        = Bin p2 m2 l2 (union t1 r2)
union (Tip k x) t = insert k x t
union t (Tip k x) = insertWith (\x y -> y) k x t  -- right bias
union Nil t       = t
```

```
union t Nil       = t
```

Функция: unionWith

Описание: объединяет два отображения, применяя заданную функцию к двум значениям из обоих отображений в случае, если найден одинаковый ключ.

Определение:

```
unionWith :: (a -> a -> a) -> IntMap a -> IntMap a -> IntMap a
unionWith f m1 m2 = unionWithKey (\k x y -> f x y) m1 m2
```

Функция: unionWithKey

Описание: вариант функции unionWith, в котором используется объединяющая функция, принимающая на вход не только два значения, но и ключ.

Определение:

```
unionWithKey :: (Key -> a -> a -> a) -> IntMap a -> IntMap a -> IntMap a
unionWithKey f t1@(Bin p1 m1 l1 r1) t2@(Bin p2 m2 l2 r2)
    | shorter m1 m2 = union1
    | shorter m2 m1 = union2
    | p1 == p2      = Bin p1 m1 (unionWithKey f l1 l2) (unionWithKey f r1 r2)
    | otherwise     = join p1 t1 p2 t2
  where
    union1 | nomatch p2 p1 m1 = join p1 t1 p2 t2
           | zero p2 m1       = Bin p1 m1 (unionWithKey f l1 t2) r1
           | otherwise        = Bin p1 m1 l1 (unionWithKey f r1 t2)
    union2 | nomatch p1 p2 m2 = join p1 t1 p2 t2
           | zero p1 m2       = Bin p2 m2 (unionWithKey f t1 l2) r2
           | otherwise        = Bin p2 m2 l2 (unionWithKey f t1 r2)

unionWithKey f (Tip k x) t = insertWithKey f k x t
unionWithKey f t (Tip k x) = insertWithKey (\k x y -> f k y x) k x t  -- right bias
unionWithKey f Nil t       = t
unionWithKey f t Nil       = t
```

Функция: unions

Описание: объединение списка отображений в одно.

Определение:

```
unions :: [IntMap a] -> IntMap a
unions xs = foldlStrict union empty xs
```

Функция: `unionsWith`

Описание: объединение списка отображений в одно с применением функции для объединения значений в случае дублирования ключа.

Определение:

```
unionsWith :: (a->a->a) -> [IntMap a] -> IntMap a
unionsWith f ts = foldlStrict (unionWith f) empty ts
```

Функция: `difference`

Описание: возвращает разницу двух отображений (рассчитывается на основе ключей).

Определение:

```
difference :: IntMap a -> IntMap b -> IntMap a
difference t1@(Bin p1 m1 l1 r1) t2@(Bin p2 m2 l2 r2)
    | shorter m1 m2 = difference1
    | shorter m2 m1 = difference2
    | p1 == p2      = bin p1 m1 (difference l1 l2) (difference r1 r2)
    | otherwise     = t1
  where
    difference1 | nomatch p2 p1 m1  = t1
                | zero p2 m1        = bin p1 m1 (difference l1 t2) r1
                | otherwise         = bin p1 m1 l1 (difference r1 t2)
    difference2 | nomatch p1 p2 m2  = t1
                | zero p1 m2        = difference t1 l2
                | otherwise         = difference t1 r2

difference t1@(Tip k x) t2
    | member k t2  = Nil
    | otherwise    = t1

difference Nil t       = Nil
difference t (Tip k x) = delete k t
difference t Nil       = t
```

Функция: `differenceWith`

Описание: возвращает разницу двух отображений (рассчитывается на основе ключей) с применением комбинирующей функции в случае, если найден одинаковый ключ в двух отображениях.

Определение:

```
differenceWith :: (a -> b -> Maybe a) -> IntMap a -> IntMap b -> IntMap a
differenceWith f m1 m2 = differenceWithKey (\k x y -> f x y) m1 m2
```

Функция: `differenceWithKey`

Описание: вариант функции `differenceWith`, в котором комбинирующая функция получает не только два значения из двух отображений, но и ключ.

Определение:

```
differenceWithKey :: (Key -> a -> b -> Maybe a) -> IntMap a -> IntMap b -> IntMap a
differenceWithKey f t1@(Bin p1 m1 l1 r1) t2@(Bin p2 m2 l2 r2)
    | shorter m1 m2 = difference1
    | shorter m2 m1 = difference2
    | p1 == p2      = bin p1 m1 (differenceWithKey f l1 l2) (differenceWithKey f r1 r2)
    | otherwise     = t1
  where
    difference1 | nomatch p2 p1 m1  = t1
                | zero p2 m1        = bin p1 m1 (differenceWithKey f l1 t2) r1
                | otherwise         = bin p1 m1 l1 (differenceWithKey f r1 t2)
    difference2 | nomatch p1 p2 m2  = t1
                | zero p1 m2        = differenceWithKey f t1 l2
                | otherwise         = differenceWithKey f t1 r2

differenceWithKey f t1@(Tip k x) t2 = case lookup k t2 of
                                        Just y  -> case f k x y of
                                                     Just y' -> Tip k y'
                                                     Nothing -> Nil
                                        Nothing -> t1
differenceWithKey f Nil t       = Nil
differenceWithKey f t (Tip k y) = updateWithKey (\k x -> f k x y) k t
differenceWithKey f t Nil       = t
```

Функция: `intersection`

Описание: возвращает пересечение двух отображений. Если в обоих отображениях найден одинаковый ключ, выбирается значение из левого (первого) отображения.

Определение:

```
intersection :: IntMap a -> IntMap b -> IntMap a
intersection t1@(Bin p1 m1 l1 r1) t2@(Bin p2 m2 l2 r2)
    | shorter m1 m2 = intersection1
    | shorter m2 m1 = intersection2
    | p1 == p2      = bin p1 m1 (intersection l1 l2) (intersection r1 r2)
    | otherwise     = Nil
  where
    intersection1 | nomatch p2 p1 m1  = Nil
                  | zero p2 m1        = intersection l1 t2
                  | otherwise         = intersection r1 t2
    intersection2 | nomatch p1 p2 m2  = Nil
                  | zero p1 m2        = intersection t1 l2
                  | otherwise         = intersection t1 r2
intersection t1@(Tip k x) t2
    | member k t2  = t1
    | otherwise    = Nil
intersection t (Tip k x) = case lookup k t of
                             Just y  -> Tip k y
                             Nothing -> Nil
intersection Nil t = Nil
intersection t Nil = Nil
```

Функция: `intersectionWith`

Описание: возвращает пересечение двух отображений с применением комбинирующей функции в случае, если найден одинаковый ключ в двух отображениях.

Определение:

```
intersectionWith :: (a -> b -> a) -> IntMap a -> IntMap b -> IntMap a
intersectionWith f m1 m2 = intersectionWithKey (\k x y -> f x y) m1 m2
```

Функция: `intersectionWithKey`

Описание: вариант функции `intersectionWith`, в котором комбинирующая функция получает не только два значения из двух отображений, но и ключ.

Определение:

```
intersectionWithKey :: (Key -> a -> b -> a) -> IntMap a -> IntMap b -> IntMap a
intersectionWithKey f t1@(Bin p1 m1 l1 r1) t2@(Bin p2 m2 l2 r2)
    | shorter m1 m2 = intersection1
    | shorter m2 m1 = intersection2
    | p1 == p2      = bin p1 m1 (intersectionWithKey f l1 l2) (intersectionWithKey f r1 r2)
    | otherwise     = Nil
```

```
  where
    intersection1 | nomatch p2 p1 m1  = Nil
                  | zero p2 m1        = intersectionWithKey f l1 t2
                  | otherwise         = intersectionWithKey f r1 t2
    intersection2 | nomatch p1 p2 m2  = Nil
                  | zero p1 m2        = intersectionWithKey f t1 l2
                  | otherwise         = intersectionWithKey f t1 r2
intersectionWithKey f t1@(Tip k x) t2 = case lookup k t2 of
                                          Just y  -> Tip k (f k x y)
                                          Nothing -> Nil
intersectionWithKey f t1 (Tip k y) = case lookup k t1 of
                                       Just x  -> Tip k (f k x y)
                                       Nothing -> Nil
intersectionWithKey f Nil t = Nil
intersectionWithKey f t Nil = Nil
```

Функция: `map`

Описание: применяет заданную функцию ко всем значениям в отображении.

Определение:

```
map :: (a -> b) -> IntMap a -> IntMap b
map f m = mapWithKey (\k x -> f x) m
```

Функция: `mapWithKey`

Описание: вариант функции `map`, в котором функция применяется не только к значениям, но и к ключам.

Определение:

```
mapWithKey :: (Key -> a -> b) -> IntMap a -> IntMap b
mapWithKey f t = case t of
                   Bin p m l r -> Bin p m (mapWithKey f l) (mapWithKey f r)
                   Tip k x     -> Tip k (f k x)
                   Nil         -> Nil
```

Функция: `mapAccum`

Описание: не только применяет заданную функцию ко всем значениям в заданном отображении, но и собирает по результатам выполнения функции некоторое значение, которое возвращает в пару с новым отображением.

Определение:

```
mapAccum :: (a -> b -> (a,c)) -> a -> IntMap b -> (a,IntMap c)
mapAccum f a m = mapAccumWithKey (\a k x -> f a x) a m
```

Функция: mapAccumWithKey

Описание: вариант функции mapAccum, в котором функция применяется не только к значениям, но и к ключам.

Определение:

```
mapAccumWithKey :: (a -> Key -> b -> (a, c)) -> a -> IntMap b -> (a, IntMap c)
mapAccumWithKey f a t = mapAccumL f a t


mapAccumL :: (a -> Key -> b -> (a, c)) -> a -> IntMap b -> (a, IntMap c)
mapAccumL f a t = case t of
                  Bin p m l r -> let (a1, l') = mapAccumL f a l
                                     (a2, r') = mapAccumL f a1 r
                                 in  (a2, Bin p m l' r')
                  Tip k x     -> let (a', x') = f a k x
                                 in  (a', Tip k x')
                  Nil         -> (a, Nil)


mapAccumR :: (a -> Key -> b -> (a, c)) -> a -> IntMap b -> (a, IntMap c)
mapAccumR f a t = case t of
                  Bin p m l r -> let (a1, r') = mapAccumR f a r
                                     (a2, l') = mapAccumR f a1 l
                                 in  (a2, Bin p m l' r')
                  Tip k x     -> let (a', x') = f a k x
                                 in  (a', Tip k x')
                  Nil         -> (a, Nil)
```

Функция: fold

Описание: сворачивает все элементы отображения в один при помощи заданной функции и заданного начального значения.

Определение:

```
fold :: (a -> b -> b) -> b -> IntMap a -> b
fold f z t = foldWithKey (\k x y -> f x y) z t
```

Функция: `foldWithKey`

Описание: вариант функции `fold`, в котором сворачивающая функция работает не только со значениями, но и с ключами.

Определение:

```
foldWithKey :: (Key -> a -> b -> b) -> b -> IntMap a -> b
foldWithKey f z t = foldr f z t


foldr :: (Key -> a -> b -> b) -> b -> IntMap a -> b
foldr f z t = case t of
                Bin 0 m l r | m < 0 -> foldr' f (foldr' f z l) r
                Bin _ _ _ _         -> foldr' f z t
                Tip k x             -> f k x z
                Nil                 -> z
```

Функция: `elems`

Описание: возвращает список всех значений, хранящихся в заданном отображении.

Определение:

```
elems :: IntMap a -> [a]
elems m = foldWithKey (\k x xs -> x:xs) [] m
```

Функция: `keys`

Описание: возвращает список всех ключей, хранящихся в заданном отображении.

Определение:

```
keys :: IntMap a -> [Key]
keys m = foldWithKey (\k x ks -> k:ks) [] m
```

Функция: `keysSet`

Описание: возвращает множество всех ключей, хранящихся в заданном отображении.

Определение:

```
keysSet :: IntMap a -> IntSet.IntSet
keysSet m = IntSet.fromDistinctAscList (keys m)
```

Функция: `assocs`

Описание: возвращает список ассоциаций всех значений и ключей, хранящихся в заданном отображении.

Определение:

```
assocs :: IntMap a -> [(Key,a)]
assocs m = toList m
```

Функция: `toList`

Описание: синоним функции `assocs`.

Определение:

```
toList :: IntMap a -> [(Key,a)]
toList t = foldWithKey (\k x xs -> (k, x):xs) [] t
```

Функция: `fromList`

Описание: создаёт отображение на основе ассоциированного списка, содержащего значения вида (*ключ*, *значение*).

Определение:

```
fromList :: [(Key,a)] -> IntMap a
fromList xs = foldlStrict ins empty xs
  where
    ins t (k, x) = insert k x t
```

Функция: `fromListWith`

Описание: создаёт отображение на основе ассоциированного списка, содержащего значения вида (*ключ*, *значение*), при этом для обработки дублирующихся ключей используется комбинирующая функция.

Определение:

```
fromListWith :: (a -> a -> a) -> [(Key,a)] -> IntMap a
fromListWith f xs = fromListWithKey (\k x y -> f x y) xs
```

Функция: `fromListWithKey`

Описание: вариант функции `fromListWith`, в котором комбинирующая функция принимает на вход не только значения, но и ключ.

Определение:

```
fromListWithKey :: (Key -> a -> a -> a) -> [(Key,a)] -> IntMap a
fromListWithKey f xs = foldlStrict ins empty xs
  where
    ins t (k, x) = insertWithKey f k x t
```

Функция: `toAscList`

Описание: вариант функции `toList`, который возвращает список, в котором ключи идут в строго возрастающем порядке.

Определение:

```
toAscList :: IntMap a -> [(Key,a)]
toAscList t = let (pos, neg) = span (\(k, x) -> k >=0) (foldr (\k x xs -> (k, x):xs) [] t)
              in  neg ++ pos
```

Функция: `fromAscList`

Описание: вариант функции `fromList`, в который подаётся на вход список, в котором ключи идут в строго возрастающем порядке.

Определение:

```
fromAscList :: [(Key,a)] -> IntMap a
fromAscList xs = fromList xs
```

Функция: `fromAscListWith`

Описание: вариант функции `fromListWith`, в который подаётся на вход список, в котором ключи идут в строго возрастающем порядке.

Определение:

```
fromAscListWith :: (a -> a -> a) -> [(Key,a)] -> IntMap a
fromAscListWith f xs = fromListWith f xs
```

Функция: `fromAscListWithKey`

Описание: вариант функции `fromListWithKey`, в который подаётся на вход список, в котором ключи идут в строго возрастающем порядке.

Определение:

```
fromAscListWithKey :: (Key -> a -> a -> a) -> [(Key,a)] -> IntMap a
fromAscListWithKey f xs = fromListWithKey f xs
```

Функция: `fromDistinctAscList`

Описание: вариант функции `fromAscList`, в котором список, передаваемый на вход, не должен иметь дублирующихся ключей.

Определение:

```
fromDistinctAscList :: [(Key,a)] -> IntMap a
fromDistinctAscList xs = fromList xs
```

Функция: `filter`

Описание: отсеивает значения из отображения в соответствии с заданным предикатом (в выходном отображении остаются только те значения, на которых предикат возвратил `True`).

Определение:

```
filter :: (a -> Bool) -> IntMap a -> IntMap a
filter p m = filterWithKey (\k x -> p x) m
```

Функция: `filterWithKey`

Описание: вариант функции `filter`, в котором предикат принимает не только значения из отображения, но и ключи.

Определение:

```
filterWithKey :: (Key -> a -> Bool) -> IntMap a -> IntMap a
filterWithKey pred t = case t of
                         Bin p m l r         -> bin p m (filterWithKey pred l)
                                                        (filterWithKey pred r)
                         Tip k x | pred k x  -> t
                                 | otherwise -> Nil
                         Nil                 -> Nil
```

Функция: `partition`

Описание: разделяет заданное отображение в соответствии с предикатом. Первое отображение в результате содержит те значения с ключами, которые удовлетворяют предикат. Второй отображение содержит, соответственно, не удовлетворяющие предикат значения.

Определение:

```
partition :: (a -> Bool) -> IntMap a -> (IntMap a,IntMap a)
partition p m = partitionWithKey (\k x -> p x) m
```

Функция: `partitionWithKey`

Описание: вариант функции `partition`, в котором предикат принимает не только значения из отображения, но и ключи.

Определение:

```
partitionWithKey :: (Key -> a -> Bool) -> IntMap a -> (IntMap a,IntMap a)
partitionWithKey pred t = case t of
                            Bin p m l r -> let (l1, l2) = partitionWithKey pred l
                                               (r1, r2) = partitionWithKey pred r
```

```
                    in  (bin p m l1 r1, bin p m l2 r2)
Tip k x | pred k x  -> (t, Nil)
        | otherwise -> (Nil, t)
Nil                 -> (Nil, Nil)
```

Функция: mapMaybe

Описание: производит применение заданной функции, которая возвращает значения из монады Maybe, к заданному отображению. Результатом является отображение, в котором собраны только результаты указанной функции типа Just.

Определение:

```
mapMaybe :: (a -> Maybe b) -> IntMap a -> IntMap b
mapMaybe f m = mapMaybeWithKey (\k x -> f x) m
```

Функция: mapMaybeWithKey

Описание: вариант функции mapMaybe, в котором предикат принимает не только значения из отображения, но и ключи.

Определение:

```
mapMaybeWithKey :: (Key -> a -> Maybe b) -> IntMap a -> IntMap b
mapMaybeWithKey f (Bin p m l r) = bin p m (mapMaybeWithKey f l) (mapMaybeWithKey f r)
mapMaybeWithKey f (Tip k x)     = case f k x of
                                    Just y  -> Tip k y
                                    Nothing -> Nil
mapMaybeWithKey f Nil           = Nil
```

Функция: mapEither

Описание: производит применение заданной функции, которая возвращает значения типа Either к заданному отображению. Результатом является пара, в которой разделены отображения типов Left и Right.

Определение:

```
mapEither :: (a -> Either b c) -> IntMap a -> (IntMap b, IntMap c)
mapEither f m = mapEitherWithKey (\k x -> f x) m
```

Функция: mapEitherWithKey

Описание: вариант функции mapEither, в котором предикат принимает не только значения из отображения, но и ключи.

Определение:

```
mapEitherWithKey :: (Key -> a -> Either b c) -> IntMap a -> (IntMap b, IntMap c)
mapEitherWithKey f (Bin p m l r) = (bin p m l1 r1, bin p m l2 r2)
  where
    (l1, l2) = mapEitherWithKey f l
    (r1, r2) = mapEitherWithKey f r
mapEitherWithKey f (Tip k x)     = case f k x of
                                       Left y  -> (Tip k y, Nil)
                                       Right z -> (Nil, Tip k z)
mapEitherWithKey f Nil           = (Nil, Nil)
```

Функция: `split`

Описание: разделяет заданное отображение на два, в первом из которых все ключи меньше заданного, а во втором — больше. Ключи, равные заданному, не попадают в результирующие отображения.

Определение:

```
split :: Key -> IntMap a -> (IntMap a,IntMap a)
split k t = case t of
             Bin p m l r | m < 0     -> (if k >= 0
                                         then let (lt, gt) = split' k l
                                              in  (union r lt, gt)
                                         else let (lt, gt) = split' k r
                                              in  (lt, union gt l))
                         | otherwise -> split' k t
             Tip ky y | k > ky       -> (t, Nil)
                      | k < ky       -> (Nil, t)
                      | otherwise    -> (Nil,Nil)
             Nil                     -> (Nil, Nil)


split' :: Key -> IntMap a -> (IntMap a,IntMap a)
split' k t = case t of
              Bin p m l r | nomatch k p m -> if k > p
                                             then (t, Nil)
                                             else (Nil, t)
                          | zero k m      -> let (lt, gt) = split k l
                                             in  (lt, union gt r)
                          | otherwise     -> let (lt, gt) = split k r
                                             in  (union l lt, gt)
              Tip ky y | k > ky           -> (t, Nil)
                       | k < ky           -> (Nil, t)
                       | otherwise        -> (Nil, Nil)
```

```
      Nil                          -> (Nil, Nil)
```

Функция: splitLookup

Описание: вариант функции split, который не только разбивает исходной отображение на два, но и возвращает значение по заданному ключу.

Определение:

```
splitLookup :: Key -> IntMap a -> (IntMap a,Maybe a,IntMap a)
splitLookup k t
  = case t of
      Bin p m l r | m < 0     -> (if k >= 0
                                    then let (lt, found, gt) = splitLookup' k l
                                         in (union r lt, found, gt)
                                    else let (lt, found, gt) = splitLookup' k r
                                         in (lt, found, union gt l))
                                | otherwise -> splitLookup' k t
      Tip ky y | k > ky       -> (t, Nothing, Nil)
               | k < ky       -> (Nil, Nothing, t)
               | otherwise    -> (Nil, Just y, Nil)
      Nil                     -> (Nil, Nothing, Nil)


splitLookup' :: Key -> IntMap a -> (IntMap a,Maybe a,IntMap a)
splitLookup' k t
  = case t of
      Bin p m l r | nomatch k p m -> if k > p
                                       then (t, Nothing, Nil)
                                       else (Nil, Nothing, t)
                  | zero k m      -> let (lt, found, gt) = splitLookup k l
                                       in  (lt, found, union gt r)
                  | otherwise     -> let (lt, found, gt) = splitLookup k r
                                       in  (union l lt, found, gt)
      Tip ky y | k > ky            -> (t, Nothing, Nil)
               | k < ky            -> (Nil, Nothing, t)
               | otherwise         -> (Nil, Just y, Nil)
      Nil                          -> (Nil, Nothing, Nil)
```

Функция: isSubmapOf

Описание: возвращает значение True, если первое отображение входит во второе.

Определение:

```
isSubmapOf :: Eq a => IntMap a -> IntMap a -> Bool
isSubmapOf m1 m2 = isSubmapOfBy (==) m1 m2
```

Функция: isSubmapOfBy

Описание: вариант функции isSubmapOf, в котором сравнение производится при помощи заданного первым аргументом предиката.

Определение:

```
isSubmapOfBy :: (a -> b -> Bool) -> IntMap a -> IntMap b -> Bool
isSubmapOfBy pred t1@(Bin p1 m1 l1 r1) t2@(Bin p2 m2 l2 r2)
    | shorter m1 m2 = False
    | shorter m2 m1 = match p1 p2 m2 && (if zero p1 m2
                                         then isSubmapOfBy pred t1 l2
                                         else isSubmapOfBy pred t1 r2)
    | otherwise     = (p1 == p2) && isSubmapOfBy pred l1 l2 && isSubmapOfBy pred r1 r2
isSubmapOfBy pred (Bin p m l r) t = False
isSubmapOfBy pred (Tip k x) t = case lookup k t of
                                  Just y  -> pred x y
                                  Nothing -> False
isSubmapOfBy pred Nil t = True
```

Функция: isProperSubmapOf

Описание: возвращает значение True, если первое отображение входит во второе, и при этом отображения не равны.

Определение:

```
isProperSubmapOf :: Eq a => IntMap a -> IntMap a -> Bool
isProperSubmapOf m1 m2 = isProperSubmapOfBy (==) m1 m2
```

Функция: isProperSubmapOfBy

Описание: вариант функции isProperSubmapOf, в котором сравнение производится при помощи заданного первым аргументом предиката.

Определение:

```
isProperSubmapOfBy :: (a -> b -> Bool) -> IntMap a -> IntMap b -> Bool
isProperSubmapOfBy pred t1 t2 = case submapCmp pred t1 t2 of
                                  LT -> True
                                  ge -> False


submapCmp pred t1@(Bin p1 m1 l1 r1) t2@(Bin p2 m2 l2 r2)
    | shorter m1 m2 = GT
    | shorter m2 m1 = submapCmpLt
    | p1 == p2      = submapCmpEq
    | otherwise     = GT
```

```
  where
    submapCmpLt | nomatch p1 p2 m2 = GT
                | zero p1 m2       = submapCmp pred t1 l2
                | otherwise        = submapCmp pred t1 r2
    submapCmpEq = case (submapCmp pred l1 l2, submapCmp pred r1 r2) of
                    (GT, _ ) -> GT
                    (_ , GT) -> GT
                    (EQ, EQ) -> EQ
                    other    -> LT
submapCmp pred (Bin p m l r) t = GT
submapCmp pred (Tip kx x) (Tip ky y)
    | (kx == ky) && pred x y = EQ
    | otherwise              = GT
submapCmp pred (Tip k x) t = case lookup k t of
                               Just y | pred x y -> LT
                               other             -> GT
submapCmp pred Nil Nil = EQ
submapCmp pred Nil t   = LT
```

Функция: `showTree`

Описание: преобразует заданное отображение в строку для вывода на экран. Функция используется для отладочных целей.

Определение:

```
showTree :: Show a => IntMap a -> String
showTree s = showTreeWith True False s
```

Функция: `showTreeWith`

Описание: вариант функции `showTree`, который используется для тонкой настройки при преобразовании отображения в строку. Первый аргумент используется для того, чтобы определять, в каком виде преобразовывать дерево отображения. Если этот аргумент равен `True`, то дерево показывается иерархическим, в противном случае — циклическим. Если второй аргумент функции равен `True`, то при преобразовании в строку не учитывается длина подстрок.

Определение:

```
showTreeWith :: Show a => Bool -> Bool -> IntMap a -> String
showTreeWith hang wide t | hang      = (showsTreeHang wide [] t) ""
                         | otherwise = (showsTree wide [] [] t) ""
```

```
showsTree :: Show a => Bool -> [String] -> [String] -> IntMap a -> ShowS
showsTree wide lbars rbars t
  = case t of
      Bin p m l r -> showsTree wide (withBar rbars) (withEmpty rbars) r .
                     showWide wide rbars .
                     showsBars lbars . showString (showBin p m) . showString "\n" .
                     showWide wide lbars .
                     showsTree wide (withEmpty lbars) (withBar lbars) l
      Tip k x     -> showsBars lbars . showString " " . shows k .
                     showString ":=" . shows x . showString "\n"
      Nil         -> showsBars lbars . showString "|\n"


showsTreeHang :: Show a => Bool -> [String] -> IntMap a -> ShowS
showsTreeHang wide bars t
  = case t of
      Bin p m l r -> showsBars bars . showString (showBin p m) . showString "\n" .
                     showWide wide bars .
                     showsTreeHang wide (withBar bars) l .
                     showWide wide bars .
                     showsTreeHang wide (withEmpty bars) r
      Tip k x     -> showsBars bars . showString " " . shows k .
                     showString ":=" . shows x . showString "\n"
      Nil         -> showsBars bars . showString "|\n"


showBin p m = "*" -- ++ show (p, m)


showWide wide bars | wide      = showString (concat (reverse bars)) . showString "|\n"
                   | otherwise = id


showsBars :: [String] -> ShowS
showsBars bars = case bars of
                   [] -> id
                   _  -> showString (concat (reverse (tail bars))) . showString node


node           = "+--"
withBar bars   = "|  ":bars
withEmpty bars = "   ":bars
```

8.16. Модуль IntSet

В модуле IntSet содержится описание не менее важной идиомы в программировании, как множество (на целых числах). Реализация множеств достаточно эффективна и основана на использовании специального вида деревьев (вместо обычных сбалансированных деревьев), описанных в работах [19, 16]. В этом модуле ипользуются имена функций, которые конфликтуют со многими функциями из стандартного модуля Prelude, поэтому использование его выглядит следующим образом:

```
import Data.IntSet (IntSet)
import qualified Data.IntSet as IntSet
```

Реализация функций для работы с множествами, предлагаемая в этом модуле, достаточно эффективна и имеет в большинстве случаев сложность $O(min(n, W))$, где W — количество битов для представления ключей.

Рассматриваемый модуль является весьма привлекательным с точки зрения обучения правильному программированию на языке Haskell в частности и в функциональном стиле в целом, поскольку модуль не содержит ни одной программной сущности, определённой в виде примитива. Все типы данных, экземпляры классов и функции определены непосредственно в модуле.

Главный алгебраический тип данных, описанный в этом модуле, — IntSet. Этот тип используется для представления множеств целых числе.

Тип: IntSet

Описание: множество целых чисел.

Определение:

```
data IntSet
  = Nil
  | Tip !Int
  | Bin !Prefix !Mask !IntSet !IntSet
```

Для удобства здесь также определены два синонима типов: Prefix и Mask, которые равны типу Int.

Для типа IntSet определены экземпляры следующих классов: Data, Eq, Monoid, Ord, Read, Show и Typeable.

Функция: (\\)

Описание: синоним функции difference (см. стр. 353).

Определение:

```
(\\) :: IntSet -> IntSet -> IntSet
m1 \\ m2 = difference m1 m2
```

Функция: `null`

Описание: возвращает значение `True`, если заданное множество пустое.

Определение:

```
null :: IntSet -> Bool
null Nil   = True
null other = False
```

Функция: `size`

Описание: возвращает кардинальное число заданного множества (количество элементов в множестве.

Определение:

```
size :: IntSet -> Int
size t = case t of
          Bin p m l r -> size l + size r
          Tip y       -> 1
          Nil         -> 0
```

Функция: `member`

Описание: предикат, возвращающий значение `True`, если заданный элемент принадлежит заданному множеству.

Определение:

```
member :: Int -> IntSet -> Bool
member x t = case t of
               Bin p m l r | nomatch x p m -> False
                           | zero x m      -> member x l
                           | otherwise     -> member x r
               Tip y                       -> (x == y)
               Nil                         -> False
```

Функция: `notMember`

Описание: обращение предиката `member`.

Определение:

```
notMember :: Int -> IntSet -> Bool
notMember k = not . member k
```

Функция: `isSubsetOf`

Описание: предикат, возвращающий значение `True`, если заданное множество является подмножеством множества, передаваемого вторым аргументом.

Определение:

```
isSubsetOf :: IntSet -> IntSet -> Bool
isSubsetOf t1@(Bin p1 m1 l1 r1) t2@(Bin p2 m2 l2 r2)
    | shorter m1 m2 = False
    | shorter m2 m1 = match p1 p2 m2 && (if zero p1 m2
                                         then isSubsetOf t1 l2
                                         else isSubsetOf t1 r2)
    | otherwise     = (p1 == p2) && isSubsetOf l1 l2 && isSubsetOf r1 r2
isSubsetOf (Bin p m l r) t = False
isSubsetOf (Tip x) t       = member x t
isSubsetOf Nil t           = True
```

Функция: `isProperSubsetOf`

Описание: предикат, возвращающий значение `True`, если заданное множество является собственным подмножеством (подмножеством, не равным самому множеству) множества, передаваемого вторым аргументом.

Определение:

```
isProperSubsetOf :: IntSet -> IntSet -> Bool
isProperSubsetOf t1 t2 = case subsetCmp t1 t2 of
                           LT -> True
                           ge -> False


subsetCmp t1@(Bin p1 m1 l1 r1) t2@(Bin p2 m2 l2 r2)
    | shorter m1 m2 = GT
    | shorter m2 m1 = subsetCmpLt
    | p1 == p2      = subsetCmpEq
    | otherwise     = GT
  where
    subsetCmpLt | nomatch p1 p2 m2 = GT
                | zero p1 m2       = subsetCmp t1 l2
                | otherwise        = subsetCmp t1 r2
    subsetCmpEq = case (subsetCmp l1 l2, subsetCmp r1 r2) of
```

```
                    (GT, _ ) -> GT
                    (_ , GT) -> GT
                    (EQ, EQ) -> EQ
                    other    -> LT
subsetCmp (Bin p m l r) t  = GT
subsetCmp (Tip x) (Tip y)
    | x == y    = EQ
    | otherwise = GT
subsetCmp (Tip x) t
    | member x t = LT
    | otherwise  = GT
subsetCmp Nil Nil = EQ
subsetCmp Nil t   = LT
```

Функция: `empty`

Описание: создаёт пустое множество.

Определение:

```
empty :: IntSet
empty = Nil
```

Функция: `singleton`

Описание: создаёт множество из одного элемента.

Определение:

```
singleton :: Int -> IntSet
singleton x = Tip x
```

Функция: `insert`

Описание: добавляет новый элемент в заданное множество. Если элемент уже присутствует в множестве, он заменяется.

Определение:

```
insert :: Int -> IntSet -> IntSet
insert x t = case t of
               Bin p m l r | nomatch x p m -> join x (Tip x) p t
                           | zero x m      -> Bin p m (insert x l) r
                           | otherwise     -> Bin p m l (insert x r)
               Tip y       | x == y        -> Tip x
                           | otherwise     -> join x (Tip x) y t
               Nil                         -> Tip x
```

Функция: `delete`

Описание: удаляет элемент из множества. Если в заданном множестве нет удаляемого элемента, возвращается оригинальное множество.

Определение:

```
delete :: Int -> IntSet -> IntSet
delete x t = case t of
               Bin p m l r | nomatch x p m -> t
                           | zero x m      -> bin p m (delete x l) r
                           | otherwise     -> bin p m l (delete x r)
               Tip y       | x == y        -> Nil
                           | otherwise     -> t
               Nil                         -> Nil
```

Функция: `union`

Описание: возвращает объединение двух множеств.

Определение:

```
union :: IntSet -> IntSet -> IntSet
union t1@(Bin p1 m1 l1 r1) t2@(Bin p2 m2 l2 r2)
    | shorter m1 m2 = union1
    | shorter m2 m1 = union2
    | p1 == p2      = Bin p1 m1 (union l1 l2) (union r1 r2)
    | otherwise     = join p1 t1 p2 t2
  where
    union1 | nomatch p2 p1 m1 = join p1 t1 p2 t2
           | zero p2 m1       = Bin p1 m1 (union l1 t2) r1
           | otherwise        = Bin p1 m1 l1 (union r1 t2)
    union2 | nomatch p1 p2 m2 = join p1 t1 p2 t2
           | zero p1 m2       = Bin p2 m2 (union t1 l2) r2
           | otherwise        = Bin p2 m2 l2 (union t1 r2)
union (Tip x) t = insert x t
union t (Tip x) = insertR x t
union Nil t     = t
union t Nil     = t


insertR :: Int -> IntSet -> IntSet
insertR x t = case t of
                Bin p m l r | nomatch x p m -> join x (Tip x) p t
                            | zero x m      -> Bin p m (insert x l) r
                            | otherwise     -> Bin p m l (insert x r)
                Tip y       | x == y        -> t
                            | otherwise     -> join x (Tip x) y t
```

```
              Nil                          -> Tip x
```

Функция: `unions`

Описание: возвращает объединение списка множеств.

Определение:

```
unions :: [IntSet] -> IntSet
unions xs = foldlStrict union empty xs
```

Функция: `difference`

Описание: возвращает разность двух множеств.

Определение:

```
difference :: IntSet -> IntSet -> IntSet
difference t1@(Bin p1 m1 l1 r1) t2@(Bin p2 m2 l2 r2)
    | shorter m1 m2 = difference1
    | shorter m2 m1 = difference2
    | p1 == p2      = bin p1 m1 (difference l1 l2) (difference r1 r2)
    | otherwise     = t1
  where
    difference1 | nomatch p2 p1 m1 = t1
                | zero p2 m1       = bin p1 m1 (difference l1 t2) r1
                | otherwise        = bin p1 m1 l1 (difference r1 t2)
    difference2 | nomatch p1 p2 m2 = t1
                | zero p1 m2       = difference t1 l2
                | otherwise        = difference t1 r2
difference t1@(Tip x) t2 | member x t2 = Nil
                         | otherwise   = t1
difference Nil t     = Nil
difference t (Tip x) = delete x t
difference t Nil     = t
```

Функция: `intersection`

Описание: возвращает пересечение двух множеств.

Определение:

```
intersection :: IntSet -> IntSet -> IntSet
intersection t1@(Bin p1 m1 l1 r1) t2@(Bin p2 m2 l2 r2)
   | shorter m1 m2 = intersection1
   | shorter m2 m1 = intersection2
   | p1 == p2      = bin p1 m1 (intersection l1 l2) (intersection r1 r2)
   | otherwise     = Nil
  where
```

```
    intersection1 | nomatch p2 p1 m1 = Nil
                  | zero p2 m1       = intersection l1 t2
                  | otherwise        = intersection r1 t2
    intersection2 | nomatch p1 p2 m2 = Nil
                  | zero p1 m2       = intersection t1 l2
                  | otherwise        = intersection t1 r2
intersection t1@(Tip x) t2 | member x t2 = t1
                           | otherwise   = Nil
intersection t (Tip x) = case lookup x t of
                           Just y  -> Tip y
                           Nothing -> Nil
intersection Nil t = Nil
intersection t Nil = Nil
```

Функция: `filter`

Описание: фильтрует заданное множество в соответствии с некоторым предикатом.

Определение:

```
filter :: (Int -> Bool) -> IntSet -> IntSet
filter pred t = case t of
                  Bin p m l r       -> bin p m (filter pred l) (filter pred r)
                  Tip x | pred x    -> t
                        | otherwise -> Nil
                  Nil               -> Nil
```

Функция: `partition`

Описание: возвращает пару подмножеств заданного множества, первое из которых составляют элементы, удовлетворяющие заданному предикату, второе — не удовлетворяющие соответственно.

Определение:

```
partition :: (Int -> Bool) -> IntSet -> (IntSet, IntSet)
partition pred t = case t of
                     Bin p m l r -> let (l1, l2) = partition pred l
                                        (r1, r2) = partition pred r
                                    in  (bin p m l1 r1, bin p m l2 r2)
                     Tip x | pred x    -> (t, Nil)
                           | otherwise -> (Nil, t)
                     Nil               -> (Nil, Nil)
```

Функция: `split`

Описание: разбивает заданное множество на два подмножества, возвращаемые в виде пары, в которой первое подмножество содержит элементы, меньше заданного, а второе подмножество содержит элементы строго больше заданного. Соответственно, если заданный элемент и присутствует в исходном множестве, он не включается в результат.

Определение:

```
split :: Int -> IntSet -> (IntSet,IntSet)
split x t = case t of
              Bin p m l r | m < 0     -> if x >= 0
                                         then let (lt, gt) = split' x l
                                              in  (union r lt, gt)
                                         else let (lt, gt) = split' x r
                                              in  (lt, union gt l)
                          | otherwise -> split' x t
              Tip y | x > y           -> (t, Nil)
                    | x < y           -> (Nil, t)
                    | otherwise       -> (Nil, Nil)
              Nil                     -> (Nil, Nil)


split' :: Int -> IntSet -> (IntSet, IntSet)
split' x t = case t of
               Bin p m l r | match x p m -> if zero x m
                                            then let (lt, gt) = split' x l
                                                 in  (lt, union gt r)
                                            else let (lt, gt) = split' x r
                                                 in (union l lt, gt)
                           | otherwise   -> if x < p
                                            then (Nil, t)
                                            else (t, Nil)
               Tip y | x > y             -> (t, Nil)
                     | x < y             -> (Nil, t)
                     | otherwise         -> (Nil, Nil)
               Nil                       -> (Nil, Nil)
```

Функция: `splitMember`

Описание: вариант функции `split`, который возвращает не только пару подмножеств, но и флаг присутствия заданного элемента в исходном множестве (значение типа `Bool`).

Определение:

```
splitMember :: Int -> IntSet -> (IntSet, Bool, IntSet)
splitMember x t = case t of
                  Bin p m l r | m < 0 -> if x >= 0
                                         then let (lt, found, gt) = splitMember' x l
                                              in  (union r lt, found, gt)
                                         else let (lt, found, gt) = splitMember' x r
                                              in  (lt, found, union gt l)
                              | otherwise -> splitMember' x t
                  Tip y | x > y           -> (t, False, Nil)
                        | x < y           -> (Nil, False, t)
                        | otherwise       -> (Nil, True, Nil)
                  Nil                     -> (Nil, False, Nil)


splitMember' :: Int -> IntSet -> (IntSet, Bool, IntSet)
splitMember' x t = case t of
                   Bin p m l r | match x p m ->
                                   if zero x m
                                     then let (lt, found, gt) = splitMember x l
                                          in  (lt, found, union gt r)
                                     else let (lt, found, gt) = splitMember x r
                                          in  (union l lt, found, gt)
                               | otherwise   -> if x < p
                                                then (Nil, False, t)
                                                else (t, False, Nil)
                   Tip y | x > y             -> (t, False, Nil)
                         | x < y             -> (Nil, False, t)
                         | otherwise         -> (Nil, True, Nil)
                   Nil                       -> (Nil, False, Nil)
```

Функция: `map`

Описание: применяет заданную функцию ко всем элементам множества. Необходимо отметить, что множество, которое получается в результате выполнения, может иметь меньше элементов в тех случаях, когда заданная функция возвращает одинаковый результат для двух разных значений, входящих в исходное множество.

Определение:

```
map :: (Int->Int) -> IntSet -> IntSet
map f = fromList . List.map f . toList
```

Функция: `fold`

Описание: сворачивает заданное множество в одно значение, используя функцию и начальный элемент для свёртки.

Определение:

```
fold :: (Int -> b -> b) -> b -> IntSet -> b
fold f z t = case t of
               Bin 0 m l r | m < 0 -> foldr f (foldr f z l) r
               Bin p m l r         -> foldr f z t
               Tip x               -> f x z
               Nil                 -> z


foldr :: (Int -> b -> b) -> b -> IntSet -> b
foldr f z t = case t of
                Bin p m l r -> foldr f (foldr f z r) l
                Tip x       -> f x z
                Nil         -> z
```

Функция: `elems`

Описание: возвращает список элементов множества.

Определение:

```
elems :: IntSet -> [Int]
elems s = toList s
```

Функция: `toList`

Описание: синоним функции `elems`.

Определение:

```
toList :: IntSet -> [Int]
toList t = fold (:) [] t
```

Функция: `fromList`

Описание: создаёт множество на основе заданного списка целых чисел.

Определение:

```
fromList :: [Int] -> IntSet
fromList xs = foldlStrict ins empty xs
  where
    ins t x  = insert x t
```

Функция: `toAscList`

Описание: создаёт упорядоченный от меньшего к большему список на основе заданного множества.

Определение:

```
toAscList :: IntSet -> [Int]
toAscList t = toList t
```

Функция: `fromAscList`

Описание: создаёт множество на основе списка целых чисел, элементы в котором идут в возрастающем порядке.

Определение:

```
fromAscList :: [Int] -> IntSet
fromAscList xs = fromList xs
```

Функция: `fromDistinctAscList`

Описание: вариант функции `fromAscList`, в который необходимо передавать список, не имеющий повторяющихся элементов.

Определение:

```
fromDistinctAscList :: [Int] -> IntSet
fromDistinctAscList xs = fromList xs
```

Функция: `showTree`

Описание: преобразует дерево, представляющее множество, в строку. Функция обычно используется в целях отладки.

Определение:

```
showTree :: IntSet -> String
showTree s = showTreeWith True False s
```

Функция: `showTreeWith`

Описание: преобразует дерево, представляющее множество, в строку. Более общая функция, чем `showTree`. Первый аргумент отвечает за способ вывода. Если он принимает значение `True`, в строку выводится дерево с отступами, в противном случае — циклическое дерево. Если второй аргумент равен `True`, то при преобразовании не учитывается получаемая ширина подстрок. Функция обычно используется в целях отладки.

Определение:

```
showTreeWith :: Bool -> Bool -> IntSet -> String
showTreeWith hang wide t | hang      = (showsTreeHang wide [] t) ""
                         | otherwise = (showsTree wide [] [] t) ""


showsTree :: Bool -> [String] -> [String] -> IntSet -> ShowS
showsTree wide lbars rbars t
  = case t of
      Bin p m l r -> showsTree wide (withBar rbars) (withEmpty rbars) r .
                     showWide wide rbars .
                     showsBars lbars . showString (showBin p m) . showString "\n" .
                     showWide wide lbars .
                     showsTree wide (withEmpty lbars) (withBar lbars) l
      Tip x       -> showsBars lbars . showString " " . shows x . showString "\n"
      Nil         -> showsBars lbars . showString "|\n"


showsTreeHang :: Bool -> [String] -> IntSet -> ShowS
showsTreeHang wide bars t
  = case t of
      Bin p m l r -> showsBars bars . showString (showBin p m) . showString "\n" .
                     showWide wide bars .
                     showsTreeHang wide (withBar bars) l .
                     showWide wide bars .
                     showsTreeHang wide (withEmpty bars) r
      Tip x       -> showsBars bars . showString " " . shows x . showString "\n"
      Nil         -> showsBars bars . showString "|\n"


showBin p m = "*" -- ++ show (p, m)


showWide wide bars | wide      = showString (concat (reverse bars)) . showString "|\n"
                   | otherwise = id


showsBars :: [String] -> ShowS
showsBars bars = case bars of
                   [] -> id
                   _  -> showString (concat (reverse (tail bars))) . showString node


node           = "+--"
withBar bars   = "|  ":bars
withEmpty bars = "   ":bars
```

8.17. Модуль IORef

Модуль IORef содержит описания программных сущностей, предоставляющих механизмы работы с изменяемыми ссылками в рамках монады IO. Использование:

```
import Data.IORef
```

Все программные сущности, описанные в этом модуле, определены в виде примитивов, поэтому упоминание об этом приводиться не будет.

Тип: IORef

Описание: изменяемая ссылка (переменная) в рамках монады IO.

Определение:

Для данного типа определены экземпляры следующих классов: Typeable, Typeable1 и Eq.

Функция: newIORef

Описание: создаёт новую ссылку.

Определение:

```
newIORef :: a -> IO (IORef a)
```

Функция: readIORef

Описание: возвращает значение заданной ссылки.

Определение:

```
readIORef :: IORef a -> IO a
```

Функция: writeIORef

Описание: записывает новое значение в заданную ссылку.

Определение:

```
writeIORef :: IORef a -> a -> IO ()
```

Функция: modifyIORef

Описание: изменяет значение в заданной ссылке при помощи некоторой функции.

Определение:

```
modifyIORef :: IORef a -> (a -> a) -> IO ()
```

Функция: `atomicModifyIORef`
Описание: атомарно изменяет содержимое ссылки. Эта функция полезна при разработке многопоточных приложений, когда эта функция гарантирует то, что к заданной ссылке имеется только одно обращение в каждый момент времени.
Определение:

```
atomicModifyIORef :: IORef a -> (a -> (a, b)) -> IO b
```

Функция: `mkWeakIORef`
Описание: создаёт слабый указатель на ссылку.
Определение:

```
mkWeakIORef :: IORef a -> IO () -> IO (Weak (IORef a))
```

8.18. Модуль Ix

В модуле `Ix` описан класс, который используется как интерфейс к типам данных, представляющих значения, которые могут использоваться в качестве индексов (индексация сопоставляет некоторое непрерывное множество с набором значений экземпляров описываемого класса). Например, этот класс используется для индексации массивов `IArray` (см. стр. 233) и `MArray` (см. стр. 237). Использование:

```
import Data.Ix
```

Соответственно, в этом модуле описан только сам класс и несколько его экземпляров.

Класс: `Ix`
Описание: класс типов, чьи значения могут использоваться для индексации.
Определение:

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

Во всех методах класса первый аргумент вида `(l, u)` представляет пару чисел, которые являются нижней и верхней границей области индексации. Каждый экземпляр этого класса должен выполнять следующие законы:

1) `inRange (l, u) i == elem i (range (l, u))`

2) `range (l, u) !! index (l, u) i == i, when inRange (l, u) i`

3) `map (index (l, u)) (range (l, u))) == [0..rangeSize (l, u) - 1]`

4) `rangeSize (l, u) == length (range (l, u))`

Метод `range` возвращает список индексов, определённых нижней и верхней границей области индексации. Метод `index` возвращает номер позиции заданного индекса в области индексации (позиции нумеруются с 0). Также и метод `inRange` возвращает значение `True`, если заданное значение индекса лежит в области индексации.

Наконец, метод `rangeSize` возвращает размер области индексации. Этот метод имеет реализацию по умолчанию, а поэтому может не определяться для экземпляров.

Для класса `Ix` определены следующие экземпляры: `Bool`, `Char`, `Day`, `GeneralCategory`, `IOMode`, `Int`, `Int16`, `Int32`, `Int64`, `Int8`, `Integer`, `Month`, `Ordering`, `SeekMode`, `Word`, `Word16`, `Word32`, `Word64` и `Word8`. Кроме того, для кортежей размеров от 0 для 5 также определены экземпляры класса `Ix`.

Для этого класса также могут быть автоматически построены экземпляры. Они могут быть построены только для перечислений либо для алгебраических типов данных с одним конструктором, компоненты которого также являются экземплярами класса `Ix`. Для кортежей произвольного размера также могут быть построены экземпляры класса `Ix` автоматически опять же при условии, что в такие кортежи входят в качестве компонентов только экземпляры этого класса.

8.19. Модуль `List`

Модуль `List` опять же создан в рамках разгрузки стандартного модуля `Prelude` от излишка определений программных сущностей и разнесения таких определений по разным модулям в зависимости от типа данных. Соответственно, в этом модуле собраны определения функций, которые работают со списками. А поскольку подавляющее большинство функций в этом модуле имеют те же

самые названия, что и функции в модуле Prelude, этот модуль необходимо импортировать квалифицированно:

```
import qualified Data.List as List
```

Ну и само собой разумеется, что в этом модуле описаны только функции, поскольку тип данных [] (список) реализован на уровне синтаксиса языка Haskell.

Далее в этом разделе описываются только функции, дополнительные к функциям стандартного модуля Prelude, поскольку все функции для работы со списками, которые определены в Prelude, определены и в рассматриваемом модуле. Перечень функций из стандартного модуля Prelude приведён в главе 6..

Функция: intersperse

Описание: «прорежает» элементы списка заданным символом. Например, если на вход этой функции подан символ ',' и список "abcde", то на выходе получится список "a,b,c,d,e".

Определение:

```
intersperse :: a -> [a] -> [a]
intersperse _   []     = []
intersperse _   [x]    = [x]
intersperse sep (x:xs) = x : sep : intersperse sep xs
```

Функция: transpose

Описание: транспонирует матрицу, представленную в виде списка списков. Например: transpose [[1, 2, 3], [4, 5, 6]] == [[1, 4], [2, 5], [3, 6]].

Определение:

```
transpose :: [[a]] -> [[a]]
transpose []           = []
transpose ([]:xss)     = transpose xss
transpose ((x:xs):xss) = (x:[h | (h:t) <- xss]):transpose (xs:[t | (h:t) <- xss])
```

Функция: mapAccumL

Описание: комбинирует свойства функций map (см. стр. 356) и foldl (см. стр. 253), возвращая пару, состоящую из вычисленного значения аккумулятора и списка, полученного при помощи применения заданной функции к каждому элементу исходного списка.

Определение:

```
mapAccumL :: (acc -> x -> (acc, y)) -> acc -> [x] -> (acc, [y])
mapAccumL _ s []     = (s, [])
mapAccumL f s (x:xs) = (s'', y:ys)
  where
    (s', y) = f s x
    (s'', ys) = mapAccumL f s' xs
```

Функция: `mapAccumR`

Описание: комбинирует свойства функций `map` (см. стр. 356) и `foldr` (см. стр. 254), возвращая пару, состоящую из вычисленного значения аккумулятора и списка, полученного при помощи применения заданной функции к каждому элементу исходного списка.

Определение:

```
mapAccumR :: (acc -> x -> (acc, y)) -> acc -> [x] -> (acc, [y])
mapAccumR _ s []     =  (s, [])
mapAccumR f s (x:xs) =  (s'', y:ys)
  where
    (s'', y) = f s' x
    (s', ys) = mapAccumR f s xs
```

Функция: `unfoldr`

Описание: функция, обратная по своему действию функции `foldr` (см. стр. 254). Разворачивает атомарное значение в список, руководствуясь значением `Nothing` в качестве критерия остановки.

Определение:

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr f b = case f b of
                Just (a, new_b) -> a:unfoldr f new_b
                Nothing         -> []
```

Функция: `group`

Описание: группирует элементы списка по признаку совпадения следующих друг за другом элементов.

Определение:

```
group :: Eq a => [a] -> [[a]]
group = groupBy (==)
```

Функция: groupBy
Описание: обобщение функции group, позволяющее передавать в качестве первого аргумента предикат для группировки.
Определение:

```
groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
groupBy _  []     = []
groupBy eq (x:xs) = (x:ys):groupBy eq zs
  where
    (ys, zs) = span (eq x) xs
```

Функция: inits
Описание: возвращает список всех начал заданного списка. Последним элементом результата будет как раз исходный список.
Определение:

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

Функция: tails
Описание: возвращает список всех окончаний заданного списка. Первым элементом результата будет как раз исходный список.
Определение:

```
tails :: [a] -> [[a]]
tails []         = [[]]
tails xxs@(_:xs) = xxs : tails xs
```

Функция: isPrefixOf
Описание: возвращает значение True, если первый список является началом второго списка (или они совпадают).

Определение:

```
isPrefixOf :: (Eq a) => [a] -> [a] -> Bool
isPrefixOf [] _          = True
isPrefixOf _  []         = False
isPrefixOf (x:xs) (y:ys) = x == y && isPrefixOf xs ys
```

Функция: `isSuffixOf`

Описание: возвращает значение `True`, если первый список является окончанием второго списка (или они совпадают).

Определение:

```
isSuffixOf :: (Eq a) => [a] -> [a] -> Bool
isSuffixOf x y = reverse x `isPrefixOf` reverse y
```

Функция: `isInfixOf`

Описание: возвращает значение `True`, если первый список содержится полностью во втором списке (или они совпадают).

Определение:

```
isInfixOf :: (Eq a) => [a] -> [a] -> Bool
isInfixOf needle haystack = any (isPrefixOf needle) (tails haystack)
```

Функция: `find`

Описание: возвращает первый элемент списка, удовлетворяющий заданному предикату. Если ни одного такого элемента в списке нет, возвращает значение `Nothing`.

Определение:

```
find :: (a -> Bool) -> [a] -> Maybe a
find p = listToMaybe . filter p
```

Функция: `partition`

Описание: разбивает список на два подсписка, в первом из которых содержатся элементы, удовлетворяющие заданному предикату, во втором — не удовлетворяющие соответственно.

Определение:

```
partition :: (a -> Bool) -> [a] -> ([a], [a])
partition p xs = foldr (select p) ([], []) xs
  where
    select p x ~(ts, fs) | p x       = (x:ts, fs)
                         | otherwise = (ts, x:fs)
```

Функция: elemIndex

Описание: возвращает индекс (начиная с 0) первого элемента списка, который равен заданному. Если ничего не найдено, возвращает значение Nothing.

Определение:

```
elemIndex :: Eq a => a -> [a] -> Maybe Int
elemIndex x = findIndex (x ==)
```

Функция: elemIndices

Описание: возвращает список индексов всех вхождений заданного элемента в список.

Определение:

```
elemIndices :: Eq a => a -> [a] -> [Int]
elemIndices x = findIndices (x ==)
```

Функция: findIndex

Описание: обобщение функции elemIndex, позволяющее передавать первым аргументом предикат, при помощи которого ищутся элементы.

Определение:

```
findIndex :: (a -> Bool) -> [a] -> Maybe Int
findIndex p = listToMaybe . findIndices p
```

Функция: findIndices

Описание: обобщение функции elemIndices, позволяющее передавать первым аргументом предикат, при помощи которого ищутся элементы.

Определение:

```
findIndices :: (a -> Bool) -> [a] -> [Int]
findIndices p xs = [i | (x, i) <- zip xs [0..], p x]
```

Набор функций zip3 — zip7 является вариантами функции zip (см. стр. 269) для количества входных списков от 3 до 7 соответственно. Результатом работы будут кортежи размера от 3 до 7.

Также и набор функций zipWith3 — zipWith7 является вариантами функции zipWith (см. стр. 269) для количества входных списков от 3 до 7 соответственно. Результатом работы будут кортежи размера от 3 до 7.

Наконец, набор функций unzip3 — unzip7 являются вариантами функции unzip (см. стр. 270) для размера кортежей во входном списке от 3 до 7 соответственно. Результатом работы будет кортеж размера от 3 до 7.

Функция: nub

Описание: возвращает список, составленный из элементов исходного списка, в котором нет повторяющихся элементов.

Определение:

```
nub :: (Eq a) => [a] -> [a]
nub l = nub' l []
  where
    nub' [] _                    = []
    nub' (x:xs) ls | x `elem` ls = nub' xs ls
                   | otherwise   = x:nub' xs (x:ls)
```

Функция: nubBy

Описание: обобщение функции nub, позволяющее передавать в качестве первого параметра предикат, при помощи которого осуществляется сравнение элементов.

Определение:

```
nubBy :: (a -> a -> Bool) -> [a] -> [a]
nubBy eq l = nubBy' l []
  where
    nubBy' [] _                        = []
    nubBy' (y:ys) xs | elem_by eq y xs = nubBy' ys xs
                     | otherwise       = y:nubBy' ys (y:xs)
```

Функция: delete

Описание: удаляет первое вхождение заданного элемента из списка.

Определение:

```
delete :: (Eq a) => a -> [a] -> [a]
delete = deleteBy (==)
```

Функция: deleteBy

Описание: обобщение функции delete, позволяющее передавать в качестве первого параметра предикат, при помощи которого осуществляется сравнение элементов.

Определение:

```
deleteBy :: (a -> a -> Bool) -> a -> [a] -> [a]
deleteBy _  _ []     = []
deleteBy eq x (y:ys) = if x `eq` y
                         then ys
                         else y:deleteBy eq x ys
```

Функция: deleteFirstBy

Описание: вариант функции deleteBy, в котором элементы второго списка удаляются из первого (первые вхождения).

Определение:

```
deleteFirstsBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]
deleteFirstsBy eq = foldl (flip (deleteBy eq))
```

Функция: (\\)

Описание: возвращает неассоциативную разницу двух списков. В результате применения (xsys) будет список, состоящий из элементов списка xs, из которого удалены первые вхождения каждого элемента списка ys.

Определение:

```
(\\) :: (Eq a) => [a] -> [a] -> [a]
(\\) =  foldl (flip delete)
```

Функция: union

Описание: возвращает объединение двух списков. Элементы из второго списка, которые дублируют элементы первого списка, не включаются в результат. Однако первый список включается в результат без изменений, несмотря на дублирование элементов (если таковое имеется).

Определение:

```
union :: (Eq a) => [a] -> [a] -> [a]
union = unionBy (==)
```

Функция: `unionBy`

Описание: обобщение функции `union`, позволяющее передавать в качестве первого параметра предикат, при помощи которого осуществляется сравнение элементов.

Определение:

```
unionBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]
unionBy eq xs ys = xs ++ foldl (flip (deleteBy eq)) (nubBy eq ys) xs
```

Функция: `intersect`

Описание: возвращает пересечение двух списков, состоящее только из тех элементов, которые входят в оба заданных списка. Однако если в первом списке имеются дублирующиеся элементы, все они попадут в результат.

Определение:

```
intersect :: (Eq a) => [a] -> [a] -> [a]
intersect = intersectBy (==)
```

Функция: `intersectBy`

Описание: обобщение функции `intersect`, позволяющее передавать в качестве первого параметра предикат, при помощи которого осуществляется сравнение элементов.

Определение:

```
intersectBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]
intersectBy eq xs ys = [x | x <- xs, any (eq x) ys]
```

Функция: `insert`

Описание: вставляет заданный элемент в список на позицию, на которой он будет меньше или равен следующему элементу в списке. В частности, если исходный список отсортирован, то и результат окажется отсортированным.

Определение:

```
insert :: Ord a => a -> [a] -> [a]
insert e ls = insertBy (compare) e ls
```

Функция: insertBy

Описание: обобщение функции **insert**, позволяющее передавать в качестве первого параметра предикат, при помощи которого осуществляется сравнение элементов.

Определение:

```
insertBy :: (a -> a -> Ordering) -> a -> [a] -> [a]
insertBy _   x [] = [x]
insertBy cmp x ys@(y:ys') = case cmp x y of
                              GT -> y:insertBy cmp x ys'
                              _  -> x:ys
```

Функция: sortBy

Описание: вариант функции **sort** (см. стр. 270), в который можно передавать в качестве первого параметра предикат, при помощи которого осуществляется сравнение элементов.

Определение:

```
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
sortBy cmp l = mergesort cmp l
```

Функция: maximumBy

Описание: вариант функции **maximum** (см. стр. 311), в который можно передавать в качестве первого параметра предикат, при помощи которого осуществляется сравнение элементов.

Определение:

```
maximumBy :: (a -> a -> Ordering) -> [a] -> a
maximumBy _ []   = error "List.maximumBy: empty list"
maximumBy cmp xs = foldl1 max xs
  where
    max x y = case cmp x y of
                GT -> x
                _  -> y
```

Функция: minimumBy

Описание: вариант функции **minimum** (см. стр. 312), в который можно передавать в качестве первого параметра предикат, при помощи которого осуществляется сравнение элементов.

Определение:

```
minimumBy :: (a -> a -> Ordering) -> [a] -> a
minimumBy _ []   = error "List.minimumBy: empty list"
minimumBy cmp xs = foldl1 min xs
  where
    min x y = case cmp x y of
                GT -> y
                _  -> x
```

Функция: genericLength

Описание: обобщение функции `length` (см. стр. 393), позволяющее возвращать в качестве результата значение типа, являющегося экземпляром класса `Num` (а не только типа `Int`).

Определение:

```
genericLength :: (Num i) => [b] -> i
genericLength []    = 0
genericLength (_:l) = 1 + genericLength l
```

Функция: genericTake

Описание: обобщение функции `take` (см. стр. 395), позволяющее задавать в качестве входного индекса значение произвольного типа, являющегося экземпляром класса `Integral` (а не только типа `Int`).

Определение:

```
genericTake :: (Integral i) => i -> [a] -> [a]
genericTake 0 _              = []
genericTake _ []             = []
genericTake n (x:xs) | n > 0 = x : genericTake (n - 1) xs
genericTake _  _             = error "List.genericTake: negative argument"
```

Функция: genericDrop

Описание: обобщение функции `drop` (см. стр. 395), позволяющее задавать в качестве входного индекса значение произвольного типа, являющегося экземпляром класса `Integral` (а не только типа `Int`).

Определение:

```
genericDrop :: (Integral i) => i -> [a] -> [a]
genericDrop 0 xs             = xs
genericDrop _ []             = []
```

```
genericDrop n (_:xs) | n > 0 = genericDrop (n - 1) xs
genericDrop _ _              = error "List.genericDrop: negative argument"
```

Функция: genericSplitAt

Описание: обобщение функции split (см. стр. 354), позволяющее задавать в качестве входного индекса значение произвольного типа, являющегося экземпляром класса Integral (а не только типа Int).

Определение:

```
genericSplitAt :: (Integral i) => i -> [b] -> ([b], [b])
genericSplitAt 0 xs             = ([], xs)
genericSplitAt _ []             = ([], [])
genericSplitAt n (x:xs) | n > 0 = (x:xs', xs'')
  where
    (xs', xs'') = genericSplitAt (n-1) xs
genericSplitAt _ _              = error "List.genericSplitAt: negative argument"
```

Функция: genericIndex

Описание: обобщение функции index (см. стр. 394), позволяющее задавать в качестве входного индекса значение произвольного типа, являющегося экземпляром класса Integral (а не только типа Int).

Определение:

```
genericIndex :: (Integral a) => [b] -> a -> b
genericIndex (x:_)  0             = x
genericIndex (_:xs) n | n > 0     = genericIndex xs (n-1)
                      | otherwise = error "List.genericIndex: negative argument."
genericIndex _ _                  = error "List.genericIndex: index too large."
```

Функция: genericReplicate

Описание: обобщение функции replicate (см. стр. 259), позволяющее задавать в качестве входного индекса значение произвольного типа, являющегося экземпляром класса Integral (а не только типа Int).

Определение:

```
genericReplicate :: (Integral i) => i -> a -> [a]
genericReplicate n x = genericTake n (repeat x)
```

8.20. Модуль Map

Модуль Map представляет собой расширение модуля IntMap (см. раздел 8.15.), в котором все программные сущности работают с ключами произвольного типа. Такие отображения получаются более общими, нежели отображения с целочисленными ключами.

Поскольку функции из этого модуля очень часто имеют наименования такие же, как в стандартном модуле Prelude, равно как и в других модулях, обычно этот модуль импортируют квалифицированно:

```
import Data.Map (Map)
import qualified Data.Map as Map
```

Реализация отображений, предлагаемая в этом модуле, основана на так называемых деревьях, сбалансированных по размеру (или деревьях ограниченного баланса), как это описано в работах [2, 17].

Подавляющее большинство функций в этом модуле являются обобщениями функций модуля IntMap (и все его функции обобщены здесь) для ключей произвольного типа. Поэтому ниже перечисляются только те функции, которых нет в модуле IntMap; для понимания же того, как работают обобщённые функции, достаточно обратиться с разделу 8.15..

Единственный определённый в рассматриваемом модуле алгебраический тип данных — Map.

Тип: Map
Описание: тип для представления отображения ключей типа k на значения типа a.
Определение:

```
data Map k a
  = Tip
  | Bin !Size !k a !(Map k a) !(Map k a)

type Size = Int
```

Для этого типа определены экземпляры следующих классов: Typeable2, Foldable, Functor, Traversable, Data, Eq, Monoid, Ord, Read и Show.

Функция: `insertWith'`

Описание: вариант функции `insertWith` (см. стр. 328), в котором комбинирующая функция применяется строго.

Определение:

```
insertWith' :: Ord k => (a -> a -> a) -> k -> a -> Map k a -> Map k a
insertWith' f k x m = insertWithKey' (\k x y -> f x y) k x m
```

Функция: `insertWithKey'`

Описание: вариант функции `insertWithKey` (см. стр. 328), в котором комбинирующая функция применяется строго.

Определение:

```
insertWithKey' :: Ord k => (k -> a -> a -> a) -> k -> a -> Map k a -> Map k a
insertWithKey' f kx x t
  = case t of
      Tip             -> singleton kx x
      Bin sy ky y l r -> case compare kx ky of
                           LT -> balance ky y (insertWithKey' f kx x l) r
                           GT -> balance ky y l (insertWithKey' f kx x r)
                           EQ -> let x' = f kx x y in seq x' (Bin sy kx x' l r)
```

Функция: `alter`

Описание: модифицирует значение по заданному ключу. Если ключа не существует в отображении, то функция создаёт его, записывая заданное значение. Сложность — $O(\log n)$, где n — размер отображения.

Определение:

```
alter :: Ord k => (Maybe a -> Maybe a) -> k -> Map k a -> Map k a
alter f k t
  = case t of
      Tip             -> case f Nothing of
                           Nothing -> Tip
                           Just x  -> singleton k x
      Bin sx kx x l r -> case compare k kx of
                           LT -> balance kx x (alter f k l) r
                           GT -> balance kx x l (alter f k r)
                           EQ -> case f (Just x) of
                                   Just x' -> Bin sx kx x' l r
                                   Nothing -> glue l r
```

Функция: mapKeys

Описание: применяет заданную функцию ко всем ключам в отображении. Размер отображения в результате может оказаться меньше, поскольку если функция для применения к ключам даст одинаковые значения на разных входных ключах, в результирующее отображение попадает только первое значение и первый ключ. Сложность — $O(n * \log n)$, где n — размер отображения.

Определение:

```
mapKeys :: Ord k2 => (k1 -> k2) -> Map k1 a -> Map k2 a
mapKeys = mapKeysWith (\x y -> x)
```

Функция: mapKeysWith

Описание: вариант функции **mapKeys**, который применяет первую функцию к значениям в отображении, а вторую — к ключам. Первая функция применяется тогда, когда функция для ключей дала одинаковые выходные ключи на двух разных входных ключах. Сложность — $O(n * \log n)$, где n — размер отображения.

Определение:

```
mapKeysWith :: Ord k2 => (a -> a -> a) -> (k1 -> k2) -> Map k1 a -> Map k2 a
mapKeysWith c f = fromListWith c . List.map fFirst . toList
  where
    fFirst (x, y) = (f x, y)
```

Функция: mapKeysMonotonic

Описание: вариант функции **mapKeys**, который применяется для строго монотонных функций. Данную функцию можно применять тогда, когда известно, что заданная функция для ключей строго монотонна (то есть не проводится проверка на совпадение ключей в результирующем отображении). Проверка на монотонность не производится, а потому разработчик программного обеспечения должен сам гарантировать это свойство у функции. Сложность — $O(n)$, где n — размер отображения.

Определение:

```
mapKeysMonotonic :: (k1 -> k2) -> Map k1 a -> Map k2 a
mapKeysMonotonic f Tip               = Tip
mapKeysMonotonic f (Bin sz k x l r) = Bin sz (f k) x (mapKeysMonotonic f l)
                                                     (mapKeysMonotonic f r)
```

Функция: `lookupIndex`

Описание: возвращает позицию заданного ключа в отображении. Позиция ключа находится в интервале от 0 до размера отображения без единицы. Поскольку ключ может быть не найден в отображении, результат функции заключён в монаду (функция обобщена для произвольной монады). Сложность — $O(\log n)$, где n — размер отображения.

Определение:

```
lookupIndex :: (Monad m,Ord k) => k -> Map k a -> m Int
lookupIndex k t = case lookup 0 t of
                    Nothing -> fail "Data.Map.lookupIndex: Key not found."
                    Just x  -> return x
  where
    lookup idx Tip              = Nothing
    lookup idx (Bin _ kx x l r) = case compare k kx of
                                    LT -> lookup idx l
                                    GT -> lookup (idx + size l + 1) r
                                    EQ -> Just (idx + size l)
```

Функция: `elemAt`

Описание: возвращает пару вида (*ключ*, *значение*) по заданному индексу. Если индекс выходит за границы отображения, вызывается функция `error` (см. стр. 133). Сложность — $O(\log n)$, где n — размер отображения.

Определение:

```
elemAt :: Int -> Map k a -> (k, a)
elemAt i Tip              = error "Map.elemAt: index out of range"
elemAt i (Bin _ kx x l r) = case compare i sizeL of
                              LT -> elemAt i l
                              GT -> elemAt (i - sizeL - 1) r
                              EQ -> (kx, x)
  where
    sizeL = size l
```

Функция: `updateAt`

Описание: применяет заданную функцию к элементу отображения по заданному индексу, записывая результат функции в качестве нового значения по использованному ключу. Если индекс выходит за границы отображения, вызывается функция `error` (см. стр. 133). Сложность — $O(\log n)$, где n — размер отображения.

Определение:

```
updateAt :: (k -> a -> Maybe a) -> Int -> Map k a -> Map k a
updateAt f i Tip                = error "Map.updateAt: index out of range"
updateAt f i (Bin sx kx x l r) = case compare i sizeL of
                                    LT -> updateAt f i l
                                    GT -> updateAt f (i - sizeL - 1) r
                                    EQ -> case f kx x of
                                            Just x' -> Bin sx kx x' l r
                                            Nothing -> glue l r
  where
    sizeL = size l
```

Функция: `deleteAt`

Описание: удаляет элемент из отображения по заданному индексу. Сложность — $O(\log n)$, где n — размер отображения.

Определение:

```
deleteAt :: Int -> Map k a -> Map k a
deleteAt i map = updateAt (\k x -> Nothing) i map
```

Функция: `valid`

Описание: возвращает значение `True`, если внутренняя структура дерева, представляющего отображение, валидна. Данная функция обычно используется в целях отладки. Сложность — $O(n)$, где n — размер отображения.

Определение:

```
valid :: Ord k => Map k a -> Bool
valid t = balanced t && ordered t && validsize t


balanced :: Map k a -> Bool
balanced t = case t of
               Tip              -> True
               Bin sz kx x l r  -> (size l + size r <= 1 ||
                                    (size l <= delta * size r &&
                                     size r <= delta * size l)) &&
                                   balanced l && balanced r
```

```
ordered t = bounded (const True) (const True) t
  where
    bounded lo hi t = case t of
                        Tip             -> True
                        Bin sz kx x l r -> (lo kx) &&
                                           (hi kx) &&
                                           bounded lo (< kx) l &&
                                           bounded (> kx) hi r


validsize t = (realsize t == Just (size t))
  where
    realsize t = case t of
                   Tip             -> Just 0
                   Bin sz kx x l r -> case (realsize l,realsize r) of
                                        (Just n, Just m) | n + m + 1 == sz -> Just sz
                                        other                              -> Nothing
```

Все последующие функции имеют сложность $O(\log n)$, где n — размер отображения, поэтому этот факт указываться не будет. Это связано с тем, что все нижеследующие функции работают с ключами отображения.

Функция: `findMin`

Описание: возвращает пару вида (*ключ*, *значение*) по минимальному ключу в отображении.

Определение:

```
findMin :: Map k a -> (k, a)
findMin (Bin _ kx x Tip r) = (kx, x)
findMin (Bin _ kx x l r)   = findMin l
findMin Tip                = error "Map.findMin: empty map has no minimal element"
```

Функция: `findMax`

Описание: возвращает пару вида (*ключ*, *значение*) по максимальному ключу в отображении.

Определение:

```
findMax :: Map k a -> (k, a)
findMax (Bin _ kx x l Tip) = (kx, x)
findMax (Bin _ kx x l r)   = findMax r
findMax Tip                = error "Map.findMax: empty map has no maximal element"
```

Функция: `deleteMin`

Описание: удаляет из отображения элемент по минимальному ключу.

Определение:

```
deleteMin :: Map k a -> Map k a
deleteMin (Bin _ kx x Tip r) = r
deleteMin (Bin _ kx x l r)   = balance kx x (deleteMin l) r
deleteMin Tip                = Tip
```

Функция: `deleteMax`

Описание: удаляет из отображения элемент по максимальному ключу.

Определение:

```
deleteMax :: Map k a -> Map k a
deleteMax (Bin _ kx x l Tip) = l
deleteMax (Bin _ kx x l r)   = balance kx x l (deleteMax r)
deleteMax Tip                = Tip
```

Функция: `deleteFindMin`

Описание: соединяет свойства функций `findMin` и `deleteMin`, удаляя из отображения значение по минимальному ключу, но при этом возвращая пару, первым элементом которой является удалённое значение в виде пары (*ключ*, *значение*), а вторым — новое отображение.

Определение:

```
deleteFindMin :: Map k a -> ((k, a), Map k a)
deleteFindMin t
  = case t of
      Bin _ k x Tip r -> ((k, x), r)
      Bin _ k x l r   -> let (km, l') = deleteFindMin l
                         in  (km, balance k x l' r)
      Tip             -> (error "Map.deleteFindMin: can not return the minimal element
                                 of an empty map", Tip)
```

Функция: `deleteFindMax`

Описание: соединяет свойства функций `findMax` и `deleteMax`, удаляя из отображения значение по максимальному ключу, но при этом возвращая пару, первым элементом которой является удалённое значение в виде пары (*ключ*, *значение*), а вторым — новое отображение.

Определение:

```
deleteFindMax :: Map k a -> ((k, a), Map k a)
deleteFindMax t
  = case t of
      Bin _ k x l Tip -> ((k, x), l)
      Bin _ k x l r   -> let (km, r') = deleteFindMax r
                         in  (km, balance k x l r')
      Tip             -> (error "Map.deleteFindMax: can not return
                                 the maximal element of an empty map", Tip)
```

Функция: updateMin

Описание: заменяет значение по минимальному ключу в соответствии с заданной функцией.

Определение:

```
updateMin :: (a -> Maybe a) -> Map k a -> Map k a
updateMin f m = updateMinWithKey (\k x -> f x) m
```

Функция: updateMax

Описание: заменяет значение по максимальному ключу в соответствии с заданной функцией.

Определение:

```
updateMax :: (a -> Maybe a) -> Map k a -> Map k a
updateMax f m = updateMaxWithKey (\k x -> f x) m
```

Функция: updateMinWithKey

Описание: вариант функции updateMin, принимающий на вход функцию, которая работает не только со значениями, но и с ключами.

Определение:

```
updateMinWithKey :: (k -> a -> Maybe a) -> Map k a -> Map k a
updateMinWithKey f t = case t of
                         Bin sx kx x Tip r -> case f kx x of
                                                Nothing -> r
                                                Just x' -> Bin sx kx x' Tip r
                         Bin sx kx x l r   -> balance kx x (updateMinWithKey f l) r
                         Tip               -> Tip
```

Функция: `updateMaxWithKey`

Описание: вариант функции `updateMax`, принимающий на вход функцию, которая работает не только со значениями, но и с ключами.

Определение:

```
updateMaxWithKey :: (k -> a -> Maybe a) -> Map k a -> Map k a
updateMaxWithKey f t = case t of
                         Bin sx kx x l Tip -> case f kx x of
                                                Nothing -> l
                                                Just x' -> Bin sx kx x' l Tip
                         Bin sx kx x l r   -> balance kx x l (updateMaxWithKey f r)
                         Tip               -> Tip
```

Функция: `minView`

Описание: вариант функции `deleteFindMin` (см. стр. 398), который заключает результат в монаду. Соответственно, если на вход функции подано пустое дерево, вызывается монадический метод `fail` (см. стр. 211).

Определение:

```
minView :: Monad m => Map k a -> m (Map k a, (k, a))
minView Tip = fail "Map.minView: empty map"
minView x   = return (swap $ deleteFindMin x)
```

Функция: `maxView`

Описание: вариант функции `deleteFindMax` (см. стр. 399), который заключает результат в монаду. Соответственно, если на вход функции подано пустое дерево, вызывается монадический метод `fail` (см. стр. 211).

Определение:

```
maxView :: Monad m => Map k a -> m (Map k a, (k, a))
maxView Tip = fail "Map.maxView: empty map"
maxView x   = return (swap $ deleteFindMax x)
```

В данном модуле описано ещё несколько вспомогательных функций для склейки, балансировки и слияния. Для изучения их работы можно обратиться к исходному коду модуля.

8.21. Модуль Maybe

В модуле Maybe дублируются описания типа Maybe и функции для его обработки. Данный модуль создан в экспериментальном порядке в целях постепенной разгрузки стандартного модуля Prelude. Некоторые определённые в модуле Maybe программные сущности определены и в модуле Prelude. Использование:

```
import Data.Maybe
```

Соответственно, в рассматриваемый модуль вынесены определения: алгебраического типа данных Maybe (см. стр. 109) и функции maybe (см. стр. 145)

Также в модуле Maybe определены экземпляры типа Maybe для следующих классов: Alternative, Applicative, Foldable, Functor, FunctorM, Monad, MonadFix, MonadPlus, Traversable, Typeable1, Data, Eq, Ord, Read и Show.

Функция: isJust

Описание: возвращает значение True, если переданное ей значение обёрнуто конструктором Just.

Определение:

```
isJust :: Maybe a -> Bool
isJust Nothing = False
isJust _       = True
```

Функция: isNothing

Описание: возвращает значение True, если переданное ей значение — Nothing.

Определение:

```
isNothing :: Maybe a -> Bool
isNothing Nothing = True
isNothing _       = False
```

Функция: fromJust

Описание: разворачивает контейнер Maybe, возвращая значение, обёрнутое конструктором Just. Не может работать со значениями типа Nothing.

Определение:

```
fromJust :: Maybe a -> a
fromJust Nothing  = error "Maybe.fromJust: Nothing"
fromJust (Just x) = x
```

Функция: `fromMaybe`

Описание: вариант функции `fromJust`, который может вернуть переданное в качестве первого аргумента значение по умолчанию в случае, если второй аргумент равен `Nothing`.

Определение:

```
fromMaybe :: a -> Maybe a -> a
fromMaybe d x = case x of
                  Nothing -> d
                  Just v  -> v
```

Функция: `listToMaybe`

Описание: конвертирует список в значение типа `Maybe`. Если список пуст, возвращает значение `Nothing`. Если же список не пуст, возвращается голова списка, обёрнутая конструктором `Just`.

Определение:

```
listToMaybe :: [a] -> Maybe a
listToMaybe []    = Nothing
listToMaybe (a:_) = Just a
```

Функция: `maybeToList`

Описание: обратная функция к функции `listToMaybe`. Преобразует значение `Maybe` в список.

Определение:

```
maybeToList :: Maybe a -> [a]
maybeToList Nothing  = []
maybeToList (Just x) = [x]
```

Функция: `catMaybes`

Описание: преобразует список значений типа `Maybe` в список. Все значения `Nothing` игнорируются.

Определение:

```
catMaybes :: [Maybe a] -> [a]
catMaybes ls = [x | Just x <- ls]
```

Функция: mapMaybe
Описание: применяет заданную функцию, возвращающую значение типа Maybe к списку. Результирующий список содержит простые значения.
Определение:

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe _ []     = []
mapMaybe f (x:xs) = let rs = mapMaybe f xs
                    in  case f x of
                          Nothing -> rs
                          Just r  -> r:rs
```

8.22. Модуль Monoid

Служебный модуль Monoid содержит определение класса Monoid и нескольких вспомогательных типов для работы со стандартными типами данных. Этот модуль разработан в соответствии с положениями работы [10]. Использование:

```
import Monoid
```

Главная программная сущность этого модуля — класс Monoid, представляющий собой описание одного из важнейших понятий теории категорий — *моноида*.
Класс: Monoid
Описание: интерфейс к типам данных, которые можно представить в виде моноида (понятие из теории категорий). Все экземпляры этого класса должны безусловно выполнять законы, определённые теорией для моноидов.
Определение:

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
```

Метод mempty является функцией тождества относительно метода mappend. Другими словами, метод mempty возвращает единичный элемент слева относительно операции моноида. Также и метод mappend представляет собой именно операцию моноида, которая должна быть ассоциативной. Эти два метода являются обязательными для определения в любом экземпляре.

Метод mconcat сворачивает список значений при помощи моноида. Для большинства экземпляров можно пользоваться определением этого метода по умолчанию. Однако в случае необходимости можно определить более оптимальную реализацию метода.

Для класса Monoid определены экземпляры следующих типов: All, Any, ByteString (строгий и ленивый варианты), IntSet, Ordering, () (а также кортежи размером от 1 до 5), (->), Dual, Endo, IntMap, Product, Seq, Set, Sum, Map и [].

Тип: Dual

Описание: комоноид (дуальный к моноиду тип), получаемый обращением аргументов метода mappend.

Определение:

```
newtype Dual a
  = Dual {
           getDual :: a
         }
```

Для этого типа определён экземпляр только класса Monoid.

Тип: Endo

Описание: моноид эндоморфизмов над композициями. Ассоциативной операцией моноида является операция композиции (.) для функций. Единицей является функция тождества id (см. стр. 137).

Определение:

```
newtype Endo a
  = Endo {
           appEndo :: (a -> a)
         }
```

Для этого типа определён экземпляр только класса Monoid.

Тип: All

Описание: булевский моноид над конъюнкцией. Ассоциативной операцией является конъюнкция (&&) над булевскими значениями истинности. Единицей — значение True.

Определение:

```
newtype All
  = All {
          getAll :: Bool
        }
```

Для этого типа определены экземпляры следующих типов: Bounded, Eq, Monoid, Ord, Read и Show.

Тип: Any
Описание: булевский моноид над дизъюнкцией. Ассоциативной операцией является дизъюнкция (||) над булевскими значениями истинности. Единицей — значение False.
Определение:

```
newtype Any
  = Any {
          getAny :: Bool
        }
```

Для этого типа определены экземпляры следующих типов: Bounded, Eq, Monoid, Ord, Read и Show.

Тип: Sum
Описание: моноид над сложением. Ассоциативной операцией является операция сложения (+) в полугруппе чисел. Единицей — значение 0.
Определение:

```
newtype Sum a
  = Sum {
          getSum :: a
        }
```

Для этого типа определены экземпляры следующих типов: Bounded, Eq, Monoid, Ord, Read и Show.

Тип: Product
Описание: моноид над умножением. Ассоциативной операцией является операция умножения (*) в полугруппе чисел. Единицей — значение 1.

Определение:

```
newtype Product a
  = Product {
              getProduct :: a
            }
```

Для этого типа определены экземпляры следующих типов: Bounded, Eq, Monoid, Ord, Read и Show.

8.23. Модуль Ord

Модуль Ord предназначен для разгрузки стандартного модуля Prelude. В него вынесено определение класса Ord (см. стр. 121) и определения нескольких десятков экземпляров для этого класса. Использование:

```
import Data.Ord
```

Класс Ord определяет класс типов, в которых имеет смысл отношение порядка. Соответственно, определение этого класса выглядит следующим образом:

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

Соответственно, в рассматриваемом модуле определены экземпляры этого класса для следующих типов: All, Any, ArithException, Array, ArrayException, AsyncException, Bool, BufferMode, BufferState, ByteString, CCc, CChar, CClock, CDev, CDouble, CFloat, CGid, CIno, CInt, CIntMax, CIntPtr, CLDouble, CLLong, CLong, CMode, CNlink, COff, Complex, CPid, CPtrdiff, CRLim, CSChar, CShort, CSigAtomic, CSize, CSpeed, CSsize, CTcflag, CTime, CUChar, CUInt, CUIntMax, CUIntPtr, CULLong, CULong, CUShort, CUid, CWchar, CalendarTime, Char, ClockTime, Constr, ConstrRep, DataRep, Day, Double, Either, Errno, Exception, ExitCode, FDType, Fd, Fixed, Fixity, Float, ForeignPtr, FunPtr, GeneralCategory, Handle, HandlePosn, HashData, IOArray, IOErrorType,

IOException, IOMode, IORef, Inserts, Int, Int16, Int32, Int64, Int8, IntMap, IntPtr, IntSet, Integer, Key, KeyPr, Lexeme, Map, Maybe, Month, MVar, Ordering, PackedString, Permissions, Product, Ptr, Ratio, SeekMode, Seq, Set, StableName, StablePtr, STArray STRef, Sum, ThreadId, TimeDiff, TimeLocale, Tree, TVar, TyCon, TypeRep, UArray, Unique, Version, ViewL, ViewR, Word, Word16, Word32, Word64, Word8, WordPtr и [].

Кроме того, в этом модуле также определены экземпляры класса Eq для кортежей размером от 0 до 14.

Экземпляры этого класса могут быть автоматически определены для произвольных алгебраических типов данных, чьими компонентами являются исключительно экземпляры этого класса. При определении экземпляров вручную можно определить либо метод compare, либо операцию (<=). Остальные методы выражены через эти два. Использование метода compare является более эффективным для сложных типов.

Тип: Ordering

Описание: перечисление, представляющее константы для выражения отношений порядка между значениями, над которыми возможен порядок. Значения этого типа возвращаются функцией comparing.

Определение:

```
data Ordering
  = LT
  | EQ
  | GT
```

Конструктор LT отвечает за отношение «меньше чем». Конструктор EQ отвечает за отношение «равно». И наконец, конструктор GT отвечает за отношение «больше чем».

Функция: comparing

Описание: весьма полезный комбинатор для использования в семействе функций, оканчивающихся на постфикс «By», которые обычно принимают на вход функцию для сравнения.

Определение:

```
comparing :: Ord a => (b -> a) -> b -> b -> Ordering
comparing p x y = compare (p x) (p y)
```

8.24. Модуль Ratio

В модуле Ratio дублируются описания типов Ratio и Rational, а также функций для работы с ними. Данный модуль создан в экспериментальном порядке в целях постепенной разгрузки стандартного модуля Prelude. Все определённые в модуле Ration программные сущности определены и в модуле Prelude. Использование:

```
import Data.Ratio
```

Соответственно, в рассматриваемый модуль вынесены определения: алгебраического типа данных Ratio (см. стр. 112), синонима Rational и функций (%) (см. стр. 170), numerator (см. стр. 148), denominator (см. стр. 130) и approxRational (см. стр. 126).

Также в модуле Ratio определены экземпляры типа Ratio для следующих классов: Typeable1, Data, Enum, Eq, Fractional, NFData, Num, Ord, Read, Real, RealFrac и Show.

8.25. Модуль Sequence

В модуле Sequence определены программные сущности для работы с ограниченными последовательностями — очередями типа FIFO. Поскольку последовательности ограничены, все операции над ними являются строгими, поэтому для многих задач использование последовательностей является более эффективным, нежели использование списков.

Многие функции в этом модуле имеют те же самые наименования, что и функции для работы со списками из стандартного модуля Prelude, поэтому рассматриваемый модуль необходимо импортировать квалифицированно либо вручную скрывать ненужные функции из импорта модуля Prelude. Импортирование модуля:

```
import qualified Data.Sequence as Seq
```

Реализация конечных последовательностей основана на специального вида деревьях, как описано в работе [8].

Тип: Seq

Описание: представление конечных последовательностей для решения разнообразных задач.

Определение:

```
newtype Seq a = Seq (FingerTree (Elem a))


data FingerTree a
  = Empty
  | Single a
  | Deep !Int !(Digit a) (FingerTree (Node a)) !(Digit a)
```

Для этого типа данных определены экземпляры следующих классов: Foldable, Functor, Monad, MonadPlus, Traversable, Typeable1, Data, Eq, Monoid, Ord, Read и Show.

Функция: empty

Описание: создаёт пустую конечную последовательность.

Определение:

```
empty :: Seq a
empty = Seq Empty
```

Функция: singleton

Описание: создаёт последовательность из одного элемента.

Определение:

```
singleton :: a -> Seq a
singleton x = Seq (Single (Elem x))
```

Функция: (<|)

Описание: добавляет в последовательность заданный элемент слева.

Определение:

```
infixr 5 <|


(<|) :: a -> Seq a -> Seq a
x <| Seq xs = Seq (Elem x `consTree` xs)
```

```
consTree :: Sized a => a -> FingerTree a -> FingerTree a
consTree a Empty                        = Single a
consTree a (Single b)                   = deep (One a) Empty (One b)
consTree a (Deep s (Four b c d e) m sf) = m `seq` Deep (size a + s) (Two a b)
                                                       (node3 c d e `consTree` m) sf
consTree a (Deep s (Three b c d) m sf)  = Deep (size a + s) (Four a b c d) m sf
consTree a (Deep s (Two b c) m sf)      = Deep (size a + s) (Three a b c) m sf
consTree a (Deep s (One b) m sf)        = Deep (size a + s) (Two a b) m sf
```

Функция: (|>)

Описание: добавляет в последовательность заданный элемент справа.

Определение:

```
infixr 5 |>


(|>) :: Seq a -> a -> Seq a
Seq xs |> x = Seq (xs `snocTree` Elem x)


snocTree :: Sized a => FingerTree a -> a -> FingerTree a
snocTree Empty a                        = Single a
snocTree (Single a) b                   = deep (One a) Empty (One b)
snocTree (Deep s pr m (Four a b c d)) e = m `seq` Deep (s + size e) pr
                                                       (m `snocTree` node3 a b c)
                                                       (Two d e)
snocTree (Deep s pr m (Three a b c)) d  = Deep (s + size d) pr m (Four a b c d)
snocTree (Deep s pr m (Two a b)) c      = Deep (s + size c) pr m (Three a b c)
snocTree (Deep s pr m (One a)) b        = Deep (s + size b) pr m (Two a b)
```

Функция: (><)

Описание: конкатенирует две последовательности.

Определение:

```
infixr 5 ><


(><) :: Seq a -> Seq a -> Seq a
Seq xs >< Seq ys = Seq (appendTree0 xs ys)
```

```
appendTree0 :: FingerTree (Elem a) -> FingerTree (Elem a) -> FingerTree (Elem a)
appendTree0 Empty xs                                    = xs
appendTree0 xs Empty                                    = xs
appendTree0 (Single x) xs                               = x `consTree` xs
appendTree0 xs (Single x)                               = xs `snocTree` x
appendTree0 (Deep s1 pr1 m1 sf1) (Deep s2 pr2 m2 sf2) = Deep (s1 + s2) pr1
                                                          (addDigits0 m1 sf1 pr2 m2) sf2
```

Дополнительно к этой функции определены ещё несколько схожих, практически не различающихся по своему виду, работающих с разными типами последовательностей. Изучить эти функции можно, обратившись к исходному коду модуля.

Функция: `fromList`

Описание: преобразует конечный список в последовательность.

Определение:

```
fromList :: [a] -> Seq a
fromList = Data.List.foldl' (|>) empty
```

Функция: `null`

Описание: возвращает значение `True`, если заданная последовательность пуста.

Определение:

```
null :: Seq a -> Bool
null (Seq Empty) = True
null _           = False
```

Функция: `length`

Описание: возвращает длину последовательности (количество элементов в ней).

Определение:

```
length :: Seq a -> Int
length (Seq xs) = size xs
```

Тип: `ViewL`

Описание: вид на последовательность с самого левого элемента.

Определение:

```
data ViewL a = EmptyL | a :< Seq a
  deriving (Eq, Ord, Show, Read)
```

Для этого типа определены экземпляры следующих классов: Foldable, Functor, Traversable, Typeable1, Data, Eq, Ord, Read и Show.

Тип: ViewR

Описание: вид на последовательность с самого правого элемента.

Определение:

```
data ViewR a = EmptyR | Seq a :> a
  deriving (Eq, Ord, Show, Read)
```

Для этого типа определены экземпляры следующих классов: Foldable, Functor, Traversable, Typeable1, Data, Eq, Ord, Read и Show.

Функция: viewl

Описание: преобразует последовательность в левонаправленный список.

Определение:

```
viewl :: Seq a -> ViewL a
viewl (Seq xs) = case viewLTree xs of
                   Nothing2           -> EmptyL
                   Just2 (Elem x) xs' -> x :< Seq xs'
```

Функция: viewr

Описание: преобразует последовательность в правонаправленный список.

Определение:

```
viewr :: Seq a -> ViewR a
viewr (Seq xs) = case viewRTree xs of
                   Nothing2           -> EmptyR
                   Just2 xs' (Elem x) -> Seq xs' :> x
```

Функция: index

Описание: возвращает элемент из последовательности по заданному индексу.

Определение:

```
index :: Seq a -> Int -> a
index (Seq xs) i | 0 <= i && i < size xs = case lookupTree i xs of
                                               Place _ (Elem x) -> x
                 | otherwise             = error "index out of bounds"
```

Функция: adjust

Описание: применяет заданную функцию к элементу последовательности по индексу.

Определение:

```
adjust :: (a -> a) -> Int -> Seq a -> Seq a
adjust f i (Seq xs) | 0 <= i && i < size xs = Seq (adjustTree (const (fmap f)) i xs)
                    | otherwise             = Seq xs
```

Функция: `update`

Описание: заменяет элемент в последовательности по заданному индексу.

Определение:

```
update :: Int -> a -> Seq a -> Seq a
update i x = adjust (const x) i
```

Функция: `take`

Описание: возвращает последовательность из первых `i` элементов исходной последовательности.

Определение:

```
take :: Int -> Seq a -> Seq a
take i = fst . splitAt i
```

Функция: `drop`

Описание: возвращает последовательность без первых `i` элементов исходной последовательности.

Определение:

```
drop :: Int -> Seq a -> Seq a
drop i = snd . splitAt i
```

Функция: `splitAt`

Описание: возвращает пару последовательностей, полученных из исходной при помощи разделения её в указанном месте (по индексу).

Определение:

```
splitAt :: Int -> Seq a -> (Seq a, Seq a)
splitAt i (Seq xs) = (Seq l, Seq r)
  where
    (l, r) = split i xs
```

```
split :: Int -> FingerTree (Elem a) -> (FingerTree (Elem a), FingerTree (Elem a))
split i Empty            = i `seq` (Empty, Empty)
split i xs | size xs > i = (l, consTree x r)
           | otherwise   = (xs, Empty)
  where
    Split l x r = splitTree i xs
```

Функция: `reverse`
Описание: обращает заданную последовательность.
Определение:

```
reverse :: Seq a -> Seq a
reverse (Seq xs) = Seq (reverseTree id xs)
```

В данном модуле описано ещё несколько вспомогательных функций и типов для различных утилитарных действий. Нет никакой надобности во включении описания этих программных сущностей в справочник, поскольку разработчик программного обеспечения вряд ли будет пользоваться ими в своих программах (тем более, что они даже не экспортируются из модуля). Для изучения их работы можно обратиться к исходному коду модуля.

Остаётся отметить, что за этот модуль в поставке языка Haskell отвечает Р. Патерсон, с которым можно связаться по адресу электронной почты `ross@soi.city.ac.uk`.

8.26. Модуль `Set`

Модуль `Set` представляет собой расширение модуля `IntSet` (см. раздел 8.16.), в котором все программные сущности работают с ключами произвольного типа. Такие множества получаются более общими, нежели множества с целочисленными ключами.

Поскольку функции из этого модуля очень часто имеют наименования такие же, как в стандартном модуле `Prelude`, равно как и в других модулях, обычно этот модуль импортируют квалифицированно:

```
import Data.Set (Set)
import qualified Data.Set as Set
```

Реализация множеств, предлагаемая в этом модуле, основана на так называемых деревьях, сбалансированных по размеру (или деревьях ограниченного баланса), как это описано в работах [2, 17].

Подавляющее большинство функций в этом модуле являются обобщениями функций модуля IntSet (и все его функции обобщены здесь) для ключей произвольного типа. Поэтому ниже перечисляются только те функции, которых нет в модуле IntSet; для понимания же того, как работают обобщённые функции, достаточно обратиться с разделу 8.16..

Единственный определённый в рассматриваемом модуле алгебраический тип данных — Set.

Тип: Set
Описание: тип для представления множеств.
Определение:

```
data Set a
  = Tip
  | Bin !Size a !(Set a) !(Set a)

type Size = Int
```

Для данного типа определены следующие экземпляры: Foldable, Typeable1, Data, Eq, Monoid, Ord, Read и Show.

Функция: mapMonotonic
Описание: вариант функции map, который применяется для строго монотонных функций. Данную функцию можно применять тогда, когда известно, что заданная функция для преобразования значений множества строго монотонна. Проверка на монотонность не производится, а потому разработчик программного обеспечения должен сам гарантировать это свойство у функции.
Определение:

```
mapMonotonic :: (a -> b) -> Set a -> Set b
mapMonotonic f Tip            = Tip
mapMonotonic f (Bin sz x l r) = Bin sz (f x) (mapMonotonic f l) (mapMonotonic f r)
```

Функция: findMin
Описание: возвращает минимальный элемент множества.

Определение:

```
findMin :: Set a -> a
findMin (Bin _ x Tip r) = x
findMin (Bin _ x l r)   = findMin l
findMin Tip             = error "Set.findMin: empty set has no minimal element"
```

Функция: `findMax`

Описание: возвращает максимальный элемент множества.

Определение:

```
findMax :: Set a -> a
findMax (Bin _ x l Tip) = x
findMax (Bin _ x l r)   = findMax r
findMax Tip             = error "Set.findMax: empty set has no maximal element"
```

Функция: `deleteMin`

Описание: удаляет из множества минимальный элемент.

Определение:

```
deleteMin :: Set a -> Set a
deleteMin (Bin _ x Tip r) = r
deleteMin (Bin _ x l r)   = balance x (deleteMin l) r
deleteMin Tip             = Tip
```

Функция: `deleteMax`

Описание: удаляет из множества максимальный элемент.

Определение:

```
deleteMax :: Set a -> Set a
deleteMax (Bin _ x l Tip) = l
deleteMax (Bin _ x l r)   = balance x l (deleteMax r)
deleteMax Tip             = Tip
```

Функция: `deleteFindMin`

Описание: соединяет свойства функций `findMin` и `deleteMin`, удаляя из множества минимальный элемент, но при этом возвращая пару, первым элементом которой является удалённый из множества, а вторым — новое множество.

Определение:

```
deleteFindMin :: Set a -> (a, Set a)
deleteFindMin t = case t of
                    Bin _ x Tip r -> (x, r)
                    Bin _ x l r   -> let (xm, l') = deleteFindMin l
                                     in  (xm, balance x l' r)
                    Tip           -> (error "Set.deleteFindMin: can not return
                                             the minimal element of an empty set", Tip)
```

Функция: `deleteFindMax`

Описание: соединяет свойства функций `findMax` и `deleteMax`, удаляя из множества максимальный элемент, но при этом возвращая пару, первым элементом которой является удалённый из множества, а вторым — новое множество.

Определение:

```
deleteFindMax :: Set a -> (a, Set a)
deleteFindMax t = case t of
                    Bin _ x l Tip -> (x, l)
                    Bin _ x l r   -> let (xm, r') = deleteFindMax r
                                     in  (xm, balance x l r')
                    Tip           -> (error "Set.deleteFindMax: can not return
                                             the maximal element of an empty set", Tip)
```

Функция: `minView`

Описание: вариант функции `deleteFindMin` (см. стр. 398), который заключает результат в монаду. Соответственно, если на вход функции подано пустое дерево, вызывается монадический метод `fail` (см. стр. 211).

Определение:

```
minView :: Monad m => Set a -> m (Set a, a)
minView Tip = fail "Set.minView: empty set"
minView x   = return (swap $ deleteFindMin x)
```

Функция: `maxView`

Описание: вариант функции `deleteFindMax` (см. стр. 399), который заключает результат в монаду. Соответственно, если на вход функции подано пустое дерево, вызывается монадический метод `fail` (см. стр. 211).

Определение:

```
maxView :: Monad m => Set a -> m (Set a, a)
maxView Tip = fail "Set.maxView: empty set"
maxView x   = return (swap $ deleteFindMax x)
```

Функция: `valid`

Описание: возвращает значение `True`, если внутренняя структура дерева, представляющего множество, валидна. Данная функция обычно используется в целях отладки.

Определение:

```
valid :: Ord a => Set a -> Bool
valid t = balanced t && ordered t && validsize t


balanced :: Set a -> Bool
balanced t = case t of
               Tip           -> True
               Bin sz x l r  -> (size l + size r <= 1 ||
                                (size l <= delta * size r &&
                                 size r <= delta * size l)) &&
                                balanced l && balanced r


ordered t = bounded (const True) (const True) t
  where
    bounded lo hi t = case t of
                        Tip          -> True
                        Bin sz x l r -> (lo x) && (hi x) &&
                                        bounded lo (< x) l &&
                                        bounded (> x) hi r


validsize t = (realsize t == Just (size t))
  where
    realsize t = case t of
                   Tip          -> Just 0
                   Bin sz x l r -> case (realsize l, realsize r) of
                                     (Just n, Just m) | n + m + 1 == sz -> Just sz
                                     other                              -> Nothing
```

В данном модуле описано ещё несколько вспомогательных функций для склейки, балансировки и слияния множеств. Для изучения их работы можно обратиться к исходному коду модуля.

8.27. Модуль STRef

В модуле `STRef` описаны изменяемые ссылки в строгой монаде `ST` (см. подраздел 7.5.3.). Использование:

```
import Data.STRef
```

Главным (и единственным) алгебраическим типом данных в этом модуле является тип `STref`.

Тип: `STRef`

Описание: значение `STRef s a` является изменяемой ссылкой в состоянии `s`, содержащей некоторое значение `a`.

Определение:

```
data STRef s a = ...
```

Тип определён в виде примитива.

Для данного типа определены экземпляры следующих классов: `Typeable2` и `Eq`.

Функция: `newSTRef`

Описание: создаёт новую ссылку с заданным значением в текущем потоке состояний.

Определение:

```
newSTRef :: a -> ST s (STRef s a)
```

Функция определена в виде примитива.

Функция: `readSTRef`

Описание: возвращает значение ссылки.

Определение:

```
readSTRef :: STRef s a -> ST s a
```

Функция определена в виде примитива.

Функция: `writeSTRef`

Описание: записывает новое значение в ссылку.

Определение:

```
writeSTRef :: STRef s a -> a -> ST s ()
```

Функция определена в виде примитива.

Функция: `modifySTRef`

Описание: изменяет значение в ссылке при помощи заданной функции.

Определение:

```
modifySTRef :: STRef s a -> (a -> a) -> ST s ()
modifySTRef ref f = writeSTRef ref . f =<< readSTRef ref
```

8.27.1. Модуль Lazy

Модуль `Lazy` содержит те же самые определения, что и модуль `STRef`, за исключением того, что он поддерживает отложенные вычисления. Поскольку он является «зависимым» модулем от `STRef`, его импорт должен выглядеть следующим образом:

```
import Data.STRef.Lazy
```

Никаких иных программных сущностей, кроме одноимённых в модуле `STRef`, в рассматриваемом модуле не описано.

8.27.2. Модуль Strict

В целях единообразия (по отношению к модулю `Lazy`) в стандартной поставке библиотек языка Haskell имеется модуль `Strict`, который является «подчинённым» модулю `STRef`. Этот модуль всего лишь реимпортирует сам модуль `STRef`, не добавляя никаких новых определений.

8.28. Модуль Traversable

В модуле `Traversable` описан класс типов данных, которые могут быть поэлементно перебраны слева направо, при этом над элементами могут совершаться дополнительные действия. Этот класс описывает специальный вид функторов, как это определено в работах [15, 6]. Использование:

```
import Data.Traversable
```

Необходимо отметить, что в этом модуле определены функции `mapM` и `sequence`, которые также определены в стандартном модуле `Prelude`, поэтому

необходимо либо скрывать стандартные функции, либо импортировать рассматриваемый модуль квалифицированно.

Главная программная сущность, описанная в этом модуле, — класс Traversable.

Класс: Traversable

Описание: функтор, представляющий структуры данных, которые могут быть перебраны слева направо. В экземплярах обязательными для определения являются метод traverse либо метод sequenceA. Остальные методы имеют определение по умолчанию.

Определение:

```
class (Functor t, Foldable t) => Traversable t where
  traverse  :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  mapM      :: Monad m => (a -> m b) -> t a -> m (t b)
  sequence  :: Monad m => t (m a) -> m (t a)
```

Метод traverse позволяет обойти структуру данных слева направо, преобразуя значения из неё в действия. Действия выполняются по мере обхода, результат собирается. Окончательный результат возвращается методом. Также и метод sequenceA выполняет каждое действие в структуре, собирая и затем возвращая результат.

Метод mapM преобразует каждое действие внутри структуры в монадическое действие (для заданной монады), выполняет все действия слева направо и возвращает результат выполнения. Также и метод sequence выполняет слева направо уже монадические в рамках структуры действия, собирает и возвращает результат выполнения.

Экземплярами этого класса являются следующие типы данных: Digit, Elem, FingerTree, Maybe, Node, Seq, Tree, ViewL, ViewR, Array, Map и [].

Функция: for

Описание: вариант метода traverse, к которого порядок следования аргументов обратный.

Определение:

```
for :: (Traversable t, Applicative f) => t a -> (a -> f b) -> f (t b)
for = flip traverse
```

Функция: `forM`
Описание: вариант метода `mapM`, у которого порядок следования аргументов обратный.
Определение:

```
forM :: (Traversable t, Monad m) => t a -> (a -> m b) -> m (t b)
forM = flip mapM
```

Функция: `fmapDefault`
Описание: эта функция может быть использована в качестве аргумента для метода `fmap` в экземплярах класса `Functor` (см. стр. 211).
Определение:

```
fmapDefault :: Traversable t => (a -> b) -> t a -> t b
fmapDefault f = getId . traverse (Id . f)
```

Функция: `foldMapDefault`
Описание: эта функция может быть использована в качестве аргумента для метода `foldMap` в экземплярах класса `Foldable` (см. стр. 306).
Определение:

```
foldMapDefault :: (Traversable t, Monoid m) => (a -> m) -> t a -> m
foldMapDefault f = getConst . traverse (Const . f)
```

Данный модуль является обновлённой и более совершенной версией модуля `FunctorM`, который в свою очередь объявлен устаревшим и выводится из состава стандартных библиотек языка Haskell. Совместно с модулем `Foldable` (см. раздел 8.11.) этот модуль предоставляет всю функциональность (и даже больше), которую предоставлял модуль `FunctorM`.

Остаётся отметить, что за этот модуль в поставке языка Haskell отвечает Р. Патерсон, с которым можно связаться по адресу электронной почты `ross@soi.city.ac.uk`.

8.29. Модуль `Tree`

Модуль `Tree` содержит описание произвольных деревьев (розовых кустов) для произвольных нужд. В модуле предлагается наиболее общий интерфейс для работы с деревьями. Использование:

```
import Data.Tree
```

Главный тип данных в этом модуле: `Tree`.

Тип: `Tree`

Описание: тип для представления розовых кустов, то есть деревьев с произвольным количеством ветвей, выходящих из каждой вершины.

Определение:

```
data Tree a
  = Node
    {
      rootLabel :: a
      subForest :: (Forest a)
    }

type Forest a = [Tree a]
```

Данный тип является экземпляром следующих классов: `Foldable`, `Functor`, `Traversable`, `Typeable1`, `Data`, `Eq`, `Read` и `Show`.

Функция: `drawTree`

Описание: преобразует дерево в красиво выглядящий вид для вывода на экран или в файл.

Определение:

```
drawTree :: Tree String -> String
drawTree = unlines . draw
```

Функция: `drawForest`

Описание: преобразует лес (список деревьев) в красиво выглядящий вид для вывода на экран или в файл.

Определение:

```
drawForest :: Forest String -> String
drawForest = unlines . map drawTree


draw :: Tree String -> [String]
draw (Node x ts0) = x : drawSubTrees ts0
```

```
  where
    drawSubTrees []     = []
    drawSubTrees [t]    = "|" : shift "`- " "   " (draw t)
    drawSubTrees (t:ts) = "|" : shift "+- " "|  " (draw t) ++ drawSubTrees ts
    shift first other   = zipWith (++) (first : repeat other)
```

Функция: `flatten`

Описание: возвращает список меток заданного дерева.

Определение:

```
flatten :: Tree a -> [a]
flatten t = squish t []
  where
    squish (Node x ts) xs = x:Prelude.foldr squish xs ts
```

Функция: `levels`

Описание: возвращает список списков меток вершин на каждом уровне заданного дерева.

Определение:

```
levels :: Tree a -> [[a]]
levels t = map (map rootLabel) $
           takeWhile (not . null) $
           iterate (concatMap subForest) [t]
```

Функция: `unfoldTree`

Описание: создаёт дерево при помощи функции построения и начального значения для неё.

Определение:

```
unfoldTree :: (b -> (a, [b])) -> b -> Tree a
unfoldTree f b = let (a, bs) = f b
                 in  Node a (unfoldForest f bs)
```

Функция: `unfoldForest`

Описание: создаёт лес (список деревьев) при помощи функции построения и начального значения для неё.

Определение:

```
unfoldForest :: (b -> (a, [b])) -> [b] -> Forest a
unfoldForest f = map (unfoldTree f)
```

Функция: unfoldTreeM

Описание: монадический построитель дерева при помощи заданной функции и начального значения для неё (дерево строится по принципу «сначала в глубину»).

Определение:

```
unfoldTreeM :: Monad m => (b -> m (a, [b])) -> b -> m (Tree a)
unfoldTreeM f b = do (a, bs) <- f b
                     ts <- unfoldForestM f bs
                     return (Node a ts)
```

Функция: unfoldForestM

Описание: монадический построитель леса (списка деревьев) при помощи заданной функции и начального значения для неё (лес строится по принципу «сначала в глубину»).

Определение:

```
unfoldForestM :: Monad m => (b -> m (a, [b])) -> [b] -> m (Forest a)
unfoldForestM f = Prelude.mapM (unfoldTreeM f)
```

Функция: unfoldTreeM-BF

Описание: вариант функции unfoldTreeM, который строит дерево по принципу «сначала в ширину» (работа функции основана на материалах статьи [18]).

Определение:

```
unfoldTreeM_BF :: Monad m => (b -> m (a, [b])) -> b -> m (Tree a)
unfoldTreeM_BF f b = liftM getElement $ unfoldForestQ f (singleton b)
  where
    getElement xs = case viewl xs of
                      x :< _ -> x
                      EmptyL -> error "unfoldTreeM_BF"
```

Функция: unfoldForestM-BF

Описание: вариант функции unfoldForestM, который строит лес по принципу «сначала в ширину» (работа функции основана на материалах статьи [18]).

Определение:

```
unfoldForestM_BF :: Monad m => (b -> m (a, [b])) -> [b] -> m (Forest a)
unfoldForestM_BF f = liftM toList . unfoldForestQ f . fromList


unfoldForestQ :: Monad m => (b -> m (a, [b])) -> Seq b -> m (Seq (Tree a))
unfoldForestQ f aQ = case viewl aQ of
                       EmptyL -> return empty
                       a :< aQ -> do (b, as) <- f a
                                     tQ <- unfoldForestQ f (Prelude.foldl (|>) aQ as)
                                     let (tQ', ts) = splitOnto [] as tQ
                                     return (Node b ts <| tQ')
  where
    splitOnto :: [a'] -> [b'] -> Seq a' -> (Seq a', [a'])
    splitOnto as [] q = (q, as)
    splitOnto as (_:bs) q = case viewr q of
                              q' :> a -> splitOnto (a:as) bs q'
                              EmptyR  -> error "unfoldForestQ"
```

8.30. Модуль Tuple

В модуле `Tuple` дублируются описания функций для обработки кортежей. Данный модуль создан в экспериментальном порядке в целях постепенной разгрузки стандартного модуля `Prelude`. Все определённые в модуле `Tuple` программные сущности определены и в модуле `Prelude`. Использование:

```
import Data.Tuple
```

Соответственно, в рассматриваемый модуль вынесены определения функций: `fst` (см. стр. 136), `snd` (см. стр. 163), `curry` (см. стр. 129) и `uncurry` (см. стр. 166).

8.31. Модуль Typeable

Модуль `Typeable` предоставляет интерфейс для безопасного преобразования типов данных между собой. Это достигается при помощи приписывания типам некоторой метаинформации, которая описывает сам тип. Такие описания типов позволяют достаточно эффективно сравнивать типы данных, что позволяет использовать такие типы в модуле `Dynamic` (см. раздел 8.7.). Использование:

```
import Data.Typeable
```

Главная программная сущность в этом модуле — класс `Typeable`. Впрочем, вместе с этим классом определены ещё 7 классов от `Typeable1` до `Typeable7`, все из которых имеют одинаковое предназначение. Поэтому далее будет описан только класс `Typeable`.

Класс: `Typeable`

Описание: интерфейс, который позволяет получить представление типа, являющегося экземпляром этого класса.

Определение:

```
class Typeable a where
  typeOf :: a -> TypeRep
```

Единственный метод `typeOf` получает на вход значение определённого типа и возвращает представление этого типа в виде значения типа `TypeRep` (см. ниже). Любое значение, переданное в этот метод, игнорируется, поэтому в него можно передавать значение (`undefined :: a`).

В качестве экземпляра этого класса в стандартной поставке определены все типы данных, конструкторы которых не параметризуют никаких типов. Соответственно, экземплярами классов `Typeable1` — `Typeable7` являются типы, определённые в стандартной поставке, у которых в конструкторах от 1 до 7 переменных типов. Для тех алгебраических типов данных, у которых имеется несколько конструкторов, для вычисления количества параметризуемых типов используется тот конструктор, у которого максимальное число таких типов. Например, тип `Maybe` (см. стр. 109) является экземпляром класса `Typeable1`, поскольку его конструктор `Just` параметризует один тип, а конструктор `Nothing` не параметризует никаких типов.

Функция: `cast`

Описание: безопасное преобразование типов.

Определение:

```
cast :: (Typeable a, Typeable b) => a -> Maybe b
cast x = r
  where
    r = if typeOf x == typeOf (fromJust r)
          then Just $ unsafeCoerce x
          else Nothing
```

Функция: gcast

Описание: вариант функции cast, осуществляющий преобразование в рамках конструктора некоторого типа.

Определение:

```
gcast :: (Typeable a, Typeable b) => c a -> Maybe (c b)
gcast x = r
  where
    r = if typeOf (getArg x) == typeOf (getArg (fromJust r))
          then Just $ unsafeCoerce x
          else Nothing
    getArg :: c x -> x
    getArg = undefined
```

Функция: gcast1

Описание: вариант функции gcast для конструкторов сорта * -> *.

Определение:

```
gcast1 :: (Typeable1 t, Typeable1 t') => c (t a) -> Maybe (c (t' a))
gcast1 x = r
  where
    r = if typeOf1 (getArg x) == typeOf1 (getArg (fromJust r))
          then Just $ unsafeCoerce x
          else Nothing
    getArg :: c x -> x
    getArg = undefined
```

Функция: gcast2

Описание: вариант функции gcast для конструкторов сорта * -> * -> *.

Определение:

```
gcast2 :: (Typeable2 t, Typeable2 t') => c (t a b) -> Maybe (c (t' a b))
gcast2 x = r
  where
    r = if typeOf2 (getArg x) == typeOf2 (getArg (fromJust r))
          then Just $ unsafeCoerce x
          else Nothing
    getArg :: c x -> x
    getArg = undefined
```

Тип: TypeRep

Описание: конкретное (в противоположность абстрактному) представление мономорфного типа. Это представление поддерживает весьма эффективное сравнение.

Определение:

Тип определён в виде примитива.

Этот тип данных является экземпляром следующих классов: Data, Eq, Show и Typeable.

Тип: TyCon

Описание: абстрактное представление конструктора типа. Объекты этого типа могут быть создны при помощи функции mkTyCon.

Определение:

Тип определён в виде примитива.

Этот тип данных является экземпляром следующих классов: Data, Eq, Show и Typeable.

Функция: mkTyCon

Описание: создаёт представление конструктора типа с заданным именем. Имя конструктора должно быть уникальным в рамках программы, поэтому рекомендуется пользоваться полностью квалифицированными именами.

Определение:

```
mkTyCon :: String -> TyCon
mkTyCon str = TyCon (mkTyConKey str) str
```

Функция: mkTyConApp

Описание: применяет конструктор к последовательности типов.

Определение:

```
mkTyConApp  :: TyCon -> [TypeRep] -> TypeRep
mkTyConApp tc@(TyCon tc_k _) args = TypeRep (appKeys tc_k arg_ks) tc args
  where
    arg_ks = [k | TypeRep k _ _ <- args]
```

Функция: mkAppTy

Описание: добавляет аргумент типа TypeRep в описание типа.

Определение:

```
mkAppTy :: TypeRep -> TypeRep -> TypeRep
mkAppTy (TypeRep tr_k tc trs) arg_tr = let (TypeRep arg_k _ _) = arg_tr
                                       in  TypeRep (appKey tr_k arg_k) tc (trs ++ [arg_tr])
```

Функция: mkFunTy

Описание: специальный вариант функции mkTyConApp, который применяет конструктор функционального типа к заданным типам.

Определение:

```
mkFunTy  :: TypeRep -> TypeRep -> TypeRep
mkFunTy f a = mkTyConApp funTc [f, a]
```

Функция: splitTyConApp

Описание: возвращает пару, полученную при помощи разделения конструктора от набора параметризуемых им типов.

Определение:

```
splitTyConApp :: TypeRep -> (TyCon,[TypeRep])
splitTyConApp (TypeRep _ tc trs) = (tc, trs)
```

Функция: funResultTy

Описание: применяет тип к функциональному типу. Возвращает значение Just u, если первый аргумент представляет функциональный тип t -> u, а второй — значение типа t. Если это не так, возвращает значение Nothing.

Определение:

```
funResultTy :: TypeRep -> TypeRep -> Maybe TypeRep
funResultTy trFun trArg = case splitTyConApp trFun of
                            (tc, [t1, t2]) | tc == funTc && t1 == trArg -> Just t2
                            _                                           -> Nothing
```

Функция: typeRepTyCon

Описание: возвращает представление конструктора из представления типа данных.

Определение:

```
typeRepTyCon :: TypeRep -> TyCon
typeRepTyCon (TypeRep _ tc _) = tc
```

Функция: typeRepArgs
Описание: возвращает список аргументов, которые суть представления типов, применяемый к конструктору.
Определение:

```
typeRepArgs :: TypeRep -> [TypeRep]
typeRepArgs (TypeRep _ _ args) = args
```

Функция: tyConString
Описание: возвращает строковое представление (наименование) конструктора.
Определение:

```
tyConString :: TyCon -> String
tyConString  (TyCon _ str) = str
```

Функция: typeRepKey
Описание: возвращает уникальное значение типа Int, которое используется для внутренних нужд при представлении конструкторов. Обычно этот ключ используется для всевозможных отображений. Программной реализацией гарантируется, что если два ключа равны, то равны и сами конструкторы. В монаду IO результат обрамлён потому, что результат функции может отличаться для одного и того же типа (недетерминированность) в зависимости от запуска программы.
Определение:

Функция определена в виде примитива.

Функция: typeOfDefault
Описание: помогает создать экземпляр класса Typeable при помощи экземпляра класса Typeable1.
Определение:

```
typeOfDefault :: (Typeable1 t, Typeable a) => t a -> TypeRep
typeOfDefault x = typeOf1 x `mkAppTy` typeOf (argType x)
  where
    argType :: t a -> a
    argType = undefined
```

Также в этом модуле определены функции от typeOf1Default до typeOf6Default, которые выполняют те же самые действия, что и функция typeOfDefault, только для экземпляров классов Typeable1 — Typeable6.

8.32. Модуль Unique

Модуль Unique предоставляет разработчикам программного обеспечения абстрактный интерфейс к генераторам уникальных объектов. Использование:

```
import Data.Unique
```

Для представления уникальных объектов используется изоморфный тип Unique.

Тип: Unique
Описание: абстрактное представление уникальных объектов, которые можно сравнивать друг с другом в рамках классов Eq и Ord. Кроме того, эти объекты можно хешировать в целые числа.
Определение:

```
newtype Unique = Unique Integer
  deriving (Eq, Ord)
```

Функция: newUnique
Описание: возвращает уникальный объект. Полученный при помощи данной функции объект никогда не будет равен любому предыдущему объекту, полученному при помощи вызова этой же функции. Нет никаких ограничений на количество вызовов этой функции.
Определение:

```
newUnique :: IO Unique
newUnique = do val <- takeMVar uniqSource
               let next = val + 1
               putMVar uniqSource next
               return (Unique next)
```

Функция: hashUnique
Описание: преобразует уникальный объект в значение типа Int. Два различных уникальных объекта теоретически могут преобразоваться в одно и то же значение типа Int, хотя это маловероятно на практике. Возвращаемое этой функцией значение является хорошим ключом для хеш-таблиц.

Определение:

```
hashUnique :: Unique -> Int
hashUnique (Unique u) = fromInteger (u `mod` (toInteger (maxBound :: Int) + 1))
```

8.33. Модуль Version

Модуль **Version** является общим модулем для решения вопросов генерации и представления номеров версий программного обеспечения (либо чего-нибудь ещё, что может иметь версии). Использование:

```
import Data.Version
```

Различных схем наименования версий существует много. Поэтому способ представления номеров версий, предлагаемый в этом модуле, является компромиссом между произвольной схемой, для которой невозможно предусмотреть какую-либо функциональность, и фиксированной схемой, которая может быть достаточно ограниченной.

Таким образом, в этом модуле предлагается схема наименования номеров версий, которая обобщает многие схемы. К тому же к ней предлагается возможность сравнения, установления порядка версий, а также проведения преобразований в строку и из строки.

Тип: Version

Описание: тип для представления версий. Хранит дерево подверсий, а также произвольное количество меток к версии.

Определение:

```
data Version
  = Version
    {
      versionBranch :: [Int]
      versionTags   :: [String]
    }
```

Элемент **versionBranch** представляет собой список целых чисел, представляющих версию. Обычно версии нумеруются при помощи древовидной структуры. Существует главная ветвь разработки, от которой могут отходить побочные ветви. Например, первая побочная ветвь третьей версии имеет номер «3.1» и т. д. Та-

кие ветви представляются списком целых чисел в непосредственном лексикографическом порядке. Поэтому версия «3.5.14» представляется списком `[3, 5, 14]`.

Также версии могут быть помечены произвольным количеством строковых меток. Для хранения таких меток существует элемент `versionTags`. Интерпретация этих меток возложена на тех, кто использует этот тип данных для работы с номерами версий.

Для типа `Version` определены экземпляры следующих классов: `Eq`, `Ord`, `Read`, `Show` и `Typeable`.

Функция: `showVersion`

Описание: преобразует версию в строковое представление. Например, для версии с номером `[1, 2, 3]` и метками `["tag1", "tag2"]` строковое представление будет `"1.2.3-tag1-tag2"`.

Определение:

```
showVersion :: Version -> String
showVersion (Version branch tags) = concat (intersperse "." (map show branch)) ++
                                    concatMap ('-':) tags
```

Функция: `parseVersion`

Описание: преобразует строку в формате, предлагаемом функцией `showVersion`, в представление версии типа `Version`.

Определение:

```
parseVersion :: ReadP Version
parseVersion = do branch <- sepBy1 (liftM read $ munch1 isDigit) (char '.')
                  tags   <- many (char '-' >> munch1 isAlphaNum)
                  return Version{versionBranch = branch, versionTags = tags}
```

8.34. Модуль `Word`

В модуле `Word` определены типы для представления ограниченных беззнаковых целых чисел. В отличие от модуля `Int`, в этом модуле описываются целочисленные типы без знака, занимающие различное количество байт в памяти, а потому использующиеся в различных целях. Использование:

```
import Data.Word
```

Первый тип данных, который описан в этом модуле, — `Word` — переопределяет примитивный тип.

Тип: `Word`

Описание: тип для представления беззнаковых целых чисел фиксированной точности. Все значения этого типа лежат в интервале $[-2^{29}, 2^{29} - 1]$, в точности как и значения типа `Int`. Точные значения нижней и верхней границы для конкретной реализации (в зависимости от транслятора языка) можно узнать при помощи методов `minBound` и `maxBound` класса `Bounded` (см. стр. 115).

Определение:

Тип определён в виде примитива.

Для данного типа определены экземпляры следующих классов: `Bits`, `Bounded`, `Data`, `Enum`, `Eq`, `Integral`, `Ix`, `Num`, `Ord`, `Read`, `Real`, `Show`, `Storable`, `Typeable`, `IArray`, и `MArray`.

Тип: `Word8`

Описание: тип для представления беззнаковых целых чисел, занимающих в памяти 8 бит (интервал $[0, 2^8 - 1]$).

Определение:

Тип определён в виде примитива.

Тип: `Word16`

Описание: тип для представления беззнаковых целых чисел, занимающих в памяти 16 бит (интервал $[0, 2^{16} - 1]$).

Определение:

Тип определён в виде примитива.

Тип: `Word32`

Описание: тип для представления беззнаковых целых чисел, занимающих в памяти 32 бита (интервал $[0, 2^{32} - 1]$).

Определение:

Тип определён в виде примитива.

Тип: `Word64`

Описание: тип для представления беззнаковых целых чисел, занимающих в памяти 64 бита (интервал $[0, 2^{64} - 1]$).

Определение:

Тип определён в виде примитива.

Для типов Word8, Word16, Word32 и Word64 определены экземпляры следующих классов: Bits, Bounded, Data, Enum, Eq, Integral, Ix, Num, Ord, Read, Real, Show, Storable, Typeable, IArray и MArray.

Необходимо отметить, что для перечисленных типов все арифметические операции выполняются по модулю 2^n, где n — количество бит, используемых для представления числа. Преобразование из одного целочисленного типа в другой можно производить при помощи функции fromIntegral, которая для всех перечисленных типов работает достаточно быстро. Также правила класса Enum (см. стр. 115) для типа Int работают и с определёнными в этом модуле целочисленными типами. Наконец, правый и левый сдвиги (см. раздел 8.2.) на число битов, равное или большее количеству битов, используемому для представления типа, возвращает значение 0.

Глава 9.

Пакет модулей Debug

Пакет модулей `Debug` содержит единственный стандартный модуль `Trace` (в конкретных поставках трансляторов языка Haskell могут иметься дополнительные модули), который используется для нужд отлова ошибок и мониторинга процесса исполнения функциональных программ.

9.1. Модуль Trace

Единственный стандартный модуль пакета `Debug` (в поставках некоторых трансляторов, в частности GHC, пакет может быть расширен дополнительными модулями), предназначенный для использования в целях отладки разрабатываемых программ. Использование:

```
import Debug.Trace
```

В этом модуле описаны две функции, которые используются для трассировки.
Функция: `putTraceMsg`
Описание: выводит заданное сообщение в рамках монады `IO`. Вывод осуществляется в стандартный поток ошибок `stderr`. Однако если функция вызывается из графического приложения, вывод может осуществиться в соответствии с API графической оболочки.
Определение:

```
putTraceMsg :: String -> IO ()
```

```
putTraceMsg msg = do hPutStrLn stderr msg
```

Функция: `trace`

Описание: при вызове выводит на экран (в консоль ошибок) заданную строку с сообщением об ошибке, после чего запускает второй аргумент и возвращает его результат в качестве своего результата. Эта функция не является детерминированной, а потому должна использоваться только в целях отладки или мониторинга исполнения.

Определение:

```
trace :: String -> a -> a
trace string expr = unsafePerformIO $ do putTraceMsg string
                                         return expr
```

Глава 10.

Пакет модулей `Foreign`

Пакет модулей `Foreign` включает в себя модули, которые содержат определения типов данных, классов и функций, использующиеся для межпрограммного взаимодействия с программами, написанными на других языках программирования.

Что интересно, в поставке стандартных модулей имеется модуль `Foreign`, который необходимо использовать следующим образом:

```
import Foreign
```

Этот модуль включает в себя реимпорт всех подчинённых модулей, а также определение единственной функции. Реимпорт подчинённых модулей очень полезен в том смысле, что в проект можно подключить только модуль `Foreign`, после чего для работы будут доступны все остальные модули.

Функция: `unsafePerformIO`

Описание: позволяет выполнить вычисления в монаде `IO` в любое время (при вызове функции), при этом возвращается результат вычисления, не обёрнутый монадой. Эта функция небезопасна. Чтобы она была безопасной, необходимо обеспечить детерминированность и отсутствие побочных эффектов у выполняемого действия.

Определение:

```
unsafePerformIO :: IO a -> a
```

Функция определена в виде примитива.

Если при помощи этой функции выполняются действия, содержащие побочные эффекты (непосредственно ввод или вывод), то относительный порядок этих действий в монаде становится неопределённым. Однако более страшными последствиями отличаются вызовы этой функции при использовании полиморфизма типов в ссылках. Например:

```
test :: IORef [a]
test = unsafePerformIO $ newIORef []

main = do writeIORef test [42]
          bang <- readIORef test
          print (bang :: [Char])
```

Этот код вызовет аварийный останов программы из-за полиморфной ссылки, которая не вызовет никаких проблем при обычной работе в монаде IO. И нет лёгких путей для преодоления этой проблемы при использовании описанной функции.

Остаётся отметить, что ответственный за все модули пакета Foreign имеет иной адрес электронной почты, по которому с ним можно связаться для выяснения интересующих вопросов или для внесения предложений. Этот адрес: ffi@haskell.org. Во всех нижеследующих описаниях эта информация опускается.

10.1. Модуль C

Модуль, который включает в себя при помощи реимпорта функциональность трёх нижеследующих модулей, описывающих специфические функции для работы в рамках подхода FFI (Foreign Function Interface — интерфейс к внешним функциям) с внешними программами, написанными на языке программирования C. Использование:

```
import Foreign.C
```

10.1.1. Модуль Error

Модуль Error описывает набор программных сущностей, позволяющих обрабатывать коды ошибок в стиле языка C (понятие «errno» из этого языка программирования). Использование:

```
import Foreign.C.Error
```

Коды ошибок представляют собой целые числа. В языке Haskell для их представления используется изоморфный тип Errno.

Тип: Errno

Описание: представление кодов ошибок из языка C в языке программирования Haskell. Это представление преднамеренно сделано открытым (не определено в виде примитива), чтобы разрешить пользователям определять и обрабатывать свои коды ошибок.

Определение:

```
newtype Errno = Errno CInt
```

Для данного типа определён экземпляр класса Eq.

Различные операционные системы и (или) различные библиотеки языка C определяют константы для представления кодов ошибок по-разному. Поэтому такими константами необходимо пользоваться с известной долей осторожности. В этом модуле определены константные функции для наиболее общих кодов ошибок. Но в силу открытости определения типа Errno, разработчик может добавлять свои определения, которые отсутствуют в этом модуле.

В рассматриваемом модуле определено около ста константных функций, возвращающих определённое значение типа Errno. Все эти функции имеют шаблонные наименования. Они начинаются со строчной буквы e, после которой идёт обозначение кода ошибки полностью заглавными буквами. Например: eOK, ePROTONOSUPPORT или eXDEV. Нет никакого смысла в перечислении всех этих функций здесь, поскольку их можно просмотреть в исходном коде модуля.

Далее описываются функции, которые помогают обрабатывать коды ошибок.

Функция: isValidErrno

Описание: возвращает значение True, если переданный на вход код ошибки является валидным в системе исполнения программы. Отсюда следует, что экземпляр

класса Eq для типа Errno также является зависимым от системы и работает только с валидными кодами ошибок.
Определение:

```
isValidErrno :: Errno -> Bool
isValidErrno (Errno errno) = errno /= -1
```

Функция: getErrno
Описание: возвращает текущее значение кода errno для текущего потока управления.
Определение:

```
getErrno :: IO Errno
getErrno = do e <- get_errno
              return (Errno e)
```

Функция: resetErrno
Описание: сбрасывает текущий код ошибки в значение eOK.
Определение:

```
resetErrno :: IO ()
resetErrno = set_errno 0
```

Функция: errnoToIOError
Описание: создаёт на основе кода ошибки представление этой ошибки в системе обработки ошибок языка Haskell. Третий и четвёртый аргументы функции могут быть использованы для увеличения точности преобразования. Первый аргумент представляет описание места, где произошла ошибка. Второй — собственно код ошибки. Третий (опционально) — обработчик файла, ассоциированный с ошибкой. Четвёртый (опционально) — имя файла, ассоциированное с ошибкой.
Определение:

```
errnoToIOError :: String -> Errno -> Maybe Handle -> Maybe String -> IOError
errnoToIOError loc errno maybeHdl maybeName
  = unsafePerformIO $
    do str <- strerror errno >>= peekCString
       return (userError (loc ++ ": " ++ str ++ maybe "" (": "++) maybeName))
```

Функция: `throwErrno`

Описание: выбрасывает исключение типа `IOError` на основе текущего кода ошибки. Первый аргумент используется как текстовое описание ошибки.

Определение:

```
throwErrno :: String -> IO a
throwErrno loc = do errno <- getErrno
                    ioError (errnoToIOError loc errno Nothing Nothing)
```

Функция: `throwErrnoIf`

Описание: выбрасывает исключение, соответствующее текущему значению кода ошибки, если результат заданного действия в монаде `IO` удовлетворяет заданному предикату. Первый аргумент является предикатом. Второй — текстовым описанием ошибки. Третий — монадическое действие ввода/вывода, которое необходимо выполнить.

Определение:

```
throwErrnoIf :: (a -> Bool) -> String -> IO a -> IO a
throwErrnoIf pred loc f = do res <- f
                             if pred res
                               then throwErrno loc
                               else return res
```

Функция: `throwErrnoIf-`

Описание: вариант функции `throwErrnoIf`, который игнорирует результат выполненного действия (используется тогда, когда результат не важен, но важны побочные эффекты монады `IO`).

Определение:

```
throwErrnoIf_ :: (a -> Bool) -> String -> IO a -> IO ()
throwErrnoIf_ pred loc f = void $ throwErrnoIf pred loc f
```

Функция: `throwErrnoIfRetry`

Описание: вариант функции `throwErrnoIf`, который повторяет заданное монадическое действие, если текущий код ошибки равен значению `eINTR`. Это значение является стандартным для циклов повторения в системах стандарта POSIX.

Определение:

```
throwErrnoIfRetry :: (a -> Bool) -> String -> IO a -> IO a
throwErrnoIfRetry pred loc f = do res <- f
                                  if pred res
                                    then do err <- getErrno
                                            if err == eINTR
                                              then throwErrnoIfRetry pred loc f
                                              else throwErrno loc
                                    else return res
```

Функция: `throwErrnoIfRetry-`

Описание: вариант функции `throwErrnoIfRetry`, который игнорирует результат выполненного действия (используется тогда, когда результат не важен, но важны побочные эффекты монады `IO`).

Определение:

```
throwErrnoIfRetry_ :: (a -> Bool) -> String -> IO a -> IO ()
throwErrnoIfRetry_ pred loc f  = void $ throwErrnoIfRetry pred loc f
```

Функция: `throwErrnoIfMinus1`

Описание: выкидывает исключение типа `IOError`, соответствующее текущему значению кода ошибки, если заданное действие в монаде `IO` вернуло значение `-1`.

Определение:

```
throwErrnoIfMinus1 :: Num a => String -> IO a -> IO a
throwErrnoIfMinus1 = throwErrnoIf (== -1)
```

Функция: `throwErrnoIfMinus1-`

Описание: вариант функции `throwErrnoIfMinus1`, который игнорирует результат выполненного действия (используется тогда, когда результат не важен, но важны побочные эффекты монады `IO`).

Определение:

```
throwErrnoIfMinus1_ :: Num a => String -> IO a -> IO ()
throwErrnoIfMinus1_ = throwErrnoIf_ (== -1)
```

Функция: `throwErrnoIfMinus1Retry`

Описание: выкидывает исключение типа `IOError`, соответствующее текущему значению кода ошибки, если заданное действие в монаде `IO` вернуло значение `-1`, однако пытается повторить это действие в случае, если операция была прервана.

Определение:

```
throwErrnoIfMinus1Retry :: Num a => String -> IO a -> IO a
throwErrnoIfMinus1Retry = throwErrnoIfRetry (== -1)
```

Функция: `throwErrnoIfMinus1Retry-`

Описание: вариант функции `throwErrnoIfMinus1Retry`, который игнорирует результат выполненного действия (используется тогда, когда результат не важен, но важны побочные эффекты монады `IO`).

Определение:

```
throwErrnoIfMinus1Retry_ :: Num a => String -> IO a -> IO ()
throwErrnoIfMinus1Retry_ = throwErrnoIfRetry_ (== -1)
```

Функция: `throwErrnoIfNull`

Описание: выкидывает исключение типа `IOError`, соответствующее текущему значению кода ошибки, если заданное действие в монаде `IO` вернуло значение `nullPtr` (пустой указатель).

Определение:

```
throwErrnoIfNull :: String -> IO (Ptr a) -> IO (Ptr a)
throwErrnoIfNull = throwErrnoIf (== nullPtr)
```

Функция: `throwErrnoIfNullRetry`

Описание: выкидывает исключение типа `IOError`, соответствующее текущему значению кода ошибки, если заданное действие в монаде `IO` вернуло значение `nullPtr` (пустой указатель), однако пытается повторить это действие в случае, если операция была прервана.

Определение:

```
throwErrnoIfNullRetry :: String -> IO (Ptr a) -> IO (Ptr a)
throwErrnoIfNullRetry = throwErrnoIfRetry (== nullPtr)
```

Функция: `throwErrnoIfRetryMayBlock`

Описание: вариант функции `throwErrnoIfRetry` (см. стр. 425), который проверяет операции, которые могут заблокировать поток управления, и если это произошло, выполняет заданное четвёртым аргументом монадическое действие.

Определение:

```
throwErrnoIfRetryMayBlock :: (a -> Bool) -> String -> IO a -> IO b -> IO a
throwErrnoIfRetryMayBlock pred loc f on_block
  = do res <- f
       if pred res
         then do err <- getErrno
                 if err == eINTR
                   then throwErrnoIfRetryMayBlock pred loc f on_block
                   else if err == eWOULDBLOCK || err == eAGAIN
                          then do on_block
                                  throwErrnoIfRetryMayBlock pred loc f on_block
                          else throwErrno loc
         else return res
```

Функция: `throwErrnoIfRetryMayBlock-`
Описание: вариант функции `throwErrnoIfRetryMayBlock`, который игнорирует результат выполненного действия (используется тогда, когда результат не важен, но важны побочные эффекты монады `IO`).
Определение:

```
throwErrnoIfRetryMayBlock_ :: (a -> Bool) -> String -> IO a -> IO b -> IO ()
throwErrnoIfRetryMayBlock_ pred loc f on_block
  = void $ throwErrnoIfRetryMayBlock pred loc f on_block
```

Функция: `throwErrnoIfMinus1RetryMayBlock`
Описание: вариант функции `throwErrnoIfMinus1Retry` (см. стр. 426), который проверяет операции, которые могут заблокировать поток управления, и если это произошло, выполняет заданное четвёртым аргументом монадическое действие.
Определение:

```
throwErrnoIfMinus1RetryMayBlock :: Num a => String -> IO a -> IO b -> IO a
throwErrnoIfMinus1RetryMayBlock = throwErrnoIfRetryMayBlock (== -1)
```

Функция: `throwErrnoIfMinus1RetryMayBlock-`
Описание: вариант функции `tthrowErrnoIfMinus1RetryMayBlock`, который игнорирует результат выполненного действия (используется тогда, когда результат не важен, но важны побочные эффекты монады `IO`).

Определение:

```
throwErrnoIfMinus1RetryMayBlock_ :: Num a => String -> IO a -> IO b -> IO ()
throwErrnoIfMinus1RetryMayBlock_ = throwErrnoIfRetryMayBlock_ (== -1)
```

Функция: `throwErrnoIfNullRetryMayBlock`
Описание: вариант функции `throwErrnoIfNullRetry` (см. стр. 427), который проверяет операции, которые могут заблокировать поток управления, и если это произошло, выполняет заданное четвёртым аргументом монадическое действие.
Определение:

```
throwErrnoIfNullRetryMayBlock :: String -> IO (Ptr a) -> IO b -> IO (Ptr a)
throwErrnoIfNullRetryMayBlock = throwErrnoIfRetryMayBlock (== nullPtr)
```

10.1.2. Модуль `String`

Модуль `String` содержит описания программных сущностей для маршалинга (то есть упаковки в пакеты с приписыванием определённой служебной информации для последующей передачи по сети) строк в формате языка C. Использование:

```
import Foreign.C.String
```

Процесс маршалинга (маршализация) преобразует каждый символ в формате языка Haskell (представление символа формата Unicode) в один или несколько байт, как это определено текущими установками системы локализации. Как следствие, нет никаких гарантий того, что получающаяся строка в формате языка C будет соответствовать по длине строке в формате языка Haskell. Преобразование между представлением строки в формате языка Haskell и текущим форматом системы локализации может быть теряющим информацию.

Для представления обычных строк в формате языка C используются следующие типы данных.

Тип: `CString`
Описание: представление строки в формате языка C — ссылка на последовательность байт, оканчивающуюся нулём.
Определение:

```
type CString = Ptr CChar
```

Тип: `CStringLen`

Описание: представление строки в формате языка C с явным указанием длины строки вместо окончания на нуль. Это значит, что внутри такой строки могут находиться символы с нулевым кодом.

Определение:

```
type CStringLen = (Ptr CChar, Int)
```

Функция: `peekCString`

Описание: маршализует строку в формате языка C в строку в формате языка Haskell.

Определение:

```
peekCString :: CString -> IO String
peekCString = peekCAString
```

Функция: `peekCStringLen`

Описание: маршализует строку с явно заданной длиной в строку в формате языка Haskell.

Определение:

```
peekCStringLen :: CStringLen -> IO String
peekCStringLen = peekCAStringLen
```

Функция: `newCString`

Описание: маршализует строку в формате языка Haskell в строку в формате языка C, при этом исходная строка не должна содержать символов с нулевым кодом. Новая строка в формате языка C после использования должна быть явно удалена при помощи функций `free` (см. стр. 447) или `finalizerFree` (см. стр. 447).

Определение:

```
newCString :: String -> IO CString
newCString = newCAString
```

Функция: `newCStringLen`

Описание: маршализует строку в формате языка Haskell в строку в формате языка C. Новая строка в формате языка C с явно заданной длиной после использования должна быть явно удалена при помощи функций `free` (см. стр. 447) или `finalizerFree` (см. стр. 447).

Определение:

```
newCStringLen :: String -> IO CStringLen
newCStringLen = newCAStringLen
```

Функция: `withCString`
Описание: маршализует строку в формате языка Haskell в строку в формате языка C, при этом исходная строка не должна содержать символов с нулевым кодом. Новая строка в формате языка C после использования автоматически уничтожается сборщиком мусора, а потому указатель на неё не должен использоваться после этого.
Определение:

```
withCString :: String -> (CString -> IO a) -> IO a
withCString = withCAString
```

Функция: `withCStringLen`
Описание: маршализует строку в формате языка Haskell в строку в формате языка C с явно заданной длиной. Новая строка в формате языка C после использования автоматически уничтожается сборщиком мусора, а потому указатель на неё не должен использоваться после этого.
Определение:

```
withCStringLen :: String -> (CStringLen -> IO a) -> IO a
withCStringLen = withCAStringLen
```

Вышеперечисленные шесть функций полностью повторяют функциональность маршализации для массивов (см. ниже), поэтому их определения выглядят таким образом.

Функция: `charIsRepresentable`
Описание: возвращает значение `True`, если заданный символ может быть безопасно представлен в формате строк языка C. Все те символы, для которых этот предикат возвращает значение `False`, при преобразовании становятся символом (`?`). На текущий момент представимыми являются только символы таблицы Latin-1 из кодировки Unicode.

Определение:

```
charIsRepresentable :: Char -> IO Bool
charIsRepresentable c = return (ord c < 256)
```

Функция: `castCharToCChar`

Описание: преобразует символ в формате языка Haskell в формат языка C. Эта функция безопасна только для символов с кодами, меньшими 256. Для символов с большими кодами происходит потеря информации.

Определение:

```
castCharToCChar :: Char -> CChar
castCharToCChar ch = fromIntegral (ord ch)
```

Функция: `castCCharToChar`

Описание: функция, обратная по действию функции `castCharToCChar`, поскольку преобразует символ таблицы Latin-1 в формате языка C в формат языка Haskell.

Определение:

```
castCCharToChar :: CChar -> Char
castCCharToChar ch = chr (fromIntegral (fromIntegral ch :: Word8))
```

Функция: `peekCAString`

Описание: маршализует строку в формате языка C в строку в формате языка Haskell.

Определение:

```
peekCAString :: CString -> IO String
peekCAString cp = do cs <- peekArray0 nUL cp
                     return (cCharsToChars cs)
```

Функция: `peekCAStringLen`

Описание: маршализует строку с явно заданной длиной в строку в формате языка Haskell.

Определение:

```
peekCAStringLen :: CStringLen -> IO String
peekCAStringLen (cp, len) = do cs <- peekArray len cp
                               return (cCharsToChars cs)
```

Функция: `newCAString`

Описание: маршализует строку в формате языка Haskell в строку в формате языка C, при этом исходная строка не должна содержать символов с нулевым кодом. Новая строка в формате языка C после использования должна быть явно удалена при помощи функций `free` (см. стр. 447) или `finalizerFree` (см. стр. 447).

Определение:

```
newCAString :: String -> IO CString
newCAString = newArray0 nUL . charsToCChars
```

Функция: `newCAStringLen`

Описание: маршализует строку в формате языка Haskell в строку в формате языка C. Новая строка в формате языка C с явно заданной длиной после использования должна быть явно удалена при помощи функций `free` (см. стр. 447) или `finalizerFree` (см. стр. 447).

Определение:

```
newCAStringLen :: String -> IO CStringLen
newCAStringLen str = do a <- newArray (charsToCChars str)
                        return (pairLength str a)
```

Функция: `withCAString`

Описание: маршализует строку в формате языка Haskell в строку в формате языка C, при этом исходная строка не должна содержать символов с нулевым кодом. Новая строка в формате языка C после использования автоматически уничтожается сборщиком мусора, а потому указатель на неё не должен использоваться после этого.

Определение:

```
withCAString :: String -> (CString -> IO a) -> IO a
withCAString = withArray0 nUL . charsToCChars
```

Функция: `withCAStringLen`

Описание: маршализует строку в формате языка Haskell в строку в формате языка C с явно заданной длиной. Новая строка в формате языка C после использования автоматически уничтожается сборщиком мусора, а потому указатель на неё не должен использоваться после этого.

Определение:

```
withCAStringLen :: String -> (CStringLen -> IO a) -> IO a
withCAStringLen str act = withArray (charsToCChars str) $ act . pairLength str
```

Также в этом модуле описаны функции для маршалинга «широких» строк, использующихся в языке C для представления символов в кодировке Unicode (UTF-32 или UTF-16). Для представления таких строк в формате языка C используются следующие типы данных.

Тип: `CWString`

Описание: представление широкой строки в формате языка C — ссылка на последовательность байт, оканчивающуюся нулём.

Определение:

```
type CWString = Ptr CWchar
```

Тип: `CWStringLen`

Описание: представление широкой строки в формате языка C с явным указанием длины строки вместо окончания на нуль. Это значит, что внутри такой строки могут находиться символы с нулевым кодом.

Определение:

```
type CWStringLen = (Ptr CWchar, Int)
```

Функция: `peekCWString`

Описание: маршализует широкую строку в формате языка C в строку в формате языка Haskell.

Определение:

```
peekCWString :: CWString -> IO String
peekCWString cp = do cs <- peekArray0 wNUL cp
                     return (cWcharsToChars cs)
```

Функция: `peekCWStringLen`

Описание: маршализует широкую строку с явно заданной длиной в строку в формате языка Haskell.

Определение:

```
peekCWStringLen :: CWStringLen -> IO String
peekCWStringLen (cp, len) = do cs <- peekArray len cp
                               return (cWcharsToChars cs)
```

Функция: `newCWString`

Описание: маршализует строку в формате языка Haskell в широкую строку в формате языка C, при этом исходная строка не должна содержать символов с нулевым кодом. Новая строка в формате языка C после использования должна быть явно удалена при помощи функций `free` (см. стр. 447) или `finalizerFree` (см. стр. 447).

Определение:

```
newCWString :: String -> IO CWString
newCWString = newArray0 wNUL . charsToCWchars
```

Функция: `newCWStringLen`

Описание: маршализует строку в формате языка Haskell в широкую строку в формате языка C. Новая строка в формате языка C с явно заданной длиной после использования должна быть явно удалена при помощи функций `free` (см. стр. 447) или `finalizerFree` (см. стр. 447).

Определение:

```
newCWStringLen :: String -> IO CWStringLen
newCWStringLen str = do a <- newArray (charsToCWchars str)
                        return (pairLength str a)
```

Функция: `withCWString`

Описание: маршализует строку в формате языка Haskell в широкую строку в формате языка C, при этом исходная строка не должна содержать символов с нулевым кодом. Новая строка в формате языка C после использования автоматически уничтожается сборщиком мусора, а потому указатель на неё не должен использоваться после этого.

Определение:

```
withCWString :: String -> (CWString -> IO a) -> IO a
withCWString = withArray0 wNUL . charsToCWchars
```

Функция: `withCWStringLen`
Описание: маршализует строку в формате языка Haskell в широкую строку в формате языка C с явно заданной длиной. Новая строка в формате языка C после использования автоматически уничтожается сборщиком мусора, а потому указатель на неё не должен использоваться после этого.
Определение:

```
withCWStringLen :: String -> (CWStringLen -> IO a) -> IO a
withCWStringLen str act = withArray (charsToCWchars str) $ act . pairLength str
```

Также в этом модуле определены дополнительные служебные функции для преобразования, а также константные функции, используемые в процессе преобразования. Узнать их состав и понять суть их работы можно, обратившись к исходному коду модуля.

10.1.3. Модуль `Types`

В модуле `Types` описываются проекции некоторых типов языка C в соответствующие типы языка Haskell. Использование:

```
import Foreign.C.Types
```

Все типы, описываемые в этом модуле, необходимы для аккуратного представления прототипов функций на языке C для доступа к интерфейсам библиотек из программ на языке Haskell. Системы языка Haskell не обязаны в точности повторять способы представления в памяти этих типов, однако следующие правила гарантируют правильность работы с внешними функциями из программ на языке Haskell:

1) Если функция на языке C принимает на вход аргумент или возвращает значение некоторого типа `t`, использование соответствующего типа `CT` на соответствующей позиции в декларации прототипа на языке Haskell заставляет программу на нём использовать правильное множество значений исходного типа. И наоборот, произвольное значение типа `CT` на языке Haskell имеет валидное представление в языке C.

2) Значение функции `sizeOf` для значения (`undefined :: CT`) абсолютно точно равно значению оператора `sizeof (t)` в языке C.

3) Значение функции alignment для значения (undefined :: CT) соответствует ограничениям на тип в языке C.

4) Методы peek и poke класса Storable (см. стр. 468) проецируют все значения типа CT в соответствующие значения типа t.

5) Если для типа CT определён экземпляр типа Bounded (см. стр. 115), то его методы minBound и maxBound возвращают значения, соответствующие значениям t_MIN и t_MAX в языке C.

6) Если для типа CT определены экземпляры классов Eq (см. стр. 117) и (или) Ord (см. стр. 121), то отношения между значениями типа CT, определяемые предикатами этих классов, в точности равны отношениям между соответствующими значениями класса t (другими словами, сохраняются отношения эквивалентности и порядка).

7) Если для типа CT определены экземпляры классов Num (см. стр. 120), Integral (см. стр. 119), Fractional (см. стр. 118), Floating (см. стр. 117), RealFrac (см. стр. 123) и RealFloat (см. стр. 122), то значения операций из этих классов в точности равны значениям соответствующих функций языка C.

8) Если для типа CT определён экземпляр класса Bits (см. стр. 245), то значения битовых операций в точности равны значениям соответствующих операций языка C.

В следующей таблице описываются целочисленные типы языка C и их представление в языке Haskell. Все эти типы являются экземплярами следующих классов: Eq, Ord, Num, Read, Show, Enum, Typeable, Storable, Bounded, Real, Integral и Bits.

Таблица 10.1. Целочисленные типы

Тип Haskell	Тип C	Определение
CChar	**char**	**newtype** CChar = CChar Int8
CSChar	**signed char**	**newtype** CSChar = CSChar Int8
CUChar	**unsigned char**	**newtype** CUChar = CUChar Word8
CShort	**short**	**newtype** CShort = CShort Int16
CUShort	**unsigned short**	**newtype** CUShort = CUShort Word16
CInt	**int**	**newtype** CInt = CInt Int32
CUInt	**unsigned int**	**newtype** CUInt = CUInt Word32
CLong	**long**	**newtype** CLong = CLong Int32
CULong	**unsigned long**	**newtype** CULong = CULong Word32
CLLong	**long long**	**newtype** CLLong = CLLong Int64
CULLong	**unsigned long long**	**newtype** CULLong = CULLong Word64

В следующей таблице описываются интегральные типы языка C для представления различных значений (время, дата), а также их представление в языке Haskell. Все эти типы являются экземплярами следующих классов: Eq, Ord, Num, Read, Show, Enum, Typeable и Storable.

Таблица 10.2. Интегральные типы

Тип Haskell	Тип C	Определение
CClock	clock_t	**newtype** CClock = CClock Int32
CTime	time_t	**newtype** CTime = CTime Int32

В следующей таблице описываются типы языка C для представления действительных чисел и их отображения в языке Haskell. Все эти типы являются экземплярами следующих классов: Eq, Ord, Num, Read, Show, Enum, Typeable, Storable, Real, Fractional, Floating, RealFrac и RealFloat.

Таблица 10.3. Типы для представления действительных чисел

Тип Haskell	Тип C	Определение
CFloat	**float**	**newtype** CFloat = CFloat Float
CDouble	**double**	**newtype** CDouble = CDouble Double
CLDouble	**long double**	**newtype** CLDouble = CLDouble Double

Также в этом модуле определены в виде примитивов алгебраические типы данных, представляющие собой соответствия типам FILE, fpos_t и jmp_buf. Этим типам соответствуют типы языка Haskell: CFile, CFpos и CJmpBuf.

10.2. Модуль ForeignPtr

Модуль ForeignPtr содержит описания программных сущностей, которые используются для работы с указателями на внешние определения (в библиотеках, созданных при помощи иных языков программирования). Этот модуль является частью пакета Foreign (FFI), а потому обычно должен использоваться в рамках реимпорта через модуль Foreign. В случае если его необходимо использовать отдельно, этот модуль можно импортировать так:

```
import Foreign.ForeignPtr
```

Главный тип данных, описанный в этом модуле, — ForeignPtr. Все остальные программные сущности предназначены для обслуживания этого типа.

Тип: ForeignPtr

Описание: данный тип представляет ссылку на объект, который создан при помощи стороннего языка программирования, то есть на объект, который не входит в систему структур данных, обслуживаемые менеджером памяти языка Haskell. Главное отличие этого типа от простых ссылок на память посредством типа Ptr заключается в том, что рассматриваемый тип позволяет ассоциировать со своими значениями *финализаторы*. Под финализатором (иначе — деструктором) понимается функция, запускаемая тогда, когда сборщик мусора языка Haskell обнаруживает, что на внешний объект больше не существует ссылок из сегмента памяти, который обслуживается сборщиком мусора. Обычно финализатор при запуске в свою очередь запускает деструкторы из стороннего языка программирования, которые освобождают память.

Определение:

```
data ForeignPtr a = ...
```

Тип определён в виде примитива.

Для этого типа определены экземпляры следующих классов: Typeaple1, Data, Eq, Ord и Show.

Для удобства работы с финализваторами определено два дополнительных синонима.

Тип: FinalizerPtr

Описание: финализатор представляется в виде указателя на функцию, которая получает на вход обычный указатель на значение, которое должно быть уничтожено.

Определение:

```
type FinalizerPtr a = FunPtr (Ptr a -> IO ())
```

Тип: FinalizerEnvPtr

Описание: вариант типа FinalizerPtr, в котором финализатор является функцией, получающей на вход не только уничтожаемое значение, но и состояние окружения.

Определение:

```
type FinalizerEnvPtr env a = FunPtr (Ptr env -> Ptr a -> IO ())
```

Функция: newForeignPtr

Описание: преобразует обычный указатель в указатель на внешний объект, ассоциируя с указателем заданный финализатор. Этот финализатор будет выполнен, как только пропадёт последняя ссылка на внешний объект. Необходимо отметить, что неизвестно, как скоро будет запущен финализатор, это зависит от настроек сборщика мусора языка Haskell. Единственной гарантией является то, что финализатор будет запущен в любом случае перед окончанием работы программы.

Определение:

```
newForeignPtr :: FinalizerPtr a -> Ptr a -> IO (ForeignPtr a)
newForeignPtr finalizer p = do fObj <- newForeignPtr_ p
                               addForeignPtrFinalizer finalizer fObj
                               return fObj
```

Функция: newForeignPtr-
Описание: вариант функции newForeignPtr, который не ассоциирует финализатор с внешним указателем. Финализатор может быть добавлен позже при помощи функции addForeignPtrFinalizer.
Определение:

```
newForeignPtr_ :: Ptr a -> IO (ForeignPtr a)
```

Функция определена в виде примитива.
Функция: addForeignPtrFinalizer
Описание: ассоциирует с указателем на внешний объект заданный финализатор. К каждому внешнему объекту можно приписать несколько финализаторов, которые будут выполняться в порядке FILO (последний ассоциированный указатель выполнится первым).
Определение:

```
addForeignPtrFinalizer :: FinalizerPtr a -> ForeignPtr a -> IO ()
```

Функция определена в виде примитива.
Функция: newForeignPtrEnv
Описание: вариант функции newForeignPtr, который ассоциирует с внешним объектом финализатор, ожидающий на вход не только ссылку на объект, но и состояние окружения.
Определение:

```
newForeignPtrEnv :: FinalizerEnvPtr env a -> Ptr env -> Ptr a -> IO (ForeignPtr a)
newForeignPtrEnv finalizer env p = do fObj <- newForeignPtr_ p
                                      addForeignPtrFinalizerEnv finalizer env fObj
                                      return fObj
```

Функция: addForeignPtrFinalizerEnv
Описание: вариант функции addForeignPtrFinalizer, который добавляет к внешнему объекту финализатор, ожидающий на вход не только ссылку на объект, но и состояние окружения.
Определение:

```
addForeignPtrFinalizerEnv :: FinalizerEnvPtr env a -> Ptr env -> ForeignPtr a -> IO ()
```

Функция определена в виде примитива.

Функция: `withForeignPtr`

Описание: функция, позволяющая обратиться к внешнему объекту по ссылке на него. Вторым аргументом принимает на вход функцию, которая получает на вход указатель (простой) и преобразует его в монадическое действие (в монаде `IO`), которое выполняется. Внешний объект сохраняется, пока работает эта функция, даже если внутри неё он не используется. Однако необходимо отметить, что небезопасно возвращать ссылку на объект из монадического действия и использовать его после вызова этой функции, поскольку финализатор объекта может быть запущен раньше, чем это ожидается. Это происходит потому, что трансляторы языка Haskell не могут отслеживать использование простых указателей типа `Ptr`, созданные из указателей на внешние объекты. Остаётся отметить, что эта функция обычно используется для маршализации объектов, ссылки на которые хранятся в указателях типа `ForeignPtr`, при помощи методов класса `Storable` (см. стр. 468).

Определение:

```
withForeignPtr :: ForeignPtr a -> (Ptr a -> IO b) -> IO b
withForeignPtr fo io = do r <- io (unsafeForeignPtrToPtr fo)
                          touchForeignPtr fo
                          return r
```

Функция: `finalizeForeignPtr`

Описание: заставляет немедленно запустить финализатор, ассоциированный с заданным внешним объектом.

Определение:

```
finalizeForeignPtr :: ForeignPtr a -> IO ()
```

Функция определена в виде примитива.

Функция: `unsafeForeignPtrToPtr`

Описание: вычленяет указатель на внешний объект, превращая его в обычный указатель на ячейку памяти. Эта функция является потенциально опасной, поскольку если на её аргумент при вызове функции нет больше ссылок для использования, во время выполнения функции может быть запущен финализатор (время запуска финализатора недетерминировано), который делает полученный обычный указатель невалидным. Поэтому там, где необходимо гарантировать валидность указателя, необходимо пользоваться функцией

touchForeignPtr. Для того чтобы избежать появления трудноуловимых логических ошибок, при разработке программного обеспечения необходимо пользоваться функцией withForeignPtr, нежели связкой функций unsafeForeignPtrToPtr и touchForeignPtr. Эта связка обычно используется в исходном коде, сгенерированном инструментальными средствами, позволяющими автоматически обрабатывать маршализацию объектов.

Определение:

```
unsafeForeignPtrToPtr :: ForeignPtr a -> Ptr a
```

Функция определена в виде примитива.

Функция: touchForeignPtr

Описание: эта функция обеспечивает существование внешнего объекта по заданному указателю в рамках выполнения монадического действия в монаде IO. В рассматриваемом случае эта функция работает, как и функция withForeignPtr (см. стр. 441), после того, как выполняет своё монадическое действие.

Определение:

```
touchForeignPtr :: ForeignPtr a -> IO ()
```

Функция определена в виде примитива.

Функция: castForeignPtr

Описание: преобразует указатель на внешний объект в смысле изменения параметризуемого типа.

Определение:

```
castForeignPtr :: ForeignPtr a -> ForeignPtr b
```

Функция определена в виде примитива.

Функция: mallocForeignPtr

Описание: выделяет некоторое количество памяти и возвращает указатель на неё.

Определение:

```
mallocForeignPtr :: Storable a => IO (ForeignPtr a)
mallocForeignPtr = do r <- malloc
                      newForeignPtr finalizerFree r
```

Функция: mallocForeignPtrBytes
Описание: вариант функции mallocForeignPtr, в котором задаётся количество байт, необходимое для выделения.
Определение:

```
mallocForeignPtrBytes :: Int -> IO (ForeignPtr a)
mallocForeignPtrBytes n = do r <- mallocBytes n
                             newForeignPtr finalizerFree r
```

Функция: mallocForeignPtrArray
Описание: вариант функции mallocArray (см. стр. 447), снабжающий выделяемую область памяти ассоциированным с ней финализатором.
Определение:

```
mallocForeignPtrArray :: Storable a => Int -> IO (ForeignPtr a)
mallocForeignPtrArray = doMalloc undefined
  where
    doMalloc            :: Storable b => b -> Int -> IO (ForeignPtr b)
    doMalloc dummy size = mallocForeignPtrBytes (size * sizeOf dummy)
```

Функция: mallocForeignPtrArray0
Описание: вариант функции mallocArray0 (см. стр. 448), снабжающий выделяемую область памяти ассоциированным с ней финализатором.
Определение:

```
mallocForeignPtrArray0 :: Storable a => Int -> IO (ForeignPtr a)
mallocForeignPtrArray0 size = mallocForeignPtrArray (size + 1)
```

10.3. Модуль Marshal

Модуль, который включает в себя при помощи реимпорта функциональность пяти нижеследующих модулей, описывающих специфические функции для работы в рамках подхода FFI с объектами, над которыми можно производить маршализацию. Использование:

```
import Foreign.Marshal
```

10.3.1. Модуль Alloc

В модуле Alloc описаны функции для выделения памяти в процессе маршализации. Обычно этот модуль подключается автоматически при помощи реимпорта в модуле Marshal. Если имеется необходимость в использовании этого модуля самостоятельно, его подключение выглядит следующим образом:

```
import Foreign.Marshal.Alloc
```

Функция: alloca

Описание: выполняет заданное действие, передавая в него выделенный (временно) блок памяти, достаточный для содержания значений типа a. Память высвобождается тогда, когда действие, заданное первым аргументом, заканчивается (нормальным способом или при помощи исключения), так что указатель на выделенную память после выполненного действия использоваться не должен.
Определение:

```
alloca :: Storable a => (Ptr a -> IO b) -> IO b
alloca = doAlloca undefined
  where
    doAlloca :: Storable a' => a' -> (Ptr a' -> IO b') -> IO b'
    doAlloca dummy = allocaBytes (sizeOf dummy)
```

Функция: allocaBytes

Описание: вариант функции alloca, который принимает на вход первым аргументом количество байт, которое необходимо выделить.
Определение:

```
allocaBytes :: Int -> (Ptr a -> IO b) -> IO b
allocaBytes size = bracket (mallocBytes size) free
```

Функция: malloc

Описание: выделяет память, достаточную для хранения значений типа a. Память выделяется на основании метода sizeOf класса Storable (см. стр. 468). Память может быть высвобождена при помощи функций free или finalizerFree (см. ниже).

Определение:

```
malloc :: Storable a => IO (Ptr a)
malloc = doMalloc undefined
  where
    doMalloc :: Storable b => b -> IO (Ptr b)
    doMalloc dummy = mallocBytes (sizeOf dummy)
```

Функция: mallocBytes

Описание: вариант функции malloc, который принимает на вход первым аргументом количество байт, которое необходимо выделить.

Определение:

```
mallocBytes :: Int -> IO (Ptr a)
mallocBytes size = failWhenNULL "malloc" (_malloc (fromIntegral size))
```

Функция: realloc

Описание: изменяет размер памяти, которая была выделена при помощи функций malloc или mallocBytes до количества, достаточного для хранения значений типа b. Возвращаемый указатель может указывать на совершенно иной участок памяти, однако будет достаточен для хранения требуемых значений. Содержимое памяти по новому указателю будет идентично содержимому, хранимому по старому указателю. Если аргументом этой функции является значение nullPtr, то функция ведёт себя так же, как и функция malloc.

Определение:

```
realloc :: Storable b => Ptr a -> IO (Ptr b)
realloc = doRealloc undefined
  where
    doRealloc :: Storable b' => b' -> Ptr a' -> IO (Ptr b')
    doRealloc dummy ptr = let size = fromIntegral (sizeOf dummy)
                          in  failWhenNULL "realloc" (_realloc ptr size)
```

Функция: reallocBytes

Описание: вариант функции realloc, который принимает на вход первым аргументом количество байт, которое необходимо выделить. Дополнительно: если заданное количество байт равно 0, то эта функция ведёт себя так же, как и функция free.

Определение:

```
reallocBytes :: Ptr a -> Int -> IO (Ptr a)
reallocBytes ptr 0    = do free ptr; return nullPtr
reallocBytes ptr size = failWhenNULL "realloc" (_realloc ptr (fromIntegral size))
```

Функция: `free`
Описание: освобождает блок памяти, который был выделен функциями `malloc`, `mallocBytes`, `realloc`, `reallocBytes`, `new` (см. стр. 460) и любыми функциями семейства `new` из модулей `Array` (см. подраздел 10.3.2.) и `String` (см. подраздел 10.1.2.).
Определение:

```
free :: Ptr a -> IO ()
free = _free
```

Функция: `finalizerFree`
Описание: возвращает указатель на функцию, которая может быть использована в качестве финализатора для памяти, выделенной при помощи функций `malloc`, `mallocBytes`, `realloc` или `reallocBytes`.
Определение:

```
finalizerFree :: FinalizerPtr a
```

Функция определена в виде примитива.

10.3.2. Модуль Array

Модуль `Array` содержит определения функций, предназначенных для работы с массивами (выделение памяти, маршализация). Предполагается, что он подключается в проект при помощи реимпорта из модуля `Marshal`, однако если имеется необходимость использования этого модуля отдельно, его можно подключить следующим образом:

```
import Foreign.Marshal.Array
```

В этом модуле определены только функции в количестве около двадцати штук.

Функция: mallocArray

Описание: выделяет память для хранения заданного количества элементов типа, являющегося экземпляром класса Storable. Работает так же, как функция malloc (см. стр. 445), однако для множества элементов заданного типа.

Определение:

```
mallocArray :: Storable a => Int -> IO (Ptr a)
mallocArray = doMalloc undefined
  where
    doMalloc :: Storable a' => a' -> Int -> IO (Ptr a')
    doMalloc dummy size = mallocBytes (size * sizeOf dummy)
```

Функция: mallocArray0

Описание: вариант функции mallocArray, добавляющий дополнительную память в конце выделяемого участка для хранения замыкающего (терминального) элемента (аналог символа с кодом 0 в строках формата языка C).

Определение:

```
mallocArray0 :: Storable a => Int -> IO (Ptr a)
mallocArray0 size = mallocArray (size + 1)
```

Функция: allocaArray

Описание: выделяет временную память для хранения заданного количества элементов типа, являющегося экземпляром класса Storable. Работает так же, как функция alloca (см. стр. 445), однако для множества элементов заданного типа. Память автоматически высвобождается сборщиком мусора.

Определение:

```
allocaArray :: Storable a => Int -> (Ptr a -> IO b) -> IO b
allocaArray = doAlloca undefined
  where
    doAlloca :: Storable a' => a' -> Int -> (Ptr a' -> IO b') -> IO b'
    doAlloca dummy size = allocaBytes (size * sizeOf dummy)
```

Функция: allocaArray0

Описание: вариант функции allocaArray, добавляющий дополнительную память в конце выделяемого участка для хранения замыкающего (терминального) элемента (аналог символа с кодом 0 в строках формата языка C).

Определение:

```
allocaArray0 :: Storable a => Int -> (Ptr a -> IO b) -> IO b
allocaArray0 size = allocaArray (size + 1)
```

Функция: `reallocArray`

Описание: изменяет размер массива, выделяя дополнительную или высвобождая излишнюю память.

Определение:

```
reallocArray :: Storable a => Ptr a -> Int -> IO (Ptr a)
reallocArray = doRealloc undefined
  where
    doRealloc :: Storable a' => a' -> Ptr a' -> Int -> IO (Ptr a')
    doRealloc dummy ptr size = reallocBytes ptr (size * sizeOf dummy)
```

Функция: `reallocArray0`

Описание: вариант функции `reallocArray`, добавляющий дополнительную память в конце выделяемого участка для хранения замыкающего (терминального) элемента (аналог символа с кодом 0 в строках формата языка C).

Определение:

```
reallocArray0 :: Storable a => Ptr a -> Int -> IO (Ptr a)
reallocArray0 ptr size = reallocArray ptr (size + 1)
```

Функция: `peekArray`

Описание: преобразует массив заданной длины в список языка Haskell. Эта функция обходит заданный массив сзаду наперёд, собирая список в накапливающий параметр, что требует постоянного размера стека.

Определение:

```
peekArray :: Storable a => Int -> Ptr a -> IO [a]
peekArray size ptr | size <= 0 = return []
                   | otherwise = f (size-1) []
  where
    f 0 acc = do e <- peekElemOff ptr 0
                 return (e:acc)
    f n acc = do e <- peekElemOff ptr n
                 f (n - 1) (e:acc)
```

Функция: peekArray0

Описание: вариант функции peekArray, добавляющий дополнительную память в конце выделяемого участка для хранения замыкающего (терминального) элемента (аналог символа с кодом 0 в строках формата языка C).

Определение:

```
peekArray0 :: (Storable a, Eq a) => a -> Ptr a -> IO [a]
peekArray0 marker ptr = do size <- lengthArray0 marker ptr
                           peekArray size ptr
```

Функция: pokeArray

Описание: последовательно записывает элементы заданного списка в память по заданному указателю.

Определение:

```
pokeArray :: Storable a => Ptr a -> [a] -> IO ()
pokeArray ptr vals = zipWithM_ (pokeElemOff ptr) [0..] vals
```

Функция: pokeArray0

Описание: вариант функции pokeArray, добавляющий дополнительную память в конце выделяемого участка для хранения замыкающего (терминального) элемента (аналог символа с кодом 0 в строках формата языка C). Терминальный маркер задаётся первым аргументом.

Определение:

```
pokeArray0 :: Storable a => a -> Ptr a -> [a] -> IO ()
pokeArray0 marker ptr vals = do pokeArray ptr vals
                                pokeElemOff ptr (length vals) marker
```

Функция: newArray

Описание: выделяет память и последовательно записывает в неё элементы заданного списка.

Определение:

```
newArray :: Storable a => [a] -> IO (Ptr a)
newArray vals = do ptr <- mallocArray (length vals)
                   pokeArray ptr vals
                   return ptr
```

Функция: newArray0

Описание: вариант функции **newArray**, добавляющий дополнительную память в конце выделяемого участка для хранения замыкающего (терминального) элемента (аналог символа с кодом 0 в строках формата языка C). Терминальный маркер задаётся первым аргументом.

Определение:

```
newArray0 :: Storable a => a -> [a] -> IO (Ptr a)
newArray0 marker vals = do ptr <- mallocArray0 (length vals)
                           pokeArray0 marker ptr vals
                           return ptr
```

Функция: withArray

Описание: выделяет временную память, которая высвобождается автоматически, и последовательно записывает в неё элементы заданного списка.

Определение:

```
withArray :: Storable a => [a] -> (Ptr a -> IO b) -> IO b
withArray vals = withArrayLen vals . constwithArray0
```

Функция: withArray0

Описание: вариант функции **withArray**, добавляющий дополнительную память в конце выделяемого участка для хранения замыкающего (терминального) элемента (аналог символа с кодом 0 в строках формата языка C). Терминальный маркер задаётся первым аргументом.

Определение:

```
withArray0 :: Storable a => a -> [a] -> (Ptr a -> IO b) -> IO b withArray0
marker vals = withArrayLen0 marker vals . const
```

Функция: withArrayLen

Описание: вариант функции **withArray**, однако выполняемое монадическое действие принимает на вход в качестве дополнительного параметра длину массива.

Определение:

```
withArrayLen :: Storable a => [a] -> (Int -> Ptr a -> IO b) -> IO b
withArrayLen vals f = allocaArray len $
                      \ptr -> do pokeArray ptr vals
                                 res <- f len ptr
                                 return res
```

```
  where
    len = length vals
```

Функция: withArrayLen0

Описание: вариант функции withArrayLen, добавляющий дополнительную память в конце выделяемого участка для хранения замыкающего (терминального) элемента (аналог символа с кодом 0 в строках формата языка C). Терминальный маркер задаётся первым аргументом.

Определение:

```
withArrayLen0 :: Storable a => a -> [a] -> (Int -> Ptr a -> IO b) -> IO b
withArrayLen0 marker vals f = allocaArray0 len $
                              \ptr -> do pokeArray0 marker ptr vals
                                         res <- f len ptr
                                         return res
  where
    len = length vals
```

Функция: copyArray

Описание: копирует элементы из второго массива в первый (по порядку аргументов). Области памяти, обрабатываемые этой функцией, не должны пересекаться.

Определение:

```
copyArray :: Storable a => Ptr a -> Ptr a -> Int -> IO ()
copyArray = doCopy undefined
  where
    doCopy :: Storable a' => a' -> Ptr a' -> Ptr a' -> Int -> IO ()
    doCopy dummy dest src size = copyBytes dest src (size * sizeOf dummy)
```

Функция: moveArray

Описание: перемещает элементы из второго массива в первый (по порядку аргументов). Области памяти, обрабатываемые этой функцией, не должны пересекаться.

Определение:

```
moveArray :: Storable a => Ptr a -> Ptr a -> Int -> IO ()
moveArray = doMove undefined
  where
    doMove :: Storable a' => a' -> Ptr a' -> Ptr a' -> Int -> IO ()
    doMove dummy dest src size = moveBytes dest src (size * sizeOf dummy)
```

Функция: lengthArray0

Описание: возвращает количество элементов в заданном массиве, исключая терминальный маркер.

Определение:

```
lengthArray0 :: (Storable a, Eq a) => a -> Ptr a -> IO Int
lengthArray0 marker ptr = loop 0
  where
    loop i = do val <- peekElemOff ptr i
                if val == marker
                  then return i
                  else loop (i + 1)
```

Функция: advancePtr

Описание: сдвигает указатель с начала массива на заданное количество элементов.

Определение:

```
advancePtr :: Storable a => Ptr a -> Int -> Ptr a
advancePtr = doAdvance undefined
  where
    doAdvance :: Storable a' => a' -> Ptr a' -> Int -> Ptr a'
    doAdvance dummy ptr i = ptr `plusPtr` (i * sizeOf dummy)
```

10.3.3. Модуль Error

Модуль Error содержит функции, необходимые для проверки возвращаемых при маршализации значений и генерации исключений типа userError в случаях наличия ошибок в значениях. Предполагается, что этот модуль подключается в проект при помощи реимпорта из модуля Marshal, однако если имеется необходимость использования этого модуля отдельно, его можно подключить следующим образом:

```
import Foreign.Marshal.Error
```

В модуле описано шесть функций, используемых для обработки ошибок.

Функция: throwIf

Описание: выполняет монадическое действие ввода/вывода, генерируя исключение в случае, если предикат (первый аргумент) возвращает значение True на ре-

зультате действия. Если исключение не возбуждается, возвращается результат монадического действия. Второй аргумент используется для получения строки сообщения об ошибке.
Определение:

```
throwIf :: (a -> Bool) -> (a -> String) -> IO a -> IO a
throwIf pred msgfct act = do res <- act
                             (if pred res
                                then ioError . userError . msgfct
                                else return) res
```

Функция: throwIf_
Описание: вариант функции throwIf, который используется в случае, когда результат выполнения монадического действия не нужен, но важны лишь побочные эффекты, предоставляемые монадой IO.
Определение:

```
throwIf_ :: (a -> Bool) -> (a -> String) -> IO a -> IO ()
throwIf_ pred msgfct act = void $ throwIf pred msgfct act
```

Функция: throwIfNeg
Описание: вариант функции throwIf, генерирующий исключения в случаях отрицательного результата.
Определение:

```
throwIfNeg :: (Ord a, Num a) => (a -> String) -> IO a -> IO a
throwIfNeg = throwIf (< 0)
```

Функция: throwIfNeg_
Описание: вариант функции throwIfNeg, который используется в случае, когда результат выполнения монадического действия не нужен, но важны лишь побочные эффекты, предоставляемые монадой IO.
Определение:

```
throwIfNeg_ :: (Ord a, Num a) => (a -> String) -> IO a -> IO ()
throwIfNeg_ = throwIf_ (< 0)
```

Функция: throwIfNull
Описание: вариант функции throwIf, генерирующий исключения в случаях, если в результате выполнения монадического действия возвращён пустой указатель.

Определение:

```
throwIfNull :: String -> IO (Ptr a) -> IO (Ptr a)
throwIfNull = throwIf (== nullPtr) . const
```

Функция: `void`
Описание: уничтожает результат монадического действия ввода/вывода.
Определение:

```
void :: IO a -> IO ()
void act = act >> return ()
```

10.3.4. Модуль `Pool`

В этом модуле определены программные сущности, использующиеся для работы с пулами памяти, то есть областями памяти, в рамках которых происходит работа с данными как с единым целым. Это означает, что при перераспределении памяти в рамках одного пула происходит перераспределение всех выделенных блоков. Эта идиома полезна в тех случаях, когда использование функции `alloca` (см. стр. 445) с её неявным выделением и высвобождением памяти нежелательно, но использование функций `malloc` и `free` (см. стр. 445 и стр. 447 соответственно) выглядит неуклюже.

Предполагается, что этот модуль подключается в проект при помощи реимпорта из модуля `Marshal`, однако если имеется необходимость использования этого модуля отдельно, его можно подключить следующим образом:

```
import Foreign.Marshal.Pool
```

В модуле описан главный тип данных для работы с пулами памяти.

Тип: `Pool`
Описание: тип для представления пула памяти.
Определение:

```
newtype Pool = Pool (IORef [Ptr ()])
```

Функция: `newPool`
Описание: выделяет новый пул памяти.

Определение:

```
newPool :: IO Pool
newPool = liftM Pool (newIORef [])
```

Функция: `freePool`

Описание: высвобождает пул памяти и всё, что было создано внутри него.

Определение:

```
freePool :: Pool -> IO ()
freePool (Pool pool) = readIORef pool >>= freeAll
   where freeAll []     = return ()
         freeAll (p:ps) = free p >> freeAll ps
```

Функция: `withPool`

Описание: выполняет монадическое действие внутри пула памяти, который автоматически высвобождается после окончания действия.

Определение:

```
withPool :: (Pool -> IO b) -> IO b
withPool = bracket newPool freePool
```

Функция: `pooledMalloc`

Описание: выделяет память внутри заданного пула для хранения значения заданного типа. Количество памяти определяется при помощи метода `sizeOf` класса `Storable` (см. стр. 468).

Определение:

```
pooledMalloc :: Storable a => Pool -> IO (Ptr a)
pooledMalloc = pm undefined
  where
    pm :: Storable a' => a' -> Pool -> IO (Ptr a')
    pm dummy pool = pooledMallocBytes pool (sizeOf dummy)
```

Функция: `pooledMallocBytes`

Описание: выделяет заданное количество байт внутри пула памяти.

Определение:

```
pooledMallocBytes :: Pool -> Int -> IO (Ptr a)
pooledMallocBytes (Pool pool) size = do ptr <- mallocBytes size
                                        ptrs <- readIORef pool
                                        writeIORef pool (ptr:ptrs)
```

```
                                    return (castPtr ptr)
```

Функция: `pooledRealloc`

Описание: перераспределяет память внутри заданного пула для хранения значения заданного типа. Количество памяти определяется при помощи метода `sizeOf` класса `Storable` (см. стр. 468).

Определение:

```
pooledRealloc :: Storable a => Pool -> Ptr a -> IO (Ptr a)
pooledRealloc = pr undefined
  where
    pr :: Storable a' => a' -> Pool -> Ptr a' -> IO (Ptr a')
    pr dummy pool ptr = pooledReallocBytes pool ptr (sizeOf dummy)
```

Функция: `pooledReallocBytes`

Описание: перераспределяет заданное количество байт внутри пула памяти.

Определение:

```
pooledReallocBytes :: Pool -> Ptr a -> Int -> IO (Ptr a)
pooledReallocBytes (Pool pool) ptr size
  = do let cPtr = castPtr ptr
       throwIf (not . (cPtr `elem`)) (\_ -> "pointer not in pool") (readIORef pool)
       newPtr <- reallocBytes cPtr size
       ptrs <- readIORef pool
       writeIORef pool (newPtr : delete cPtr ptrs)
       return (castPtr newPtr)
```

Функция: `pooledMallocArray`

Описание: выделяет память для заданного количества элементов массива внутри заданного пула.

Определение:

```
pooledMallocArray :: Storable a => Pool -> Int -> IO (Ptr a)
pooledMallocArray = pma undefined
  where
    pma :: Storable a' => a' -> Pool -> Int -> IO (Ptr a')
    pma dummy pool size = pooledMallocBytes pool (size * sizeOf dummy)
```

Функция: pooledMallocArray0

Описание: вариант функции pooledMallocArray, добавляющий дополнительную память в конце выделяемого участка для хранения замыкающего (терминального) элемента (аналог символа с кодом 0 в строках формата языка C).

Определение:

```
pooledMallocArray0 :: Storable a => Pool -> Int -> IO (Ptr a)
pooledMallocArray0 pool size = pooledMallocArray pool (size + 1)
```

Функция: pooledReallocArray

Описание: изменяет длину массива, выделяя новую память или высвобождая её излишки. Функция работает внутри заданного пула памяти.

Определение:

```
pooledReallocArray :: Storable a => Pool -> Ptr a -> Int -> IO (Ptr a)
pooledReallocArray = pra undefined
  where
    pra :: Storable a' => a' -> Pool -> Ptr a' -> Int -> IO (Ptr a')
    pra dummy pool ptr size = pooledReallocBytes pool ptr (size * sizeOf dummy)
```

Функция: pooledReallocArray0

Описание: вариант функции pooledReallocArray, добавляющий дополнительную память в конце выделяемого участка для хранения замыкающего (терминального) элемента (аналог символа с кодом 0 в строках формата языка C).

Определение:

```
pooledReallocArray0 :: Storable a => Pool -> Ptr a -> Int -> IO (Ptr a)
pooledReallocArray0 pool ptr size = pooledReallocArray pool ptr (size + 1)
```

Функция: pooledNew

Описание: выделяет память внутри заданного пула памяти и маршализует заданное значение в эту выделенную память.

Определение:

```
pooledNew :: Storable a => Pool -> a -> IO (Ptr a)
pooledNew pool val = do ptr <- pooledMalloc pool
                        poke ptr val
                        return ptr
```

Функция: `pooledNewArray`
Описание: выделяет последовательный набор ячеек памяти внутри заданного пула и записывает в него элементы заданного списка.
Определение:

```
pooledNewArray :: Storable a => Pool -> [a] -> IO (Ptr a)
pooledNewArray pool vals = do ptr <- pooledMallocArray pool (length vals)
                              pokeArray ptr vals
                              return ptr
```

Функция: `pooledNewArray0`
Описание: вариант функции `pooledNewArray`, добавляющий дополнительную память в конце выделяемого участка для хранения замыкающего (терминального) элемента (аналог символа с кодом 0 в строках формата языка C).
Определение:

```
pooledNewArray0 :: Storable a => Pool -> a -> [a] -> IO (Ptr a)
pooledNewArray0 pool marker vals = do ptr <- pooledMallocArray0 pool (length vals)
                                      pokeArray0 marker ptr vals
                                      return ptr
```

10.3.5. Модуль `Utils`

Модуль `Utils` содержит определения функций для маршализации примитивных типов. Эти функции используются для маршализации в других модулях пакета `Foreign`. Предполагается, что этот модуль подключается в проект при помощи реимпорта из модуля `Marshal`, однако если имеется необходимость использования этого модуля отдельно, его можно подключить следующим образом:

```
import Foreign.Marshal.Utils
```

Функция: `with`
Описание: вызов `with val f` выполняет монадическое действие ввода/вывода `f`, передавая ему в качестве параметра указатель на временно выделенный блок памяти, в который маршализовано значение `val`. Другими словами, эта функция является комбинацией функций `alloca` и `poke` (см. стр. 445 и стр. 468 соответственно). Память, выделенная для хранения значения `val`, высвобождается после

окончания монадического действия (нормального или посредством генерации исключения), так что использование указателя после вызова функции невозможно.
Определение:

```
with :: Storable a => a -> (Ptr a -> IO b) -> IO b
with val f = alloca $
             \ptr -> do poke ptr val
                        res <- f ptr
                        return res
```

Функция: `new`
Описание: выделяет блок памяти для хранения заданного значения и маршализует его в эту память (комбинация функций `malloc` и `poke` — см. стр. 445 и стр. 468 соответственно). Размер выделяемой памяти определяется значением метода `sizeOf` класса `Storable` (см. стр. 468). Когда выделенная память больше не используется, её можно высвободить при помощи функций `free` или `finalizerFree` (см. стр. 447).
Определение:

```
new :: Storable a => a -> IO (Ptr a)
new val = do ptr <- malloc
             poke ptr val
             return ptr
```

Функция: `fromBool`
Описание: преобразует значение типа `Bool` из языка Haskell в числовое значение, используемое в языках программирования типа C.
Определение:

```
fromBool :: Num a => Bool -> a
fromBool False = 0
fromBool True  = 1
```

Функция: `toBool`
Описание: преобразует числовое представление булевского значения истинности, используемое в таких языках, как C, в значение типа `Bool`.
Определение:

```
toBool :: Num a => a -> Bool
toBool = (/= 0)
```

Функция: maybeNew

Описание: выделяет память и маршализует значение, обёрнутое монадой Maybe. Для представления значения Nothing используется пустой указатель nullPtr.

Определение:

```
maybeNew :: (a -> IO (Ptr a)) -> (Maybe a -> IO (Ptr a))
maybeNew = maybe (return nullPtr)
```

Функция: maybeWith

Описание: преобразует комбинатор семейства with* в функцию, работающую со значениями типа Maybe. Для представления значения Nothing используется пустой указатель nullPtr.

Определение:

```
maybeWith :: (a -> (Ptr b -> IO c) -> IO c) -> (Maybe a -> (Ptr b -> IO c) -> IO c)
maybeWith = maybe ($ nullPtr)
```

Функция: maybePeek

Описание: преобразует комбинатор семейства peek* в функцию, работающую со значениями типа Maybe. Для представления значения Nothing используется пустой указатель nullPtr.

Определение:

```
maybePeek :: (Ptr a -> IO b) -> Ptr a -> IO (Maybe b)
maybePeek peek ptr | ptr == nullPtr = return Nothing
           | otherwise      = do a <- peek ptr
                                 return (Just a)
```

Функция: withMany

Описание: применяет комбинатор семейства with* на список значений, выдавая на выходе список маршализованных объектов.

Определение:

```
withMany :: (a -> (b -> res) -> res) -> [a] -> ([b] -> res) -> res
withMany _       []     f = f []
withMany withFoo (x:xs) f = withFoo x $
                            \x' -> withMany withFoo xs (\xs' -> f (x':xs'))
```

Функция: copyBytes
Описание: копирует заданное количество байт из второй области памяти (второй аргумент) в первую (первый аргумент). Области копирования *не могут* пересекаться. Аналог (обёртка над) функции memcpy.
Определение:

```
copyBytes :: Ptr a -> Ptr a -> Int -> IO ()
copyBytes dest src size = memcpy dest src (fromIntegral size)
```

Функция: moveBytes
Описание: копирует заданное количество байт из второй области памяти (второй аргумент) в первую (первый аргумент). Области копирования *могут* пересекаться. Аналог (обёртка над) функции memmove.
Определение:

```
moveBytes :: Ptr a -> Ptr a -> Int -> IO ()
moveBytes dest src size = memmove dest src (fromIntegral size)
```

10.4. Модуль Ptr

Модуль Ptr предлагает для работы определения типизированных указателей на области памяти, которые используются для хранения внешних данных, описанных на других языках программирования. Этот модуль является частью технологии FFI и должен подключаться в проект при помощи реимпорта из модуля Foreign, однако если имеется необходимость использования этого модуля отдельно, его можно подключить следующим образом:

```
import Foreign.Ptr
```

Все программные сущности в этом модуле определены в виде примитивов, поэтому ниже этот факт указываться не будет, а для описываемых функций приводятся только сигнатуры.
Тип: Ptr
Описание: указатель на объект или массив объектов, который может быть маршализован в представление языка Haskell или из него для типа a. Обычно этот тип a является экземпляром класса Storable (см. стр. 468), который определяет

некоторые методы для маршалинга. Однако это требование не является обязательным.

Определение:

```
data Ptr a = ...
```

Этот тип является экземпляром следующих классов: Typeable1, IArray, MArray, Data, Eq, Ord, Show и Storable.

Функция: nullPtr

Описание: возвращает нулевой указатель, который обычно используется в целях маркировки. Этот указатель не указывает на какую-либо память.

Определение:

```
nullPtr :: Ptr a
```

Функция: castPtr

Описание: преобразует указатель с одного типа данных на другой.

Определение:

```
castPtr :: Ptr a -> Ptr b
```

Функция: plusPtr

Описание: сдвигает указатель на заданное количество байт.

Определение:

```
plusPtr :: Ptr a -> Int -> Ptr b
```

Функция: alignPtr

Описание: для заданных произвольного адреса и ограничения на выравнивание эта функция возвращает следующий адрес, удовлетворяющий ограничению.

Определение:

```
alignPtr :: Ptr a -> Int -> Ptr a
```

Функция: minusPtr

Описание: возвращает разницу (в байтах) между двумя указателями.

Определение:

```
minusPtr :: Ptr a -> Ptr b -> Int
```

Тип: `FunPtr`
Описание: указатель на функцию, которая может быть вызвана из стороннего кода (кода, написанного на ином языке программирования, нежели Haskell). Параметризуемый тип `a` обычно представляет собой внешний тип. При этом функция, которая находится по указателю, принимает на вход ноль или более аргументов только маршализуемых типов, а возвращает либо маршализуемый тип, либо его же, но обёрнутый в монаду `IO`.
Определение:

```
data FunPtr a = ...
```

Значением типа `FunPtr` может быть либо функция, вычисленная в другой функции, либо статическая декларация импорта из внешнего модуля. Например, подобного вида:

```
foreign import ccal "stdlib.h &free" p_free :: FunPtr (Ptr a -> IO ())
```

Тип `FunPtr` является экземпляром следующих классов: `Typeable1`, `IArray`, `MArray`, `Eq`, `Ord`, `Show` и `Storable`.
Функция: `nullFunPtr`
Описание: такой же нулевой указатель, как и `nullPtr`, который не указывает на какое-либо место в памяти.
Определение:

```
nullFunPtr :: FunPtr a
```

Функция: `castFunPtr`
Описание: преобразует указатель на функцию одного типа в указатель на функцию другого типа.
Определение:

```
castFunPtr :: FunPtr a -> FunPtr b
```

Функция определена в виде примитива.

Функция: `castFunPtrToPtr`
Описание: преобразует указатель на функцию в простой указатель. Необходимо отметить, что эта функция валидна только на архитектурах, где данные и функции находятся в одном пространстве имён.

Определение:

```
castFunPtrToPtr :: FunPtr a -> Ptr b
```

Функция: `castPtrToFunPtr`
Описание: преобразует простой указатель в указатель на функцию. Необходимо отметить, что эта функция валидна только на архитектурах, где данные и функции находятся в одном пространстве имён.
Определение:

```
castPtrToFunPtr :: Ptr a -> FunPtr b
```

Функция: `freeHaskellFunPtr`
Описание: высвобождает память, занимаемую под функцию. Эта функция должна вызываться всегда, когда внешняя функция больше не требуется, поскольку в противном случае могут произойти утечки памяти.
Определение:

```
freeHaskellFunPtr :: FunPtr a -> IO ()
```

Тип: `IntPtr`
Описание: указатель на знаковый целочисленный тип, который может быть получен при помощи «забывающего» (необратимого) преобразования из простого указателя.
Определение:

```
data IntPtr = ...
```

Данный тип является экземпляром следующих классов: `Bits`, `Bounded`, `Enum`, `Eq`, `Integral`, `Num`, `Ord`, `Read`, `Real`, `Show`, `Storable` и `Typeable`.
Функция: `ptrToIntPtr`
Описание: преобразует простой указатель в указатель на знаковый целочисленный тип.
Определение:

```
ptrToIntPtr :: Ptr a -> IntPtr
```

Функция: `intPtrToPtr`
Описание: преобразует указатель на знаковый целочисленный тип в простой указатель.

Определение:

```
intPtrToPtr :: IntPtr -> Ptr a
```

Тип: `WordPtr`
Описание: указатель на беззнаковый целочисленный тип, который может быть получен при помощи «забывающего» (необратимого) преобразования из простого указателя.
Определение:

```
data WordPtr = ...
```

Данный тип является экземпляром следующих классов: `Bits`, `Bounded`, `Enum`, `Eq`, `Integral`, `Num`, `Ord`, `Read`, `Real`, `Show`, `Storable` и `Typeable`.
Функция: `ptrToWordPtr`
Описание: преобразует простой указатель в указатель на беззнаковый целочисленный тип.
Определение:

```
ptrToWordPtr :: Ptr a -> WordPtr
```

Функция: `wordPtrToPtr`
Описание: преобразует указатель на беззнаковый целочисленный тип в простой указатель.
Определение:

```
wordPtrToPtr :: WordPtr -> Ptr a
```

10.5. Модуль `StablePtr`

Модуль `StablePtr` содержит определения программных сущностей, используемых для работы со стабильными указателями, то есть такими, которые не уничтожаются автоматически сборщиком мусора, равно как и не меняют своего местоположения во время процесса сборки мусора (обычные указатели могут поменять своё местоположение во время сборки мусора). Такие стабильные указатели могут использоваться во внешних программах в качестве непрозрачных ссылок на значения языка Haskell.

Предполагается, что этот модуль подключается в проект при помощи реимпорта из модуля Foreign, однако если имеется необходимость использования этого модуля отдельно, его можно подключить следующим образом:

```
import Foreign.stablePtr
```

Все программные сущности в этом модуле определены в виде примитивов, поэтому ниже этот факт указываться не будет, а для описываемых функций приводятся только сигнатуры.

Главный тип, описываемый в этом модуле, — StablePtr.

Тип: StablePtr

Описание: значение этого типа является стабильным указателем на область памяти, в которой содержится значение типа a.

Определение:

```
data StablePtr a = ...
```

Для этого типа определены экземпляры следующих классов: Typeable1, IArray, MArray, Data, Eq и Storable.

Функция: newStablePtr

Описание: создаёт новый стабильный указатель, по которому записывается заданное значение.

Определение:

```
newStablePtr :: a -> IO (StablePtr a)
```

Функция: deRefStablePtr

Описание: получает значение, хранящееся по адресу стабильного указателя. Если память по указателю уже высвобождена, то поведение этой ссылки не определено.

Определение:

```
deRefStablePtr :: StablePtr a -> IO a
```

Функция: freeStablePtr

Описание: высвобождает память из-под стабильного указателя.

Определение:

```
freeStablePtr :: StablePtr a -> IO ()
```

Функция: `castStablePtrToPtr`
Описание: преобразует стабильный указатель в обыкновенный. Функция не даёт никаких гарантий относительно результата, за исключением того, что применение к результату функции `castPtrToStablePtr` даст первоначальный результат.
Определение:

```
castStablePtrToPtr :: StablePtr a -> Ptr ()
```

Функция: `castPtrToStablePtr`
Описание: обращение функции `castStablePtrToPtr`.
Определение:

```
castPtrToStablePtr :: Ptr () -> StablePtr a
```

10.6. Модуль `Storable`

В модуле `Storable` описан важнейший интерфейс для маршализации — класс `Storable`. Этот класс предоставляет интерфейсные функции для осуществления примитивного маршалинга типов данных, а потому является непосредственной частью системы FFI языка Haskell. Предполагается, что этот модуль подключается в проект при помощи реимпорта из модуля `Foreign`, однако если имеется необходимость использования этого модуля отдельно, его можно подключить следующим образом:

```
import Foreign.storable
```

В модуле описан единственный класс, ничего более.

Класс: `Storable`
Описание: методы этого класса облегчают запись значений примитивных типов в последовательную память (которая может быть выделена при помощи функций, описанных ранее в этой главе), а также чтение значений из последовательной памяти. Кроме того, этот класс включает в себя методы для вычисления необходимого количества памяти, а также для получения ограничений на выравнивание значений.

Определение:

```
class Storable a where
  sizeOf      :: a -> Int
  alignment   :: a -> Int
  peekElemOff :: Ptr a -> Int -> IO a
  pokeElemOff :: Ptr a -> Int -> a -> IO ()
  peekByteOff :: Ptr b -> Int -> IO a
  pokeByteOff :: Ptr b -> Int -> a -> IO ()
  peek        :: Ptr a -> IO a
  poke        :: Ptr a -> a -> IO ()
```

Адреса памяти представлены в виде указателей типа `Ptr` (см. стр. 462) на типы данных, являющиеся экземплярами рассматриваемого класса. Такие указатели помогают обеспечить определённый уровень безопасности при работе с FFI (невозможно использовать указатель на один тип в качестве указателя на другой тип без явного преобразования типов). Кроме того, такие указатели помогают системе типизации языка Haskell применять определённые методы маршализации для заданных типов данных.

Все процессы маршализации между языком Haskell и прочими сторонними языками программирования заключаются в преобразовании алгебраических типов данных языка Haskell в неструктурированные последовательности байт, и наоборот. Для кодирования таких процессов маршализации необходимо манипулировать двоичным представлением значений в последовательной памяти. Класс `Storable` как раз и предлагает интерфейс для подобной манипуляции.

Метод `sizeOf` возвращает количество байт, необходимое для хранения значения заданного типа. Значение аргумента не используется в вычислениях, поэтому в качестве значения можно (и обычно это делается) передавать значение `undefined` соответствующего типа. Также и метод `alignment` возвращает количество байт, необходимых для выравнивания значений в памяти.

Пара методов `peekElemOff` и `pokeElemOff` используются для чтения и записи элементов массива в последовательной памяти. Первый аргумент задаёт указатель на начало массива, второй — смещение в элементах массива (не в байтах). Третий аргумент метода `pokeElemOff`, соответственно, задаёт значение, которое необходимо записать. Результат этих методов возвращается в монаде `IO`.

Абсолютно так же работает пара методов `peekByteOff` и `pokeByteOff`, которые вторым аргументом принимают смещение от начала адреса в байтах. Результат этих методов также возвращается в монаде `IO`.

Наконец, методы `peek` и `poke` используются для чтения и записи значения по определённому адресу в памяти. Предыдущие две пары методов могут быть выражены через эти примитивные функции. Обычно в экземплярах класса `Storable` определяют только методы `peek` и `poke`.

Осталось отметить, что экземплярами класса `Storable` определаны все примитивные типы языка Haskell, все внешние типы, описанные в модуле `Types` (см. подраздел 10.1.3.), а также все целочисленные типы для представления знаковых и беззнаковых величин различной точности. Также в качестве экземпляров определены указатели на функции (`FunPtr`), простые (`Ptr`) и стабильные (`StablePtr`) указатели. Само собой разумеется, что типы, параметризуемые такими указателями, сами должны быть экземплярами класса `Storable`.

Глава 11.

Пакет модулей System

Пакет модулей System включает в себя широчайший набор модулей, в которых собраны средства, позволяющие программисту оперировать с системными возможностями компьютеров и операционных систем. Поскольку все эти модули стандартные, приведённые в них определения не зависят от платформы, а потому могут использоваться там, где может использоваться сам язык программирования Haskell.

11.1. Модуль Cmd

В модуле Cmd определены две функции, предназначенные для выполнения внешних команд. Использование модуля:

```
import System.Cmd
```

Функция: system
Описание: возвращает код исполнения команды операционной системы, которая передаётся в эту функцию при помощи имени команды. Этот вызов может завершиться неудачей, возвратив один из следующих кодов: PermissionDenied — процесс не имеет прав доступа для осуществления операции; ResourcesExhausted — недостаточно ресурсов для выполнения операции; UnsupportedOperation — в операционной системе нет заданной команды.

Определение:

```
system :: String -> IO ExitCode
```

Функция определена в виде примитива.

Функция: rawSystem

Описание: вариант функции system, который передаёт команде операционной системы набор аргументов, который закодирован в виде списка.

Определение:

```
rawSystem :: String -> [String] -> IO ExitCode
rawSystem cmd args = system (unwords (cmd : map translate args))
```

11.2. Модуль CPUTime

Модуль CPUTime содержит определения функций, которые позволяют программисту на языке Haskell обратиться к времени использования процессора. Использование модуля:

```
import System.CPUTime
```

Модуль содержит две функции, которые позволяют получить время использования процессора.

Функция: getCPUTime

Описание: возвращает количество пикосекунд, которое используется текущей программой в общем массиве процессорного времени. Точность этого значения зависит от реализации. Функция возвращает значение в монаде IO, поскольку значение недетерминировано (может отличаться от вызова к вызову).

Определение:

```
getCPUTime :: IO Integer
getCPUTime = do (usec, unsec, ssec, snsec) <- getCPUUsage
                return (picoSec * fromIntegral usec  +
                        1000 * fromIntegral unsec +
                        picoSec * fromIntegral ssec  +
                        1000 * fromIntegral snsec)
```

Функция: cpuTimePrecision
Описание: константа, определяющая минимальное количество времени, которое релевантно в процессе вычисления процессорного времени, занимаемого текущей программой. Ниже приведена реализация функции для системы HUGS 98.
Определение:

```
cpuTimePrecision :: Integer
cpuTimePrecision = round ((1000000000000::Integer) % fromIntegral (clockTicks))
```

11.3. Модуль Directory

В модуле Directory определены системонезависимые функции для работы с каталогами файловой системы. Использование модуля:

```
import System.Directory
```

Каждый идентификатор каталога содержит набор элементов, являющихся поименованными указателями на некоторый объект в файловой системе (файлы, каталоги и т. д.). Некоторые такие объекты могут быть скрыты, недоступны или иметь определённые административные функции (например, объекты «.» или «..»). Однако в данном модуле все элементы идентификатора каталога рассматриваются в качестве содержимого каталога, за исключением содержимого подкаталогов текущего каталога.

К произвольному элементу файловой структуры можно обратиться при помощи *пути*. В большинстве операционных систем к каждому объекту имеется по крайней мере один абсолютный путь. В некоторых операционных системах можно использовать относительные пути.

Функция: createDirectory
Описание: создаёт новый каталог по заданному пути. Этот каталог изначально является пустым (настолько, насколько позволяет операционная система). Данная функция может завершиться неуспешно в случаях: у процесса недостаточно прав доступа для создания каталога (isPermissionError или PermissionDenied); аргумент указывает на каталог, который уже существует (isAlreadyExistsError или AlreadyExists); произошла ошибка ввода/вывода на физическом уровне (HardwareFault); аргумент не является валидным идентификатором каталога (InvalidArgument); не существует пути к создаваемому

каталогу (NoSuchThing); недостаточно памяти или других ресурсов для создания каталога (ResourceExhausted); и наконец, заданный путь указывает на объект, который не является каталогом (InappropriateType).
Определение:

```
createDirectory :: FilePath -> IO ()
```

Функция определена в виде примитива.
Функция: createDirectoryIfMissing
Описание: создаёт новый каталог, если его ещё не существует. Если первый аргумент принимает значение True, то эта функция также создаёт все родительские каталоги, если они отсутствуют.
Определение:

```
createDirectoryIfMissing :: Bool -> FilePath -> IO ()

createDirectoryIfMissing parents file
  = do b <- doesDirectoryExist file
       case (b, parents, file) of
         (_,     _, "") -> return ()
         (True,  _,  _) -> return ()
         (_,  True,  _) -> mapM_ (createDirectoryIfMissing False)
                                 (tail (pathParents file))
         (_, False,  _) -> createDirectory file
```

Функция: removeDirectory
Описание: удаляет существующий каталог, заданный при помощи пути. Реализация и операционная система могут требовать выполнения дополнительных ограничений перед удалением каталога (например, такое ограничение, что удаляемый каталог должен быть пуст). Данная функция может завершиться неуспешно в случаях: произошла ошибка ввода/вывода на физическом уровне (HardwareFault); аргумент не является валидным идентификатором каталога (InvalidArgument); удаляемого каталога не существует (isDoesNotExistError или NoSuchThing); у процесса недостаточно прав доступа, чтобы удалить каталог (isPermissionErrorPermissionDenied или PermissionDenied); невыполненные требования для удаления каталога (UnsatisfiedConstraints); операция по удалению не поддерживается (UnsupportedOperation); и наконец, аргумент указывает на объект, не являющийся каталогом (InappropriateType).

Определение:

```
removeDirectory :: FilePath -> IO ()
```

Функция определена в виде примитива.

Функция: `removeDirectoryRecursive`
Описание: аналог функции `removeDirectory`, который удаляет заданный каталог рекурсивно со всеми его подкаталогами и файлами.
Определение:

```
removeDirectoryRecursive :: FilePath -> IO ()
removeDirectoryRecursive startLoc
  = do cont <- getDirectoryContents startLoc
       sequence_ [rm (startLoc `joinFileName` x) | x <- cont, x /= "." && x /= ".."]
       removeDirectory startLoc
  where
    rm :: FilePath -> IO ()
    rm f = do temp <- try (removeFile f)
              case temp of
                Left e  -> do isDir <- doesDirectoryExist f
                              unless isDir $ ioError e
                              removeDirectoryRecursive f
                Right _ -> return ()
```

Функция: `renameDirectory`
Описание: переименовывает заданный каталог. Старый каталог определяется первым аргументом, новый — вторым. Если каталог, заданный новым идентификатором, уже существует, он заменяется старым каталогом. Данная функция может завершиться неуспешно в случаях: произошла ошибка ввода/вывода на физическом уровне (`HardwareFault`); аргумент не является валидным идентификатором каталога (`InvalidArgument`); переименовываемого каталога не существует (`isDoesNotExistError` или `NoSuchThing`); у процесса недостаточно прав доступа, чтобы переименовать каталог (`isPermissionErrorPermissionDenied` или `PermissionDenied`); невыполненные требования для переименования каталога (`UnsatisfiedConstraints`); операция по переименованию не поддерживается (`UnsupportedOperation`); и наконец, аргумент указывает на объект, не являющийся каталогом (`InappropriateType`).

Определение:

```
renameDirectory :: FilePath -> FilePath -> IO ()
```

Функция определена в виде примитива.

Функция: `getDirectoryContents`

Описание: возвращает список всех объектов в заданном каталоге. Данная функция может завершиться неуспешно в случаях: произошла ошибка ввода/вывода на физическом уровне (`HardwareFault`); аргумент не является валидным идентификатором каталога (`InvalidArgument`); запрашиваемого каталога не существует (`isDoesNotExistError` или `NoSuchThing`); у процесса недостаточно прав доступа, чтобы просканировать каталог (`isPermissionErrorPermissionDenied` или `PermissionDenied`); недостаточно памяти или других ресурсов для выполнения операции (`ResourceExhausted`); и наконец, аргумент указывает на объект, не являющийся каталогом (`InappropriateType`).

Определение:

```
getDirectoryContents :: FilePath -> IO [FilePath]
```

Функция определена в виде примитива.

Функция: `getCurrentDirectory`

Описание: если в операционной системе используется понятие текущего каталога, то эта функция возвращает путь к нему. Данная функция может завершиться неуспешно в случаях: произошла ошибка ввода/вывода на физическом уровне (`HardwareFault`); запрашиваемого каталога не существует (`isDoesNotExistError` или `NoSuchThing`); у процесса недостаточно прав доступа, чтобы получить каталог (`isPermissionErrorPermissionDenied` или `PermissionDenied`); недостаточно памяти или других ресурсов для выполнения операции (`ResourceExhausted`); и наконец, аргумент указывает на объект, не являющийся каталогом (`InappropriateType`).

Определение:

```
getCurrentDirectory :: IO FilePath
```

Функция определена в виде примитива.

Функция: `setCurrentDirectory`

Описание: если в операционной системе используется понятие текущего каталога, то эта функция устанавливает новый текущий каталог по заданному пути. Дан-

ная функция может завершиться неуспешно в случаях: произошла ошибка ввода/вывода на физическом уровне (HardwareFault); аргумент не является валидным идентификатором каталога (InvalidArgument); запрашиваемого каталога не существует (isDoesNotExistError или NoSuchThing); у процесса недостаточно прав доступа, чтобы сохранить каталог (isPermissionErrorPermissionDenied или PermissionDenied); недостаточно памяти или других ресурсов для выполнения операции (ResourceExhausted); и наконец, аргумент указывает на объект, не являющийся каталогом (InappropriateType).
Определение:

```
setCurrentDirectory :: FilePath -> IO ()
```

Функция определена в виде примитива.
Функция: getHomeDirectory
Описание: возвращает домашний каталог текущего пользователя (если такое понятие используется в текущей операционной системе). Данная функция может завершиться неуспешно в случаях: операция не поддерживается (UnsupportedOperation); получаемого каталога не существует (isDoesNotExistError или NoSuchThing).
Определение:

```
getHomeDirectory :: IO FilePath
```

Функция определена в виде примитива.
Функция: getAppUserDataDirectory
Описание: возвращает каталог текущего пользователя для хранения приложений и релевантных данных (если такое понятие используется в текущей операционной системе). Данная функция может завершиться неуспешно в случаях: операция не поддерживается (UnsupportedOperation); получаемого каталога не существует (isDoesNotExistError или NoSuchThing).
Определение:

```
getAppUserDataDirectory :: String -> IO FilePath
```

Функция определена в виде примитива.
Функция: getUserDocumentsDirectory
Описание: возвращает каталог текущего пользователя для хранения документов (если такое понятие используется в текущей операционной системе).

Данная функция может завершиться неуспешно в случаях: операция не поддерживается (UnsupportedOperation); получаемого каталога не существует (isDoesNotExistError или NoSuchThing).
Определение:

```
getUserDocumentsDirectory :: IO FilePath
```

Функция определена в виде примитива.
Функция: getTemporaryDirectory
Описание: возвращает каталог текущего пользователя, используемый для хранения временных данных (если такое понятие используется в текущей операционной системе). Данная функция может завершиться неуспешно в случае, если операция не поддерживается (UnsupportedOperation).
Определение:

```
getTemporaryDirectory :: IO FilePath
```

Функция определена в виде примитива.
Функция: removeFile
Описание: удаляет каталог для заданного файла (который сам по себе каталогом не является). Реализация и операционная система могут требовать выполнения дополнительных ограничений перед удалением каталога. Данная функция может завершиться неуспешно в случаях: произошла ошибка ввода/вывода на физическом уровне (HardwareFault); аргумент не является валидным идентификатором каталога (InvalidArgument); удаляемого каталога не существует (isDoesNotExistError или NoSuchThing); у процесса недостаточно прав доступа, чтобы удалить каталог (isPermissionErrorPermissionDenied или PermissionDenied); невыполненные требования для удаления каталога (UnsatisfiedConstraints); и наконец, аргумент указывает на объект, не являющийся каталогом (InappropriateType).
Определение:

```
removeFile :: FilePath -> IO ()
```

Функция определена в виде примитива.
Функция: renameFile
Описание: изменяет идентификатор некоторого существующего объекта файловой системы на новый. Если объект, заданный новым идентификатором,

уже существует, он заменяется старым объектом. Данная функция может завершиться неуспешно в случаях: произошла ошибка ввода/вывода на физическом уровне (HardwareFault); аргумент не является валидным идентификатором объекта (InvalidArgument); переименовываемого объекта не существует (isDoesNotExistError или NoSuchThing); у процесса недостаточно прав доступа, чтобы переименовать объект (isPermissionErrorPermissionDenied или PermissionDenied); недостаточно ресурсов для переименовывания объекта (ResourceExhausted); невыполненные требования для переименования каталога (UnsatisfiedConstraints); операция по переименованию не поддерживается (UnsupportedOperation); и наконец, аргумент указывает на объект, не являющийся каталогом (InappropriateType).

Определение:

```
renameFile :: FilePath -> FilePath -> IO ()
```

Функция определена в виде примитива.

Функция: copyFile

Описание: копирует существующий файл, заданный первым аргументом, по пути, заданному вторым аргументом. Если объект по новому имени файла существует, он заменяется содержимым старого объекта. Идентификатор каталога не может быть новым именем файла. Права доступа по возможности также копируются вместе с файлом.

Определение:

```
copyFile :: FilePath -> FilePath -> IO ()
copyFile fromFPath toFPath = do readFile fromFPath >>= writeFile toFPath
                                try (copyPermissions fromFPath toFPath)
                                return ()
```

Функция: canonicalizePath

Описание: для заданного пути возвращает канонический путь, что подразумевает, что для двух разных путей к одному и тому же объекту файловой системы канонический путь будет один.

Определение:

```
canonicalizePath :: FilePath -> IO FilePath
canonicalizePath fpath = return fpath
```

Функция: `findExecutable`

Описание: для заданного имени файла ищет упоминание такового в переменной окружения `PATH`, возвращая, в случае наличия, абсолютный путь к файлу. Возвращает значение `Nothing`, если ничего не найдено.

Определение:

```
findExecutable :: String -> IO (Maybe FilePath)
findExecutable binary
  = withCString binary $
    \c_binary -> withCString ('.':exeExtension) $
    \c_ext -> allocaBytes long_path_size $
    \pOutPath -> alloca $
    \ppFilePart -> do res <- c_SearchPath nullPtr c_binary c_ext
                                (fromIntegral long_path_size) pOutPath ppFilePart
                      if res > 0 && res < fromIntegral long_path_size
                        then do fpath <- peekCString pOutPath
                                return (Just fpath)
                        else return Nothing
```

Функция: `doesFileExist`

Описание: возвращает значение `IO Bool` в случае, если объект по заданному пути существует, а также не является каталогом.

Определение:

```
doesFileExist :: FilePath -> IO Bool
```

Функция определена в виде примитива.

Функция: `doesDirectoryExist`

Описание: возвращает значение `IO Bool` в случае, если объект по заданному пути существует, а также не является файлом.

Определение:

```
doesDirectoryExist :: FilePath -> IO Bool
```

Функция определена в виде примитива.

Тип: `Permissions`

Описание: тип, представляющий набор прав доступа к определённому объекту файловой системы. Единственный одноимённый конструктор параметризует четыре поля типа `Bool`. Первое поле определяет флаг, возводимый тогда, когда объект доступен для чтения. Второй поле — флаг доступности для записи.

Третье поле — флаг, указывающий на то, что объект является исполняемым. Наконец, четвёртое поле — флаг, указывающий на то, что объект можно найти средствами операционной системы.
Определение:

```
data Permissions
  = Permissions
    {
      readable, writable, executable, searchable :: Bool
    }
```

Данный тип является экземпляром следующих классов: `Eq`, `Ord`, `Read` и `Show`.
Функция: `getPermissions`
Описание: по заданному пути к объекту файловой системы возвращает набор прав доступа к нему. Данная функция может завершиться неуспешно в случаях: у пользователя нет доступа для получения прав доступа (`isPermissionError`); объект файловой системы не существует (`isDoesNotExistError`).
Определение:

```
getPermissions :: FilePath -> IO Permissions
```

Функция определена в виде примитива.
Функция: `setPermissions`
Описание: по заданному пути к объекту файловой системы устанавливает набор прав доступа к нему. Данная функция может завершиться неуспешно в случаях: у пользователя нет доступа для установки прав доступа (`isPermissionError`); объект файловой системы не существует (`isDoesNotExistError`).
Определение:

```
setPermissions :: FilePath -> Permissions -> IO ()
```

Функция определена в виде примитива.

Функция: `getModificationTime`
Описание: по заданному пути к объекту файловой системы возвращает дату и время последней модификации этого объекта. Данная функция может завершиться неуспешно в случаях: у пользователя нет доступа для получения этой информации (`isPermissionError`); объект файловой системы не существует (`isDoesNotExistError`).

Определение:

```
getModificationTime :: FilePath -> IO ClockTime
```

Функция определена в виде примитива.

11.3.1. Модуль Internals

Служебный модуль `Internals`, в котором определяются функции для работы с путями, используемые в модуле `Directory`. Не должен использоваться напрямую, поскольку он сам всегда включён в модуль `Directory` из пакета `System`.

11.4. Модуль Environment

В модуле `Environment` определены некоторые функции для работы программным окружением и с переменными, заданными в нём. Использование модуля:

```
import System.Environment
```

В модуле определено шесть функций для чтения и работы с содержимым окружения.

Функция: `getArgs`

Описание: возвращает список аргументов командной строки, которые были использованы при запуске программы. В список не включается наименование самой программы.

Определение:

```
getArgs :: IO [String]
```

Функция определена в виде примитива.

Функция: `getProgName`

Описание: возвращает имя файла программы, который был запущен в операционной системе. Результат может отличаться в зависимости от типа операционной системы.

Определение:

```
getProgName :: IO String
```

Функция определена в виде примитива.

Функция: `getEnv`
Описание: возвращает значение заданной переменной окружения. Если переменной окружения не существует, генерируется исключение `isDoesNotExistError`.
Определение:

```
getEnv :: String -> IO String
```

Функция определена в виде примитива.
Функция: `withArgs`
Описание: выполняет монадическое действие ввода/вывода (второй аргумент), во время которого функция `getArgs` вернёт значение, заданное первым аргументом.
Определение:

```
withArgs :: [String] -> IO a -> IO a
```

Функция определена в виде примитива.
Функция: `withProgName`
Описание: выполняет монадическое действие ввода/вывода (второй аргумент), во время которого функция `getProgName` вернёт значение, заданное первым аргументом.
Определение:

```
withProgName :: String -> IO a -> IO a
```

Функция определена в виде примитива.
Функция: `getEnvironment`
Описание: возвращает весь набор переменных окружения в виде пар (*имя*, *значение*). Если переменная не содержит значения, то в соответствующей паре второй элемент является пустой строкой.
Определение:

```
getEnvironment :: IO [(String, String)]
```

Функция определена в виде примитива.

11.5. Модуль Exit

Модуль Exit содержит определения программных сущностей, которые используются для обработки факта выхода из программы. Использование модуля:

```
import System.Exit
```

В модуле описан один алгебраический тип данных, а также две функции, которые его используют.

Тип: ExitCode

Описание: представляет значения кода выхода из программы. Первый конструктор представляет успешное завершение программы, в то время как второй — неуспешное. Второй конструктор параметризует целочисленное значение, которое используется операционной системой для понимания причины, по которой программа была завершена с ошибкой (код ошибки).

Определение:

```
data ExitCode
  = ExitSuccess
  | ExitFailure Int
```

Данный тип имеет экземпляры следующих классов: Eq, Ord, Read и Show.

Функция: exitWith

Описание: генерирует исключение ExitException с заданным кодом ошибки. Обычно это завершает программу. Перед тем как программа будет завершена, все открытые источники будут закрыты. Любая программа, которая завершается иным способом, трактуется как завершённая функцией exitFailure. Программа, которая завершена успешно, трактуется как завершённая при помощи команды exitWith ExitSuccess.

Определение:

```
exitWith :: ExitCode -> IO a
exitWith ExitSuccess = throwIO (ExitException ExitSuccess)
exitWith code@(ExitFailure n) | n /= 0 = throwIO (ExitException code)
```

Поскольку тип ExitException не является ошибкой ввода/вывода IOError, он не может быть обработан функцией catch (см. стр. 506). Однако это исключение в любом случае является типом Exception, поэтому может обрабатываться силами модуля Exception (см. раздел 7.4.). Это значит, что произвольные завер-

шающие вычисления выполняются в этом случае также при помощи функции `bracket` (см. стр. 209).

Функция: `exitFailure`

Описание: эквивалент функции `exitWith`, который возвращает код ошибки, зависящий от реализации библиотеки.

Определение:

```
exitFailure :: IO a
exitFailure = exitWith (ExitFailure 1)
```

11.6. Модуль `Info`

Модуль, содержащий функции для получения информации об операционной системе, в которой была запущена программа. Использование модуля:

```
import System.Info
```

В модуле описано четыре функции, которые в большинстве случаев будут достаточны для работы программы с возвращаемой информацией.

Функция: `os`

Описание: возвращает наименование операционной системы, в которой выполняется программа.

Определение:

```
os :: String
```

Функция определена в виде примитива.

Функция: `arch`

Описание: возвращает наименование машинной архитектуры, в которой выполняется программа.

Определение:

```
arch :: String
```

Функция определена в виде примитива.

Функция: `compilerName`

Описание: возвращает наименование транслятора языка Haskell, при помощи которого была создана программа.

Определение:

```
compilerName :: String
```

Функция определена в виде примитива.

Функция: `compilerVersion`

Описание: возвращает версию транслятора языка Haskell, при помощи которого была создана программа. О типе `Version` см. раздел 8.33..

Определение:

```
compilerVersion :: Version
```

Функция определена в виде примитива.

11.7. Модуль `IO`

Модуль `IO` является экспериментальным, созданным для разгрузки стандартного модуля `Prelude` от многочисленных определений. Разумеется, что в этот модуль собраны определения всех программных сущностей, связанных с системой ввода/вывода языка Haskell. Сюда перенесены все стандартные функции и типы, а также определены новые, которых нет в стандартном модуле `Prelude`. С другой стороны, все функции из стандартного модуля `Prelude`, которые имеют отношение к вводу/выводу, также определены в этом модуле, поэтому ниже рассматриваться не будут.

В модуле `IO` практически все функции определены в виде примитивов. По этой причине в нижеследующем описании для всех примитивных функций будут опускаться слова об этом, а приводиться будет только сигнатура функций. Это также касается тех функций, которые определены через примитивы, основная задача которых — обёртка примитивных функций.

Использование модуля:

```
import System.IO
```

В модуле определено гигантское количество всевозможных функций для работы с вводом/выводом. Все они связаны с монадой `IO`, в которой исключительно выполняются действия ввода/вывода, поскольку такие действия зачастую недетерминированы и имеют побочные эффекты.

Тип: `FilePath`
Описание: синоним типа `String` для удобства именования типа, представляющих пути к файлам.
Определение:

```
type FilePath = String
```

Тип: `Handle`
Описание: примитивный тип для представления дескрипторов источников или пунктов назначения информации в системе ввода/вывода. Такими источниками могут быть файлы или потоки.
Определение:

```
data Handle = ...
```

Тип определён в виде примитива.

Для типа `Handle` определены экземпляры классов `Data`, `Eq`, `Show` и `Typeable`.

Функция: `stdin`
Описание: возвращает стандартный поток ввода.
Определение:

```
stdin :: Handle
```

Функция: `stdout`
Описание: возвращает стандартный поток вывода.
Определение:

```
stdout :: Handle
```

Функция: `stderr`
Описание: возвращает стандартный поток вывода сообщений об ошибках.
Определение:

```
stderr :: Handle
```

Функция: `openFile`
Описание: открывает заданный по имени файл в заданном режиме, возвращая дескриптор этого файла. Функция может сгенерировать исключение в следующих случаях: файл уже открыт и не может быть открыт повторно (`isAlreadyInUseError`); файл не существует (`isDoesNotExistError`); а также ес-

ли нет прав доступа для открытия файла (isPermissionError). Если имеется необходимость работать с бинарными файлами, то вместо этой функции необходимо пользоваться функцией openBinaryFile (см. стр. 495).
Определение:

```
openFile :: FilePath -> IOMode -> IO Handle
```

Тип: IOMode
Описание: тип для представления режимов открытия файла. Каждый конструктор в этом перечислении представляет один из доступных режимов: на чтение, на запись, на дописывание информации в файл, а также на одновременное чтение и запись.
Определение:

```
data IOMode
  = ReadMode
  | WriteMode
  | AppendMode
  | ReadWriteMode
```

Функция: hClose
Описание: закрывает заданный по дескриптору файл. Перед тем как действие по закрытию выполнится, все буферизованные данные будут записаны в файл так, как будто бы запущена функция hFlush (см. стр. 490). Попытка закрытия файла, который уже закрыт, ни к чему не приводит и не генерирует исключений. Все другие действия с закрытыми файлами приведут к исключениям. Если же по какой-либо причине во время выполнения этой функции происходит исключение, файл закрывается в любом случае (но не факт, что буфферизованные данные в него будут записаны).
Определение:

```
hClose :: Handle -> IO ()
```

Функция: hFileSize
Описание: возвращает размер заданного файла в байтах.
Определение:

```
hFileSize :: Handle -> IO Integer
```

Функция: hSetFileSize
Описание: обрезает заданный файл до заданного количества байт (по размеру файла). Попытка установить размер файла больше, чем имеется на текущий момент, не имеет эффекта.
Определение:

```
hSetFileSize :: Handle -> Integer -> IO ()
```

Функция: hIsEOF
Описание: возвращает значение IO True в случае, если в заданном открытом файле достигнут его конец.
Определение:

```
hIsEOF :: Handle -> IO Bool
```

Функция: isEOF
Описание: вариант функции hIsEOF для стандартного потока ввода stdin.
Определение:

```
isEOF :: IO Bool
isEOF = hIsEOF stdin
```

Тип: BufferMode
Описание: тип для представления режима буферизации файла. Для использования предлагается три способа буферизации: без неё, построчная буферизация и буферизация блоками заданного размера.
Определение:

```
data BufferMode
  = NoBuffering
  | LineBuffering
  | BlockBuffering (Maybe Int)
```

Данный тип является экземпляром следующих классов: Eq, Ord, Read и Show.

Функция: hSetBuffering
Описание: устанавливает для заданного файла заданный режим буферизации. Данная функция может сгенерировать исключение в случае, если архитектура не позволяет менять режим буферизации на открытых файлах (isPermissionError).

Определение:

```
hSetBuffering :: Handle -> BufferMode -> IO ()
```

Функция: hGetBuffering
Описание: возвращает текущий режим буферизации для заданного файла.
Определение:

```
hGetBuffering :: Handle -> IO BufferMode
```

Функция: hFlush
Описание: очищает буфер, записывая все данные из него в заданный файл. Эта функция может сгенерировать исключения в следующих случаях: устройство вывода заполнено (isFullError); достигнут предел ресурсов системы (isPermissionError).
Определение:

```
hFlush :: Handle -> IO ()
```

Тип: HandlePosn
Описание: представляет местоположение маркера в некотором файле.
Определение:

```
data HandlePosn = HandlePosn Handle Int deriving Eq
```

Функция: hGetPosn
Описание: возвращает положение маркера в заданном файле.
Определение:

```
hGetPosn :: Handle -> IO HandlePosn
```

Функция: hSetPosn
Описание: устанавливает маркер на заданную позицию в файле (файл определяется непосредственно в описании позиции). Функция может сгенерировать исключение в случае превышения ресурсов системы (isPermissionError).
Определение:

```
hSetPosn :: HandlePosn -> IO ()
```

Тип: SeekMode
Описание: тип для представления режима поиска маркера в файле. Используется в функции hSeek. Первый конструктор устанавливает смещение от начала файла. Второй — от текущей позиции в файле. Третий — от конца файла в обратную сторону.
Определение:

```
data SeekMode
  = AbsoluteSeek
  | RelativeSeek
  | SeekFromEnd
```

Для этого типа определены экземпляры следующих классов: Enum, Eq, Ix, Ord, Read и Show.
Функция: hSeek
Описание: устанавливает маркер в заданном файле на заданную позицию в зависимости от режима поиска. Функция может сгенерировать исключение в случае превышения ресурсов системы (isPermissionError).
Определение:

```
hSeek :: Handle -> SeekMode -> Integer -> IO ()
```

Функция: hIsOpen
Описание: возвращает значение IO True в случае, если заданный файл открыт.
Определение:

```
hIsOpen :: Handle -> IO Bool
```

Функция: hIsClosed
Описание: возвращает значение IO True в случае, если заданный файл закрыт.
Определение:

```
hIsClosed :: Handle -> IO Bool
```

Функция: hIsReadable
Описание: возвращает значение IO True в случае, если заданный файл может быть открыт для чтения.

Определение:

```
hIsReadable :: Handle -> IO Bool
```

Функция: hIsWritable
Описание: возвращает значение IO True в случае, если заданный файл может быть открыт для записи.
Определение:

```
hIsWritable :: Handle -> IO Bool
```

Функция: hIsSeekable
Описание: возвращает значение IO True в случае, если в заданном файле можно манипулировать маркером.
Определение:

```
hIsSeekable :: Handle -> IO Bool
```

Функция: hIsTerminalDevice
Описание: возвращает значение IO True в случае, если заданный дескриптор указывает на открытый терминал.
Определение:

```
hIsTerminalDevice :: Handle -> IO Bool
```

Функция: hSetEcho
Описание: устанавливает режим отображения символов в заданном терминале.
Определение:

```
hSetEcho :: Handle -> Bool -> IO ()
```

Функция: hGetEcho
Описание: возвращает режим отображения символов в заданном терминале (значение IO True возвращается в случае, если символы выводятся на экран.
Определение:

```
hGetEcho :: Handle -> IO Bool
```

Функция: hShow
Описание: возвращает в виде строки полный статус заданного дескриптора.

Определение:

```
hShow :: Handle -> IO String
```

Функция: `hWaitForInput`

Описание: возвращает значение `IO True` в случае, если заданный файл доступен для чтения. Иначе ожидает заданное количество миллисекунд, после чего возвращает значение `IO False`. Если заданное количество миллисекунд меньше нуля, функция будет ожидать бесконечно. Функция может сгенерировать исключение в случае достижения конца файла (`isEOFError`).

Определение:

```
hWaitForInput :: Handle -> Int -> IO Bool
```

Функция: `hReady`

Описание: возвращает значение `IO True` в случае, если из заданного файла доступно для чтения хотя бы одно значение. Функция может сгенерировать исключение в случае достижения конца файла (`isEOFError`).

Определение:

```
hReady :: Handle -> IO Bool
hReady h = hWaitForInput h 0
```

Функция: `hGetChar`

Описание: возвращает один символ из заданного файла, блокируя его до тех пор, пока символ не становится доступным. Функция может сгенерировать исключение в случае достижения конца файла (`isEOFError`).

Определение:

```
hGetChar :: Handle -> IO Char
```

Функция: `hGetLine`

Описание: возвращает одну строку (до символа `EOL`) из заданного файла, блокируя его до тех пор, пока символ не становится доступным. Функция может сгенерировать исключение в случае достижения конца файла (`isEOFError`).

Определение:

```
hGetLine :: Handle -> IO String
```

Функция: hLookAhead
Описание: возвращает один символ из заданного потока, блокируя его до возможности считать символ. Сам символ не вынимается из потока. Функция может сгенерировать исключение в случае достижения конца файла (isEOFError).
Определение:

```
hLookAhead :: Handle -> IO Char
```

Функция: hGetContents
Описание: возвращает всё содержимое заданного файла или потока в виде одной строки, начиная с текущей позиции маркера. Функция может сгенерировать исключение в случае, если маркер стоит на конце файла (isEOFError).
Определение:

```
hGetContents :: Handle -> IO String
```

Функция: hPutChar
Описание: записывает заданный символ в заданный файл. Во внимание принимается режим буферизации файла, поэтому символ может быть записан не сразу. Функция может сгенерировать исключение в следующих случаях: устройство вывода переполнено (isFullError) или нет доступа к устройству вывода (isPermissionError).
Определение:

```
hPutChar :: Handle -> Char -> IO ()
```

Функция: hPutStr
Описание: записывает заданную строку в заданный файл. Во внимание принимается режим буферизации файла, поэтому строка может быть записана не сразу. Функция может сгенерировать исключение в следующих случаях: устройство вывода переполнено (isFullError) или нет доступа к устройству вывода (isPermissionError).
Определение:

```
hPutStr :: Handle -> String -> IO ()
```

Функция: hPutStrLn
Описание: вариант функции hPutStr, добавляющий в конце заданной строки символ перевода строки.

Определение:

```
hPutStrLn :: Handle -> String -> IO ()
hPutStrLn hndl str = do hPutStr  hndl str
                        hPutChar hndl '\n'
```

Функция: `hPrint`

Описание: вариант функции `hPutStr`, получающий на вход значение произвольного типа, имеющего экземпляр класса `Show` (то есть значения этого типа могут быть преобразованы в строку), после чего выводящий строку в заданный файл.
Определение:

```
hPrint :: Show a => Handle -> a -> IO ()
hPrint hdl = hPutStrLn hdl . show
```

Функция: `openBinaryFile`

Описание: открывает файл, заданный по имени, в бинарном режиме. Это позволяет избежать некоторых нежелательных системных преобразований символов перевода строки. Возвращает дескриптор файла.
Определение:

```
openBinaryFile :: FilePath -> IOMode -> IO Handle
```

Функция: `hSetBinaryMode`

Описание: для заданного файла устанавливает режим доступа на бинарный (второй аргумент — `True`) или текстовый (второй аргумент — `False`, соответственно).
Определение:

```
hSetBinaryMode :: Handle -> Bool -> IO ()
```

Функция: `hPutBuf`

Описание: записывает в заданный файл определённое количество данных, взятых по заданному указателю. Функция может сгенерировать исключение, если по указателю нет данных (`ResourceVanished`).
Определение:

```
hPutBuf :: Handle -> Ptr a -> Int -> IO ()
```

Функция: hGetBuf
Описание: записывает определённое количество данных из заданного файла по заданному указателем адресу. Эта функция никогда не генерирует исключений, связанных с достижением конца файла. В этом случае она просто записывает меньшее количество байт. Если требуемое количество байт равно нулю, то функция записывает всё содержимое файла по заданному адресу.
Определение:

```
hGetBuf :: Handle -> Ptr a -> Int -> IO Int
```

Функция: hPutBufNonBlocking
Описание: вариант функции hPutBuf, который не блокирует заданный файл во время записи.
Определение:

```
hPutBufNonBlocking :: Handle -> Ptr a -> Int -> IO Int
```

Функция: hGetBufNonBlocking
Описание: вариант функции hGetBuf, который не блокирует заданный файл во время чтения.
Определение:

```
hGetBufNonBlocking :: Handle -> Ptr a -> Int -> IO Int
```

Функция: openTempFile
Описание: открывает временный файл в заданном каталоге и с заданным именем. Возвращает пару, состоящую из полного пути к новому временному файлу и его дескриптора. Открытие файла происходит в режиме одновременного чтения и записи.
Определение:

```
openTempFile :: FilePath -> String -> IO (FilePath, Handle)
```

Функция: openBinaryTempFile
Описание: вариант функции openTempFile, открывающий временный файл в бинарном режиме.
Определение:

```
openBinaryTempFile :: FilePath -> String -> IO (FilePath, Handle)
```

11.7.1. Модуль Error

В модуле Error определено множество функций для работы с ошибками ввода/вывода. Также в нём определено несколько специализированных типов для представления таких ошибок. Использование модуля:

```
import System.IO.Error
```

Главный тип данных, вокруг которого всё вращается в этом модуле, — IOError. В этом модуле он определён в виде синонима типа для представления исключений, хотя в стандарте Haskell-98 он является непрозрачным примитивным типом.

Тип: IOError

Описание: представление ошибки ввода/вывода. Любое действие ввода/вывода может сгенерировать исключение этого типа вместо того, чтобы вернуть результат. Информация о типе IOException приведена на стр. 200.

Определение:

```
type IOError = IOException
```

Функция: userError

Описание: создаёт пользовательскую ошибку с заданным строковым описанием. Метод fail класса Monad (см. стр. 211) для монады IO всегда генерирует эту ошибку.

Определение:

```
userError :: String -> IOError
```

Функция определена в виде примитива.

Функция: mkIOError

Описание: создаёт ошибку заданного типа и прочими атрибутами ошибки.

Определение:

```
mkIOError :: IOErrorType -> String -> Maybe Handle -> Maybe FilePath -> IOError
mkIOError t location maybe_hdl maybe_filename = IOError
                                                {
                                                  ioe_type = t,
                                                  ioe_location = location,
                                                  ioe_description = "",
                                                  ioe_handle = maybe_hdl,
```

```
                                        ioe_filename = maybe_filename
                                    }
```

Функция: annotateIOError

Описание: добавляет к ошибке описание и, возможно, место, где она возникла, в виде полного пути к файлу и дескриптора файла.

Определение:

```
annotateIOError :: IOError -> String -> Maybe Handle > Maybe FilePath -> IOError
annotateIOError (IOError ohdl errTy _ str opath) loc hdl path
  = IOError (hdl `mplus` ohdl) errTy loc str (path `mplus` opath)
      where
        Nothing `mplus` ys = ys
        xs      `mplus` _  = xs
```

Функция: isAlreadyExistsError

Описание: возвращает значение IO True в случае, если заданная ошибка проявляется тогда, когда исключение сгенерировано при наличии одного из аргументов.

Определение:

```
isAlreadyExistsError :: IOError -> Bool
isAlreadyExistsError = isAlreadyExistsErrorType . ioeGetErrorType
```

Функция: isDoesNotExistError

Описание: возвращает значение IO True в случае, если заданная ошибка проявляется тогда, когда исключение сгенерировано при отсутствии одного из аргументов.

Определение:

```
isDoesNotExistError :: IOError -> Bool
isDoesNotExistError = isDoesNotExistErrorType . ioeGetErrorType
```

Функция: isAlreadyInUseError

Описание: возвращает значение IO True в случае, если заданная ошибка проявляется из-за того, что один из аргументов функции является однопользовательским ресурсом и он занят другим процессом в момент вызова.

Определение:

```
isAlreadyInUseError :: IOError -> Bool
isAlreadyInUseError = isAlreadyInUseErrorType . ioeGetErrorType
```

Функция: isFullError

Описание: возвращает значение True в случае, если заданная ошибка проявляется из-за того, что устройство, в которое производится запись, полностью заполнено.

Определение:

```
isFullError :: IOError -> Bool
isFullError = isFullErrorType . ioeGetErrorType
```

Функция: isEOFError

Описание: возвращает значение True в случае, если заданная ошибка проявляется из-за того, что процесс чтения подошёл к концу файла или потока информации.

Определение:

```
isEOFError :: IOError -> Bool
isEOFError = isEOFErrorType . ioeGetErrorType
```

Функция: isIllegalOperation

Описание: возвращает значение True в случае, если заданная ошибка проявляется из-за того, что операция ввода/вывода недопустима. Любая функция, возвращающая значение в монаде IO, может сгенерировать исключение этого типа.

Определение:

```
isIllegalOperation :: IOError -> Bool
isIllegalOperation = isIllegalOperationErrorType . ioeGetErrorType
```

Функция: isPermissionError

Описание: возвращает значение True в случае, если заданная ошибка проявляется из-за того, что для выполнения операции ввода/вывода отсутствуют права доступа.

Определение:

```
isPermissionError :: IOError -> Bool
isPermissionError = isPermissionErrorType . ioeGetErrorType
```

Функция: `isUserError`

Описание: возвращает значение `True` в случае, если заданная ошибка является ошибкой, которая определена программистом.

Определение:

```
isUserError :: IOError -> Bool
isUserError = isUserErrorType . ioeGetErrorType
```

Функция: `ioeGetErrorType`

Описание: возвращает тип заданной ошибки. Является синонимом для автоматически сгенерированного метода доступа к полю типа `IOError`.

Определение:

```
ioeGetErrorType :: IOError -> IOErrorType
ioeGetErrorType ioe = ioe_type ioe
```

Функция: `ioeGetErrorString`

Описание: возвращает строку-описание заданной ошибки. Является синонимом для автоматически сгенерированного метода доступа к полю строки-описания `IOError` в случае, если ошибка не является созданной пользователем.

Определение:

```
ioeGetErrorString :: IOError -> String
ioeGetErrorString ioe | isUserErrorType (ioe_type ioe) = ioe_description ioe
                      | otherwise                      = show (ioe_type ioe)
```

Функция: `ioeGetHandle`

Описание: возвращает дескриптор источника данных, при работе с которым произошла заданная ошибка. Является синонимом для автоматически сгенерированного метода доступа к полю дескриптора `IOError`.

Определение:

```
ioeGetHandle :: IOError -> Maybe Handle
ioeGetHandle ioe = ioe_handle ioe
```

Функция: `ioeGetFileName`

Описание: возвращает путь к файлу, при работе с которым произошла заданная ошибка. Является синонимом для автоматически сгенерированного метода доступа к полю пути к файлу `IOError`.

Определение:

```
ioeGetFileName :: IOError -> Maybe FilePath
ioeGetFileName ioe = ioe_filename ioe
```

Функция: ioeSetErrorType

Описание: устанавливает тип заданной ошибки. Является синонимом для автоматически сгенерированного метода установки к полю типа IOError.

Определение:

```
ioeSetErrorType :: IOError -> IOErrorType -> IOError
ioeSetErrorType ioe errtype = ioe { ioe_type = errtype }
```

Функция: ioeSetErrorString

Описание: устанавливает строку-описание заданной ошибки. Является синонимом для автоматически сгенерированного метода установки к полю строки-описания IOError.

Определение:

```
ioeSetErrorString :: IOError -> String -> IOError
ioeSetErrorString ioe str = ioe { ioe_description = str }
```

Функция: ioeSetHandle

Описание: устанавливает дескриптор источника данных, при работе с которым произошла заданная ошибка. Является синонимом для автоматически сгенерированного метода уставноки к полю дескриптора IOError.

Определение:

```
ioeSetHandle :: IOError -> Handle -> IOError
ioeSetHandle ioe hdl = ioe { ioe_handle = Just hdl }
```

Функция: ioeSetFileName

Описание: устанавливает путь к файлу, при работе с которым произошла заданная ошибка. Является синонимом для автоматически сгенерированного метода установки к полю пути к файлу IOError.

Определение:

```
ioeSetFileName :: IOError -> FilePath -> IOError
ioeSetFileName ioe filename = ioe { ioe_filename = Just filename }
```

Тип: `IOErrorType`
Описание: абстрактный примитивный тип-перечисление, представляющий типы ошибок ввода/вывода.
Определение:

```
data IOErrorType = ...
```

Тип определён в виде примитива.

Функция: `alreadyExistsErrorType`
Описание: константная функция, возвращающая тип ошибки, которая происходит в случае, если какой-либо аргумент действия ввода/вывода уже существует.
Определение:

```
alreadyExistsErrorType :: IOErrorType
alreadyExistsErrorType = AlreadyExists
```

Функция: `doesNotExistErrorType`
Описание: константная функция, возвращающая тип ошибки, которая происходит в случае, если какой-либо аргумент действия ввода/вывода не существует.
Определение:

```
doesNotExistErrorType :: IOErrorType
doesNotExistErrorType = NoSuchThing
```

Функция: `alreadyInUseErrorType`
Описание: константная функция, возвращающая тип ошибки, которая происходит в случае, если какой-либо аргумент действия ввода/вывода является однопользовательским и на момент доступа к нему заблокирован другим процессом.
Определение:

```
alreadyInUseErrorType :: IOErrorType
alreadyInUseErrorType = ResourceBusy
```

Функция: `fullErrorType`
Описание: константная функция, возвращающая тип ошибки, которая происходит в случае, если устройство, в которое пытается записать действие ввода/вывода, уже заполнено полностью.
Определение:

```
fullErrorType :: IOErrorType
```

```
fullErrorType = ResourceExhausted
```

Функция: `eofErrorType`

Описание: константная функция, возвращающая тип ошибки, которая происходит в случае, если во входном файле или потоке достигнут конец.

Определение:

```
eofErrorType :: IOErrorType
eofErrorType = EOF
```

Функция: `illegalOperationErrorType`

Описание: константная функция, возвращающая тип ошибки, которая происходит в случае, если совершаемое действие ввода/вывода недопустимо.

Определение:

```
illegalOperationErrorType :: IOErrorType
illegalOperationErrorType = IllegalOperation
```

Функция: `permissionErrorType`

Описание: константная функция, возвращающая тип ошибки, которая происходит в случае, если для совершения действие ввода/вывода нет прав доступа.

Определение:

```
permissionErrorType :: IOErrorType
permissionErrorType = PermissionDenied
```

Функция: `userErrorType`

Описание: константная функция, возвращающая тип ошибки, которая определена программистом.

Определение:

```
userErrorType :: IOErrorType
userErrorType = UserError
```

Функция: `isAlreadyExistsErrorType`

Описание: предикат, возвращающий значение `True` тогда, когда заданный тип ошибки является типом ошибки, произошедшей в случае, если какой-либо аргумент действия ввода/вывода уже существует.

Определение:

```
isAlreadyExistsErrorType :: IOErrorType -> Bool
```

```
isAlreadyExistsErrorType AlreadyExists = True
isAlreadyExistsErrorType _             = False
```

Функция: isDoesNotExistErrorType

Описание: предикат, возвращающий значение True тогда, когда заданный тип ошибки является типом ошибки, произошедшей в случае, если какой-либо аргумент действия ввода/вывода не существует.

Определение:

```
isDoesNotExistErrorType :: IOErrorType -> Bool
isDoesNotExistErrorType NoSuchThing = True
isDoesNotExistErrorType _           = False
```

Функция: isAlreadyInUseErrorType

Описание: предикат, возвращающий значение True тогда, когда заданный тип ошибки является типом ошибки, произошедшей в случае, если какой-либо аргумент действия ввода/вывода является однопользовательским и на момент доступа к нему заблокирован другим процессом.

Определение:

```
isAlreadyInUseErrorType :: IOErrorType -> Bool
isAlreadyInUseErrorType ResourceBusy = True
isAlreadyInUseErrorType _            = False
```

Функция: isFullErrorType

Описание: предикат, возвращающий значение True тогда, когда заданный тип ошибки является типом ошибки, произошедшей в случае, если устройство, в которое пытаются записать действие ввода/вывода, уже заполнено полностью.

Определение:

```
isFullErrorType :: IOErrorType -> Bool
isFullErrorType ResourceExhausted = True
isFullErrorType _                 = False
```

Функция: isEOFErrorType

Описание: предикат, возвращающий значение True тогда, когда заданный тип ошибки является типом ошибки, произошедшей в случае, если во входном файле или потоке достигнут конец.

Определение:

```
isEOFErrorType :: IOErrorType -> Bool
isEOFErrorType EOF = True
isEOFErrorType _   = False
```

Функция: isIllegalOperationErrorType

Описание: предикат, возвращающий значение True тогда, когда заданный тип ошибки является типом ошибки, произошедшей в случае, если совершаемое действие ввода/вывода недопустимо.

Определение:

```
isIllegalOperationErrorType :: IOErrorType -> Bool
isIllegalOperationErrorType IllegalOperation = True
isIllegalOperationErrorType _                = False
```

Функция: isPermissionErrorType

Описание: предикат, возвращающий значение True тогда, когда заданный тип ошибки является типом ошибки, произошедшей в случае, если для совершения действие ввода/вывода нет прав доступа.

Определение:

```
isPermissionErrorType :: IOErrorType -> Bool
isPermissionErrorType PermissionDenied = True
isPermissionErrorType _                = False
```

Функция: isUserErrorType

Описание: предикат, возвращающий значение True тогда, когда заданный тип ошибки является типом ошибки, которая определена программистом.

Определение:

```
isUserErrorType :: IOErrorType -> Bool
isUserErrorType UserError = True
isUserErrorType _         = False
```

Функция: ioError

Описание: генерирует ошибку ввода/вывода заданного типа.

Определение:

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

Функция определена в виде примитива.

Функция: `catch`

Описание: устанавливает функцию, которая обрабатывает сгенерированную ошибку ввода/вывода, если она произошла во время выполнения заданного действия. Первым аргументом функции является монадическое действие ввода/вывода, которое непосредственно выполняется. Если во время его выполнения произошла ошибка, то для её обработки используется функция, задаваемая вторым аргументом. Сама функция `catch` возвращает либо результат самого действия, либо результат обработки ошибки.

Определение:

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

Функция определена в виде примитива.

Функция: `try`

Описание: вариант функции `catch`, который обрабатывает ошибки ввода/вывода при помощи идиомы типа `Either` (см. подробности в разделе 8.8.). Эта функция не обрабатывает исключения, которые возникают не в процессе ввода/вывода. Для их обработки необходимо пользоваться одноимённой функцией из модуля `Exception` (см. раздел 7.4.).

Определение:

```
try :: IO a -> IO (Either IOError a)
try f = catch (do r <- f
                  return (Right r))
              (return . Left)
```

Функция: `modifyIOError`

Описание: отлавливает ошибку ввода/вывода и генерирует её изменённую версию. Изменение производится при помощи заданной первым аргументом функции.

Определение:

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

Функция определена в виде примитива.

11.7.2. Модуль Unsafe

Модуль Unsafe предоставляет программисту две функции, которые позволяют «вынуть» из монады IO значения. Использование:

```
import System.IO.Unsafe
```

Функция: unsafePerformIO
Описание: возвращает значение из монады IO. Это единственная функция, которая позволяет получить немонадическое значение из этой монады. Само собой разумеется, что функция не является безопасной, поскольку в случае недетерминированности и наличия сторонних эффектов результат выполнения функции также будет недетерминированным.
Определение:

```
unsafePerformIO :: IO a -> a
```

Функция определена в виде примитива.

Если вычисления, поданные на вход этой функции, содержат сторонние эффекты, то относительный порядок проведения операций со сторонними эффектами (конечно, в рамках монады IO) не определён. При использовании этой функции необходимо быть очень осторожным.
Функция: unsafeInterleaveIO
Описание: позволяет отложить (в смысле ленивости) действия ввода/вывода в монаде IO. Заданное действие выполняется только тогда (равно как и его сторонние эффекты), когда значение a, обёрнутое в монаду, требуется для вычислений. Эта функция используется, к примеру, для реализации ленивого чтения файлов.
Определение:

```
unsafeInterleaveIO :: IO a -> IO a
```

Функция определена в виде примитива.

11.8. Модуль Locale

Модуль Locale содержит определения программных сущностей для работы с локализацией. Возможности по адаптации локальных правил на текущее время

поддерживаются только для форматов времени и даты. Этот модуль используется в модуле Time (см. раздел 11.11.) для правильного преобразования времени в строку. Использование:

```
import System.Locale
```

В модуле определён один алгебраический тип данных для представления времени, а также три дополнительных функции к нему.

Тип: `TimeLocale`

Описание: тип для представления локального (местного) формата времени. Единственный одноимённый конструктор параметризует набор полей, которые используются для наименования различных понятий в области времени (дни недели, месяцы и т. д.)

Определение:

```
data TimeLocale
  = TimeLocale
    {
      wDays     :: [(String, String)]
      months    :: [(String, String)]
      intervals :: [(String, String)]
      amPm      :: (String, String)
      dateTimeFmt, dateFmt, timeFmt, time12Fmt :: String
    }
```

Для данного типа определены экземпляры классов Eq, Ord и Show.

Функция: `defaultTimeLocale`

Описание: константа, определяющая текущие настройки времени (англоязычные).

Определение:

```
defaultTimeLocale :: TimeLocale
defaultTimeLocale
  = TimeLocale
    {
      wDays  = [("Sunday",   "Sun"),  ("Monday",    "Mon"),
                ("Tuesday",  "Tue"),  ("Wednesday", "Wed"),
                ("Thursday", "Thu"),  ("Friday",    "Fri"),
                ("Saturday", "Sat")],
```

```
    months = [("January",   "Jan"), ("February",  "Feb"),
              ("March",     "Mar"), ("April",     "Apr"),
              ("May",       "May"), ("June",      "Jun"),
              ("July",      "Jul"), ("August",    "Aug"),
              ("September", "Sep"), ("October",   "Oct"),
              ("November",  "Nov"), ("December",  "Dec")],


    intervals = [ ("year",  "years"),
                  ("month", "months"),
                  ("day",   "days"),
                  ("hour",  "hours"),
                  ("min",   "mins"),
                  ("sec",   "secs"),
                  ("usec",  "usecs")],


      amPm = ("AM", "PM"),
      dateTimeFmt = "%a %b %e %H:%M:%S %Z %Y",
      dateFmt = "%m/%d/%y",
      timeFmt = "%H:%M:%S",
      time12Fmt = "%I:%M:%S %p"
  }
```

Функция: `iso8601DateFormat`

Описание: формат времени по стандарту ISO 8601 (YYYY-MM-DD). В выходную строку также можно добавить спецификацию времени (HH:MM:SS).

Определение:

```
iso8601DateFormat :: Maybe String -> String
iso8601DateFormat timeFmt = "%Y-%m-%d" ++ case timeFmt of
                                            Nothing  -> ""
                                            Just fmt -> ' ' : fmt
```

Функция: `rfc822DateFormat`

Описание: формат времени по стандарту RFC 822.

Определение:

```
rfc822DateFormat :: String
rfc822DateFormat = "%a, %_d %b %Y %H:%M:%S %Z"
```

11.9. Модуль Mem

В модуле Mem определяется единственная функция, которая используется для воздействия на систему управления памяти в языке Haskell. Использование модуля:

```
import System.Mem
```

Функция: performGC
Описание: немедленно при вызове запускает процесс сборки мусора.
Определение:

```
performGC :: IO ()
```

Функция определена в виде примитива.

11.9.1. Модуль StableName

Модуль StableName предлагает к использованию так называемые стабильные имена, то есть идентификаторы объектов в памяти, которые не меняются при изменении местоположения объекта. Такие стабильные имена помогают производить не совсем точное сравнение объектов (достаточно быстрое, сложности $O(1)$), равно как и позволяют осуществлять их хеширование. Использование модуля:

```
import System.Mem.StableName
```

Стабильные имена решают следующую задачу. Если имеется необходимость построить хеш-таблицу, ключами в которой выступают некоторые значения из языка Haskell, то сравнение ключей должно производиться на основе некоторых указателей, поскольку сами ключи, возможно, являются громоздкими и неудобными для сравнения (например, они бесконечны). Невозможно построить хеш-таблицу на основе указателей на адреса памяти, по которым расположены ключи, поскольку объекты могут перемещаться в памяти в процессе сборки мусора, что предполагает тот факт, что при каждом запуске сборщика мусора необходимо производить перерасчёт всей хеш-таблицы.

Решить эту проблему позволяет тип StableName.

Тип: `StableName`

Описание: абстрактный идентификатор объекта заданного типа `a`, который поддерживает сравнение и хеширование.

Определение:

```
data StableName a = ...
```

Тип определён в виде примитива.

Стабильные имена имеют важное свойство. Если два стабильных имени `sn1` и `sn2` равны (предикат (==) возвращает значение `True` для них), то верно то, что эти два имени созданы при помощи функции `makeStableName` на одном и том же объекте. Обратное, тем не менее, неверно. Если два стабильных имени не равны, то это не значит, что они созданы на разных объектах. Родительский объект может быть одним и тем же.

Стабильные имена похожи на стабильные указатели `StablePtr` (см. раздел 10.5.), однако имеются важные отличия. Для стабильных имён нет возможности освободить от них память, поскольку они выделяются и высвобождаются обычной системой управления памятью языка Haskell. Кроме того, нет возможности осуществить обратное преобразование от стабильного имени к объекту, который его породил. Причина этого заключается в том, что нет никакой гарантии в том, что при существовании стабильного имени всё ещё существует соответствующий объект.

Для типа `StableName` определены экземпляры классов `Typeable1` и `Eq`.

Функция: `makeStableName`

Описание: создаёт стабильное имя для произвольного объекта. Сам объект, передаваемый в качестве аргумента, не используется в вычислениях.

Определение:

```
makeStableName :: a -> IO (StableName a)
```

Функция определена в виде примитива.

Функция: `hashStableName`

Описание: преобразует стабильное имя в целочисленное значение. Возвращаемое значение не является уникальным, некоторые различные стабильные имена могут быть преобразованы в одно и то же целочисленное значение (на практике, однако, шансы этого малы, поэтому эта функция может использоваться для генерации ключей в хеш-таблицах).

Определение:

```
hashStableName :: StableName a -> Int
```

Функция определена в виде примитива.

11.9.2. Модуль Weak

Модуль Weak предоставляет программисту использовать так называемые слабые указатели, которые характеризуются тем, что они не обслуживаются сборщиком мусора. Это значит, что существование слабого указателя на некоторый объект никак не влияет на жизненный цикл этого объекта. Однако сам по себе такой слабый указатель может быть использован для определения факта наличия объекта в памяти (или его отсутствия), и в случае наличия — получения значения объекта.

Слабые указатели чрезвычайно полезны в деле кеширования и создания таблиц значений функций. Таблица значений функции является структурой данных, которая отображает ключ (аргумент функции) в некоторое значение (результат функции на конкретном аргументе). При применении функции к аргументу сначала проверяется, не существует ли уже заданный аргумент в таблице, равно как и вычисленное для него значение. Тонкий момент здесь заключается в том, что такая таблица не должна держать в целости и сохранности ключи. Поэтому для её построения используются слабые указатели. Однако, с другой стороны, указатели на значения должны быть обычными (сильными), поскольку такая таблица является зачастую единственным местом, где значения хранятся.

Это выглядит таким образом, что подобная таблица значений функции будет всегда хранить значения функций. Единственным способом решить эту проблему является периодическая чистка таблицы, удаляя значения тех ключей, которые больше не используются. Но слабые указатели дают другой способ решения проблемы, называемый *финализацией*. Когда уничтожается ключ, система управления памятью запускает специальную функцию, которая убирает из таблицы заданное значение (это — применение финализаторов конкретно в примере таблиц значений функций).

Использование модуля:

```
import System.Mem.Weak
```

В рассматриваемом модуле содержатся определения одного алгебраического типа данных для представления слабых указателей, а также набор функций для работы с ним.

Тип: `Weak`

Описание: слабый указатель на значение типа `a`. Слабый указатель является некоторым отношением между ключом (собственно, указателем) и значением. Если сборщик мусора определяет, что ключ должен быть оставлен, то и соответствующее значение также оставляется. Однако обратное неверно: наличие ссылки не гарантирует того, что ключ будет оставлен в процессе сборки мусора.

Определение:

```
data Weak a = ...
```

Тип определён в виде примитива.

Слабые указатели могут иметь финализаторы. Если финализатор существует, то есть гарантия, что он будет запущен по крайней мере один раз тогда, когда ключ станет недоступным в программе. Система управления памятью будет стараться запустить финализатор сразу же, как только ключ умрёт, однако сама по себе оперативность запуска не гарантируется. Также не гарантируется то, что финализатор будет вообще запущен, поскольку финализаторы выполняются при выходе из программы. Поэтому финализаторы не должны использоваться в целях высвобождения памяти, для этих целей необходимо пользоваться другими средствами, в частности генерацией и отловом исключений.

Для определённого ключа могут существовать несколько слабых указателей. В этом случае при удалении ключа все указатели будут запущены в произвольном порядке (либо параллельно, если позволяет архитектура). Если программист реализовал финализатор, в котором используется предположение о существовании всего одной ссылки на объект, то в этом случае программист должен сам заботиться об отсутствии других финализаторов.

Для данного типа определён экземпляр класса `Typeable1`.

Функция: `mkWeak`

Описание: создаёт слабый указатель на первый аргумент со значением второго аргумента и с третьим аргументом в качестве финализатора (может отсутствовать). Эта функция является наиболее общим способом создания слабых указателей.

Определение:

```
mkWeak :: k -> v -> Maybe (IO ()) -> IO (Weak v)
```

Функция определена в виде примитива.

Функция: mkWeakPtr

Описание: специализированная версия функции mkWeak, в которой ключ и значение являются одним и тем же объектом.

Определение:

```
mkWeakPtr :: k -> Maybe (IO ()) -> IO (Weak k)
mkWeakPtr key finalizer = mkWeak key key finalizer
```

Функция: mkWeakPair

Описание: вариант функции mkWeak, который в качестве значения слабого указателя создаёт пару вида (*ключ*, *значение*).

Определение:

```
mkWeakPair :: k -> v -> Maybe (IO ()) -> IO (Weak (k, v))
mkWeakPair key val finalizer = mkWeak key (key, val) finalizer
```

Функция: deRefWeak

Описание: возвращает значение из слабого указателя. Если значение ещё существует, оно возвращается в конструкторе Just. Если значения уже нет, возвращается Nothing. Значение обёрнуто в монаду IO, поскольку оно зависит от процесса сборки мусора, а потому недетерминировано.

Определение:

```
deRefWeak :: Weak v -> IO (Maybe v)
```

Функция определена в виде примитива.

Функция: finalize

Описание: запускает финализатор, ассоциированный со слабым указателем.

Определение:

```
finalize :: Weak v -> IO ()
```

Функция определена в виде примитива.

Функция: addFinalizer

Описание: добавляет финализатор к заданному слабому указателю.

Определение:

```
addFinalizer :: key -> IO () -> IO ()
addFinalizer key finalizer = do mkWeakPtr key (Just finalizer)
                                return ()
```

11.10. Модуль Random

Модуль `Random` содержит определения программных сущностей для работы со случайными числами. Использование:

```
import System.Random
```

В модуле описано два класса для представления генераторов случайных величин и для представления самих случайных величин. Также описан специальный тип данных, представляющий собой обобщённый генератор случайных величин. Ну и наконец, определены пять утилитарных функций для работы со случайными значениями.

Рассматриваемый модуль предлагает решение общей проблемы по генерации псевдослучайных чисел. Функции в этом модуле позволяют генерировать последовательности повторяющихся значений при помощи начального элемента (генератора). Либо можно использовать системный генератор для получения неповторяющихся значений при каждом запуске.

Библиотека разбита на два слоя. На первом находится ядро генератора случайных чисел, которое работает на битовом уровне. Класс `RandomGen` предоставляет базовый интерфейс к таким генераторам. Второй слой — класс `Random`, который является интерфейсом для получения конкретных значений из генератора случайных чисел. Например, экземпляр этого класса для типа `Float` будет генерировать случайные числа типа `Float`.

Реализация системного генератора использует технологию, описанную в [14] для 32-битных архитектур. Эта технология позволяет генерировать псевдослучайные последовательности с периодом примерно 2.30584^{18}. Дополнительные данные по генераторам случайных чисел в рамках функциональной парадигмы можно найти в работах [3, 4, 7, 20].

Класс: `RandomGen`

Описание: представляет обобщённый интерфейс для генераторов случайных значений.

Определение:

```
class RandomGen g where
  next     :: g -> (Int, g)
  split    :: g -> (g, g)
  genRange :: g -> (Int, Int)
```

Метод `next` возвращает очередное целочисленное случайное значение, которое равномерно распределено в интервале, который возвращается методом `genRange` (включая конечные точки), а также новое значение генератора. Соответственно, метод `genRange` возвращает пару значений, являющихся нижней и верхней границей интервала, внутри которого происходит выборка случайных чисел. Этот метод не является строгим, что означает, что метод не вычисляет своего аргумента (не использует его), а потому единственный вызов метода может быть использован для получения интервала без опасения того, что для нового генератора, полученного при помощи метода `next`, интервал будет новый.

Метод `split` позволяет получить два различных генератора случайных величин на основе одного входного. Этот метод весьма полезен в функциональных алгоритмах, поскольку позволяет передавать генераторы случайных величин в рекурсивные деревья вызова.

Единственным экземпляром этого класса является тип `StdGen`.

Тип: `StdGen`

Описание: абстрактный тип, представляющий обобщённый генератор случайных чисел. Интервал, используемый для генерации случайных чисел, входит в размер 30 бит.

Определение:

```
data StdGen = StdGen Int Int
```

Данный тип имеет экземпляры следующих классов: `RandomGen`, `Read` и `Show`.

Функция: `mkStdGen`

Описание: альтернативный способ создания генератора случайных чисел при помощи проецирования целочисленного значения на тип генератора. Опять же, различные входные значения, скорее всего, произведут различные выходные.

Определение:

```
mkStdGen :: Int -> StdGen
mkStdGen s | s < 0     = mkStdGen (-s)
           | otherwise = StdGen (s1 + 1) (s2 + 1)
  where
    (q, s1) = s `divMod` 2147483562
    s2      = q `mod` 2147483398
```

Класс: Random

Описание: позволяет получать случайные значения различных типов. Обязательному определению подлежат методы random и randomR.

Определение:

```
class Random a where
  randomR   :: RandomGen g => (a, a) -> g -> (a, g)
  random    :: RandomGen g => g -> (a, g)
  randomRs  :: RandomGen g => (a, a) -> g -> [a]
  randoms   :: RandomGen g => g -> [a]
  randomRIO :: (a, a) -> IO a
  randomIO  :: IO a
```

Метод randomR получает на вход пару, представляющую нижнюю и верхнюю границы интервала, а также генератор случайных величин. Возвращает случайное значение, равномерно распределённое в заданном интервале, а также новое значение генератора. В свою очередь метод random выполняет те же самые действия, но в качестве интервала выборки берёт интервал, заданный для типа генератора по умолчанию методом genRange класса RandomGen.

Метод randomRs является вариантом метода randomR, возвращающим бесконечный список случайных значений. Также и метод randoms является таким же вариантом метода random. Наконец, методы randomRIO и randomIO для генерации случайных чисел используют глобальный генератор, а значения возвращают в монаде IO.

Экземплярами класса являются типы: Bool, Char, Double, Float, Int и Integer.

Функция: getStdRandom

Описание: использует заданную функцию для получения случайного значения из текущего глобального генератора, а также обновляет глобальный генератор

новым, который возвращён функцией. Поскольку эта функция использует побочные эффекты, она определена в монаде IO.
Определение:

```
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
getStdRandom f = do rng <- getStdGen
                    let (v, new_rng) = f rng
                    setStdGen new_rng
                    return v
```

Использование этой функции достаточно просто. Например, для получения случайного числа в интервале от 1 до 6 (для эмуляции игры в кости) можно воспользоваться следующей функцией:

```
rollDice :: IO Int
rollDice = getStdRandom (randomR (1, 6))
```

Функция: getStdGen
Описание: возвращает глобальный генератор случайных чисел.
Определение:

```
getStdGen :: IO StdGen
getStdGen  = readIORef theStdGen
```

Функция: setStdGen
Описание: устанавливает глобальный генератор случайных чисел.
Определение:

```
setStdGen :: StdGen -> IO ()
setStdGen sgen = writeIORef theStdGen sgen
```

Функция: newStdGen
Описание: применяет метод split к глобальному генератору случайных чисел, меняет его на первый результат функции, возвращает второй результат функции.
Определение:

```
newStdGen :: IO StdGen
newStdGen = do rng <- getStdGen
               let (a, b) = split rng
               setStdGen a
               return b
```

11.11. Модуль Time

Модуль Time предоставляет программисту инструменты для работы с системным временем, включая функциональность для обработки собственно времени, информации о временно́й зоне и т. д. Функциональность модуля следует стандарту RFC 1129 в части обработки времени UTC. Использование модуля:

```
import System.Time
```

В модуле содержатся определения нескольких специализированных алгебраических типов данных, а также функций для их обработки.

Тип: ClockTime

Описание: представление внутреннего формата времени. Значения этого типа могут быть сравниваемы, преобразованы в строки или преобразованы во внешнее представление в типе ClendarTime для дальнейшего использования.

Определение:

```
data ClockTime = TOD Integer Integer
```

Для данного типа определены экземпляры следующих классов: Eq, Ord и Show.

Функция: getClockTime

Описание: возвращает текущее системное время во внутреннем формате.

Определение:

```
getClockTime :: IO ClockTime
getClockTime = do (sec, usec) <- getClockTimePrim
                  return (TOD (fromIntegral sec) ((fromIntegral usec) * 1000000))
```

Тип: TimeDiff

Описание: представляет разницу между двумя отметками времени.

Определение:

```
data TimeDiff
  = TimeDiff
    {
      tdYear    :: Int
      tdMonth   :: Int
      tdDay     :: Int
      tdHour    :: Int
      tdMin     :: Int
      tdSec     :: Int
```

```
    tdPicosec :: Integer
  }
```

Для данного типа определены экземпляры следующих классов: Eq, Ord, Read и Show.

Функция: noTimeDiff

Описание: константная функция, возвращающая нулевую разницу между двумя отметками времени.

Определение:

```
noTimeDiff :: TimeDiff
noTimeDiff = TimeDiff 0 0 0 0 0 0 0
```

Функция: diffClockTimes

Описание: возвращает разницу между двумя заданными отметками времени.

Определение:

```
diffClockTimes :: ClockTime -> ClockTime -> TimeDiff
diffClockTimes (TOD sa pa) (TOD sb pb) = noTimeDiff { tdSec = fromIntegral (sa - sb),
                                                      tdPicosec = pa - pb }
```

Функция: addToClockTime

Описание: возвращает новую отметку времени, полученную при помощи добавления к заданной отметке времени некоторого временно́го интервала.

Определение:

```
addToClockTime :: TimeDiff -> ClockTime -> ClockTime
addToClockTime (TimeDiff year mon day hour min sec psec)
               (TOD c_sec c_psec)
  = let sec_diff = toInteger sec +
                   60 * toInteger min +
                   3600 * toInteger hour +
                   24 * 3600 * toInteger day
        (d_sec, d_psec) = (c_psec + psec) `quotRem` 1000000000000
        cal = toUTCTime (TOD (c_sec + sec_diff + d_sec) d_psec)
        new_mon  = fromEnum (ctMonth cal) + r_mon
        month' = fst tmp
        yr_diff = snd tmp
        tmp | new_mon < 0  = (toEnum (12 + new_mon), (-1))
            | new_mon > 11 = (toEnum (new_mon `mod` 12), 1)
            | otherwise    = (toEnum new_mon, 0)
        (r_yr, r_mon) = mon `quotRem` 12
```

```
        year' = ctYear cal + year + r_yr + yr_diff
    in  toClockTime cal {ctMonth = month', ctYear = year'}
```

Функция: `normalizeTimeDiff`

Описание: преобразует разницу между двумя отметками времени в нормальную форму.

Определение:

```
normalizeTimeDiff :: TimeDiff -> TimeDiff
normalizeTimeDiff td = let rest0 = toInteger (tdSec td) +
                                   60 * (toInteger (tdMin td) +
                                   60 * (toInteger (tdHour td) +
                                   24 * (toInteger (tdDay td) +
                                   30 * toInteger (tdMonth td) +
                                   365 * toInteger (tdYear td))))
                           (diffYears,  rest1)    = rest0 `quotRem` (365 * 24 * 3600)
                           (diffMonths, rest2)    = rest1 `quotRem` (30 * 24 * 3600)
                           (diffDays,   rest3)    = rest2 `quotRem` (24 * 3600)
                           (diffHours,  rest4)    = rest3 `quotRem` 3600
                           (diffMins,   diffSecs) = rest4 `quotRem` 60
                       in  td { tdYear  = fromInteger diffYears,
                                tdMonth = fromInteger diffMonths,
                                tdDay   = fromInteger diffDays,
                                tdHour  = fromInteger diffHours,
                                tdMin   = fromInteger diffMins,
                                tdSec   = fromInteger diffSecs }
```

Функция: `timeDiffToString`

Описание: преобразует разницу между двумя отметками времени в строку, используя локальные соглашения о представлении времени.

Определение:

```
timeDiffToString :: TimeDiff -> String
timeDiffToString = formatTimeDiff defaultTimeLocale "%c"
```

Функция: `formatTimeDiff`

Описание: преобразует разницу между двумя отметками времени в строку, используя локальные соглашения о представлении времени и форматную строку. Результат понимаем функцией `strftime` языка C.

Определение:

```
formatTimeDiff :: TimeLocale -> String -> TimeDiff -> String
formatTimeDiff l fmt td@(TimeDiff year month day hour min sec _) = doFmt fmt
  where
    doFmt ""           = ""
    doFmt ('%':'-':cs) = doFmt ('%':cs)
    doFmt ('%':'_':cs) = doFmt ('%':cs)
    doFmt ('%':c:cs)   = decode c ++ doFmt cs
    doFmt (c:cs)       = c : doFmt cs

    decode spec = case spec of
                    'B' -> fst (months l !! fromEnum month)
                    'b' -> snd (months l !! fromEnum month)
                    'h' -> snd (months l !! fromEnum month)
                    'c' -> defaultTimeDiffFmt td
                    'C' -> show2 (year `quot` 100)
                    'D' -> doFmt "%m/%d/%y"
                    'd' -> show2 day
                    'e' -> show2' day
                    'H' -> show2 hour
                    'I' -> show2 (to12 hour)
                    'k' -> show2' hour
                    'l' -> show2' (to12 hour)
                    'M' -> show2 min
                    'm' -> show2 (fromEnum month + 1)
                    'n' -> "\n"
                    'p' -> (if hour < 12 then fst else snd) (amPm l)
                    'R' -> doFmt "%H:%M"
                    'r' -> doFmt (time12Fmt l)
                    'T' -> doFmt "%H:%M:%S"
                    't' -> "\t"
                    'S' -> show2 sec
                    's' -> show2 sec
                    'X' -> doFmt (timeFmt l)
                    'x' -> doFmt (dateFmt l)
                    'Y' -> show year
                    'y' -> show2 (year `rem` 100)
                    '%' -> "%"
                    c   -> [c]

    defaultTimeDiffFmt (TimeDiff year month day hour min sec _)
      = foldr (\(v, s) rest -> (if v /= 0
                                then show v ++ ' ':(addS v s) ++ if null rest
```

```
                                                              then ""
                                                              else ", "
                                  else "") ++ rest)
          ""
          (zip [year, month, day, hour, min, sec] (intervals l))
  addS v s = if abs v == 1
               then fst s
               else snd s
```

Тип: `CalendarTime`

Описание: представляет время в удобном для пользователя формате. В единственном конструкторе собраны поля для представления текущих года, месяца, дня, часа, минуты, секунды, пикосекунды, наименования дня недели, номера дня в году, наименования временно́й зоны, отличие от UTC в секундах и флаг наличия режима летнего времени.

Определение:

```
data CalendarTime
  = CalendarTime
    {
      ctYear    :: Int
      ctMonth   :: Month
      ctDay     :: Int
      ctHour    :: Int
      ctMin     :: Int
      ctSec     :: Int
      ctPicosec :: Integer
      ctWDay    :: Day
      ctYDay    :: Int
      ctTZName  :: String
      ctTZ      :: Int
      ctIsDST   :: Bool
    }
```

Для данного типа определены экземпляры следующих классов: `Eq`, `Ord`, `Read` и `Show`.

Тип: `Month`

Описание: тип-перечисление для представления месяцев.

Определение:

```
data Month
  = January
  | February
  | March
  | April
  | May
  | June
  | July
  | August
  | September
  | October
  | November
  | December
```

Для данного типа определены экземпляры следующих классов: Bounded, Enum, Eq, Ix, Ord, Read и Show.

Тип: Day

Описание: тип-перечисление для представления дня недели.

Определение:

```
data Day
  = Sunday
  | Monday
  | Tuesday
  | Wednesday
  | Thursday
  | Friday
  | Saturday
```

Для данного типа определены экземпляры следующих классов: Bounded, Enum, Eq, Ix, Ord, Read и Show.

Функция: toCalendarTime

Описание: преобразует время во внутреннем представлении во внешнее представление типа CalendarTime. Поскольку результат зависит от локальных настроек (а потому недетерминирован), функция возвращает результат в монаде IO.

Определение:

```
toCalendarTime :: ClockTime -> IO CalendarTime
```

Функция определена в виде примитива.

Функция: toUTCTime
Описание: преобразует время во внутреннем представлении во внешнее представление типа CalendarTime в стандартном формате UTC.
Определение:

```
toUTCTime :: ClockTime -> CalendarTime
```

Функция определена в виде примитива.
Функция: toClockTime
Описание: производит обратное преобразование времени во внешнем представлении при помощи типа CalendarTime во внутреннее представление.
Определение:

```
toClockTime :: CalendarTime -> ClockTime
```

Функция определена в виде примитива.
Функция: calendarTimeToString
Описание: форматирует заданное время в соответствии с локальными соглашениями о формате времени.
Определение:

```
calendarTimeToString :: CalendarTime -> String
calendarTimeToString = formatCalendarTime defaultTimeLocale "%c"
```

Функция: formatCalendarTime
Описание: форматирует заданное время в соответствии с локальными соглашениями о формате времени и форматной строкой. Результат понимаем функцией strftime языка C.
Определение:

```
formatCalendarTime :: TimeLocale -> String -> CalendarTime -> String
formatCalendarTime l fmt (CalendarTime year mon day hour min sec _
                                          wday yday tzname _ _) = doFmt fmt
  where
    doFmt ('%':'-':cs) = doFmt ('%':cs)
    doFmt ('%':'_':cs) = doFmt ('%':cs)
    doFmt ('%':c:cs)   = decode c ++ doFmt cs
    doFmt (c:cs) = c : doFmt cs
    doFmt "" = ""


    decode 'A' = fst (wDays l  !! fromEnum wday)
```

```
decode 'a' = snd (wDays l  !! fromEnum wday)
decode 'B' = fst (months l !! fromEnum mon)
decode 'b' = snd (months l !! fromEnum mon)
decode 'h' = snd (months l !! fromEnum mon)
decode 'C' = show2 (year `quot` 100)
decode 'c' = doFmt (dateTimeFmt l)
decode 'D' = doFmt "%m/%d/%y"
decode 'd' = show2 day
decode 'e' = show2' day
decode 'H' = show2 hour
decode 'I' = show2 (to12 hour)
decode 'j' = show3 yday
decode 'k' = show2' hour
decode 'l' = show2' (to12 hour)
decode 'M' = show2 min
decode 'm' = show2 (fromEnum mon + 1)
decode 'n' = "\n"
decode 'p' = (if hour < 12 then fst else snd) (amPm l)
decode 'R' = doFmt "%H:%M"
decode 'r' = doFmt (time12Fmt l)
decode 'T' = doFmt "%H:%M:%S"
decode 't' = "\t"
decode 'S' = show2 sec
decode 's' = show2 sec
decode 'U' = show2 ((yday + 7 - fromEnum wday) `div` 7)
decode 'u' = show (let n = fromEnum wday
                   in  if n == 0
                         then 7
                         else n)
decode 'V' = let (week, days) = (yday + 7 - if fromEnum wday > 0
                                            then fromEnum wday - 1
                                            else 6) `divMod` 7
             in  show2 (if days >= 4
                          then week + 1
                          else if week == 0
                                 then 53
                                 else week)
decode 'W' = show2 ((yday + 7 - if fromEnum wday > 0
                                 then fromEnum wday - 1
                                 else 6) `div` 7)
decode 'w' = show (fromEnum wday)
decode 'X' = doFmt (timeFmt l)
decode 'x' = doFmt (dateFmt l)
decode 'Y' = show year
```

```
    decode 'y' = show2 (year `rem` 100)
    decode 'Z' = tzname
    decode '%' = "%"
    decode c   = [c]

show2 :: Int -> String
show2 x | x' < 10   = '0': show x'
        | otherwise = show x'
  where
    x' = x `rem` 100

show2' :: Int -> String
show2' x | x' < 10   = ' ': show x'
         | otherwise = show x'
  where
    x' = x `rem` 100

show3 :: Int -> String
show3 x = show (x `quot` 100) ++ show2 (x `rem` 100)

to12 :: Int -> Int
to12 h = let h' = h `mod` 12
         in  if h' == 0
               then 12
               else h'
```

Глава 12.

Пакет модулей `Text`

Наконец, пакет модулей `Text` предназначен для работы с текстом, со строками. В этом пакете собраны модули, которые расширяют стандартные возможности языка и предоставляют программисту гибкие инструменты для обработки строковых данных.

12.1. Модуль `Printf`

В модуле `Printf` содержатся определения программных сущностей, которые позволяют форматировать строки на основе значений других типов в соответствии с поведением функции `printf` (и ей подобных) из языка C. Использование:

```
import Text.Printf
```

В модуле имеются две функции и четыре специализированных класса, которые позволяют представлять типы для использования с функциями. Первыми имеет смысл рассмотреть именно функции, поскольку классы являются вспомогательными для них.

Функция: `printf`
Описание: форматирует произвольное количество аргументов в стиле языка C. Функция может вернуть значение типа `String` либо `IO a`.

Определение:

```
printf :: (PrintfType r) => String -> r
printf fmt = spr fmt []
```

Функция: hPrintf
Описание: вариант функции printf, выводящий результат в заданный поток (файл) по дескриптору.
Определение:

```
hPrintf :: (HPrintfType r) => Handle -> String -> r
hPrintf hdl fmt = hspr hdl fmt []
```

Строка форматирования для использования в функциях printf и hprintf состоит из обычных символов, расположенных согласно спецификации форматирования. Спецификация форматирования начинается с символа (%), после которого идёт один или несколько флагов следующего вида:

1) _ — форматирование с выравниванием по левому краю (по умолчанию — по правому);

2) 0 — заполнение пустого пространства нулями (лидирующими) вместо пробелов.

После этих флагов идёт необязательное указание длины поля:

1) # — длина поля для форматирования в символах;

2) * — то же самое, но значение берётся из списка аргументов.

Далее указывается точность:

1) .# — количество десятичных знаков;

И наконец, символы форматирования:

1) c — символы (типы Char, Int и IntegeR);

2) d — числа в десятичной записи (типы Char, Int и Integer);

3) o — числа в восьмиричной записи (типы Char, Int и Integer);

4) x — числа в шестнадцатиричной записи (типы Char, Int и Integer);

5) u — беззнаковые числа в десятичной записи (типы Char, Int и Integer);

6) f — числа с плавающей точкой (типы Float и Double);

7) g — обычный формат чисел с плавающей точкой (типы Float и Double);

8) e — экспоненциальный формат чисел с плавающей точкой (типы Float и Double);

9) s — строки (тип String).

Несоответствие типов аргументов спецификации форматирования приводит к генерации исключения во время процесса исполнения.

Класс: PrintfType

Описание: предоставляет возможность использования произвольного количества аргументов в функции printf. Реализация этого класса скрыта.

Определение:

```
class PrintfType t where
  spr :: String -> [UPrintf] -> t
```

В качестве экземпляров этого класса возможны типы: a -> r, где тип r также является экземпляром этого класса; а также типы IO и [].

Класс: HPrintfType

Описание: предоставляет возможность использования произвольного количества аргументов в функции hprintf. Реализация этого класса также скрыта.

Определение:

```
class HPrintfType t where
  hspr :: Handle -> String -> [UPrintf] -> t
```

В качестве экземпляров этого класса возможны типы: a -> r, где тип r также является экземпляром этого класса; а также тип IO.

Класс: PrintfArg

Описание: определяет общий интерфейс для типов, которые могут выступать типами аргументов функций printf и hprintf.

Определение:

```
class PrintfArg a where
  toUPrintf :: a -> UPrintf
```

Экземплярами этого класса являются следующие типы: `Char`, `Double`, `Float`, `Int`, `Integer` и `[]`.

Класс: `IsChar`

Описание: предоставляет интерфейс для перекодирования значений произвольного типа в тип `Char` и обратно.

Определение:

```
class IsChar c where
  toChar   :: c -> Char
  fromChar :: Char -> c
```

Экземпляром этого класса является тип `Char`.

Остаётся отметить, что электронный адрес ответственного за этот модуль в поставке языка Haskell следующий: `lennart@augustsson.net`.

12.2. Модуль Read

В модуле `Read` определены программные сущности, позволяющие преобразовывать строки в значения произвольных типов. Этот модуль, как и многие иные в стандартной поставке, является экспериментальным, созданным с целью разгрузки стандартного модуля `Prelude` от программных сущностей. Правда, необходимо отметить, что в этом модуле для компилятора GHC имеются расширенные возможности по использованию функций синтаксического анализа. В частности, создана улучшенная версия класса `Read` (см. стр. 121), которая позволяет производить более эффективный синтаксический анализ.

Использование модуля:

```
import Text.Read
```

Большинство программных сущностей из этого модуля также определены в стандартном модуле `Prelude`, а потому описаны ранее в этом справочнике. К таким программным сущностям относятся уже упоминавшийся класс `Read`, а также функции `read` (см. стр. 152), `reads` (см. стр. 156), `readParen` (см. стр. 156) и `lex`

(см. стр. 141). Ниже описываются только дополнительные программные сущности.

Тип: `Lexeme`

Описание: тип для представления лексем различных типов. Каждый конструктор отвечает за определённый тип лексем, которые могут встретиться при разборе конструкций при помощи методов класса `Read`.

Определение:

```
data Lexeme
  = Char Char
  | String String
  | Punc String
  | Ident String
  | Symbol String
  | Int Integer
  | Rat Rational
  | EOF
```

Конструктор `Char` отвечает за символы (обычно используемые в качестве идентификаторов). Конструктор `String` отвечает за символьные последовательности, в нём содержатся проинтерпретированные escape-коды. Конструктор `Punc` представляет символы пунктуации. Конструктор `Ident` описывает идентификаторы языка Haskell. Также и конструктор `Symbol` описывает символьные операции языка Haskell (например, `(:%)` и т. д.). Конструкторы `Int` и `Rat` описывают целые и действительные числа соответственно. Наконец, конструктор `EOF` описывает символ конца файла.

Данный тип имеет экземпляры следующих классов: `Eq`, `Read` и `Show`.

Функция: `lexP`

Описание: производит синтаксический анализ одной лексемы.

Определение:

```
lexP :: ReadPrec Lexeme
lexP = lift L.lex
```

Функция: `parens`

Описание: производит синтаксический анализ выражения в скобках (в произвольном количестве правильно расставленных, согласованных скобок).

Определение:

```
parens :: ReadPrec a -> ReadPrec a
parens p = optional
  where
    optional  = p +++ mandatory
    mandatory = do L.Punc "(" <- lexP
                   x          <- reset optional
                   L.Punc ")" <- lexP
                   return x
```

Функция: `readListDefault`
Описание: функция, заменяющая собой метод `readList` класса `Read`. Используется только в стандартной поставке компилятора GHC.
Определение:

```
readListDefault :: Read a => ReadS [a]
```

Функция определена в виде примитива.

Функция: `readListPrecDefault`
Описание: функция, заменяющая собой метод `readPrec` класса `Read`. Используется только в стандартной поставке компилятора GHC.
Определение:

```
readListPrecDefault :: Read a => ReadPrec [a]
```

Функция определена в виде примитива.

Необходимо отметить, что данный модуль предоставляет базовые, а оттого примитивные средства для проведения синтаксического анализа. В поставке библиотек языка Haskell имеется мощная библиотека, заточенная специально для решения задач синтаксического анализа и смежных проблем. Наименование этой библиотеки — `Parsec`, основана она на монадах и предоставляет приятный синтаксис для написания парсеров. Рассмотрение этой библиотеки выходит за рамки этого небольшого справочника.

12.2.1. Модуль `Lex`

Модуль `Lex` является дополнительным модулем для проведения синтаксического анализа, в который вынесены функции непосредственного разбора входных строк на лексемы. В этом модуле также определён тип `Lexeme` (см. стр. 532), ко-

торый используется для представления лексем. Также здесь определена главная функция, осуществляющая лексический просмотр входных строк. Дополнительно определены несколько утилитарных функций для разбора различных числовых значений.

Использование модуля:

```
import Text.Read.Lex
```

Функция: `hsLex`

Описание: производит синтаксический анализ строки, которая понимается в качестве строки исходного кода на языке Haskell. Данная функция вызывает функцию `lexToken`, которая и осуществляет лексический анализ, но которая в то же время достаточно громоздка. Любознательный читатель может обратиться к исходному коду модуля, чтобы изучить способ разбора, предлагаемый этой функцией.

Определение:

```
hsLex :: ReadP String
hsLex = do skipSpaces
       (s, _) <- gather lexToken
       return s
```

Функция: `readIntP`

Описание: производит синтаксический анализ строки, представляющей число в произвольной системе счисления.

Определение:

```
readIntP :: Num a => a -> (Char -> Bool) -> (Char -> Int) -> ReadP a
readIntP base isDigit valDigit = do s <- munch1 isDigit
                                    return (val base 0 (map valDigit s))


readIntP' :: Num a => a -> ReadP a
readIntP' base = readIntP base isDigit valDigit
  where
    isDigit  c = maybe False (const True) (valDig base c)
    valDigit c = maybe 0 id (valDig base c)
```

Функция: `readOctP`

Описание: производит синтаксический анализ строки, представляющей число в восьмиричной системе счисления.

Определение:

```
readOctP :: Num a => ReadP a
readOctP = readIntP' 8
```

Функция: readDecP
Описание: производит синтаксический анализ строки, представляющей число в десятичной системе счисления.
Определение:

```
readDecP :: Num a => ReadP a
readDecP = readIntP' 10
```

Функция: readHexP
Описание: производит синтаксический анализ строки, представляющей число в шестнадцатиричной системе счисления.
Определение:

```
readHexP :: Num a => ReadP a
readHexP = readIntP' 16
```

12.3. Модуль Show

В модуле Show определены программные сущности, позволяющие преобразовывать значения произвольных типов в строки. Этот модуль, как и многие иные в стандартной поставке, является экспериментальным, созданным с целью разгрузки стандартного модуля Prelude от программных сущностей. Использование модуля:

```
import Text.Show
```

Подавляющее большинство программных сущностей из этого модуля также определены в стандартном модуле Prelude, а потому описаны ранее в этом справочнике. К таким программным сущностям относятся класс Show (см. стр. 124), а также функции shows (см. стр. 162), showChar (см. стр. 160), showString (см. стр. 162) и showParen (см. стр. 161). Ниже описываются только дополнительные программные сущности.

Единственной функцией, которая определена дополнительно к функциям из стандартного модуля Prelude, является функция showListWith.

Функция: showListWith

Описание: преобразует заданный список в строку, в которой элементы разделены запятыми и обрамлены в квадратные скобки [], а сами элементы преобразованы в строку при помощи заданной функции.

Определение:

```
showListWith :: (a -> ShowS) -> [a] -> ShowS
showListWith _     []     s = "[]" ++ s
showListWith showx (x:xs) s = '[' : showx x (showl xs)
  where
    showl []     = ']' : s
    showl (y:ys) = ',' : showx y (showl ys)
```

12.3.1. Модуль Functions

Специальный модуль, который предоставляет альтернативный вариант экземпляра класса Show (см. стр. 124) для функциональный типов. Использование:

```
import Text.Show.Functions
```

В этом модуле описан единственный экземпляр класса Show для типа a -> b. В соответствии с идеологией справочника определение этого экземпляра здесь не приводится.

Заключение

Справочник окончен. Конечно, автор не претендует на полноту изложения и актуальность информации. В книге показан срез положения функционального языка программирования Haskell на 2007 год. Само собой разумеется, что язык постоянно развивается, а потому через некоторое время информация в этом справочнике может перестать быть актуальной. Тем не менее автор надеется, что работа над книгой проделана не зря, и книга в действительности поможет многим как в работе, так и в обучении.

Литература

[1] Душкин Р. В. Функциональное программирование на языке Haskell. — М.: ДМК Пресс, 2007. — 608 стр., ил. ISBN 5-94074-335-8.

[2] Adams S. Efficient sets: a balancing act. In *Journal of Functional Programming*, 3(4). pp. 553—562, October 1993. (В интернете доступно по адресу: http://www.swiss.ai.mit.edu/ adams/BB/).

[3] Burton F. W., Page R. L. Distributed random number generation. *Journal of Functional Programming*, 2(2), April 1992, pp. 203—212.

[4] Carta D. G. Two fast implementations of the minimal standard random number generator. *Comm ACM*, 33(1), Jan 1990, pp. 87—88.

[5] Erkok L. Value Recursion in Monadic Computations. Oregon Graduate Institute, 2002.

[6] Gibbons J., Oliveira B. The Essence of the Iterator Pattern. In *Mathematically-Structured Functional Programming*, 2006. (В интернете доступно по адресу: http://web.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/#iterator).

[7] Hellekalek P. Don't trust parallel Monte Carlo. Department of Mathematics, University of Salzburg, 1998. (В интернете доступно по адресу: http://random.mat.sbg.ac.at/ peter/pads98.ps).

[8] Hinze R., Paterson R. Finger trees: a simple general-purpose data structure. In *Journal of Functional Programming* 16:2 (2006). pp. 197—217. (В интернете доступно по адресу: http://www.soi.city.ac.uk/ ross/papers/FingerTree.html).

[9] Hughes J. Generalising Monads to Arrows. Science of Computer Programming 37, pp. 67—111, May 2000.

[10] Jones M. P. Functional Programming with Overloading and Higher-Order Polymorphism. Advanced School of Functional Programming, 1995. (В интернете доступно по адресу: http://www.cse.ogi.edu/ mpj/).

[11] King D., Launchbury J. Lazy Depth-First Search and Linear Graph Algorithms in Haskell.

[12] Larson P.-A. Dynamic Hash Tables, In CACM 31(4), April 1988, pp. 446—457.

[13] Launchbury J., Jones S. P. Lazy Functional State Threads. In *PLDI 94*.

[14] L'Ecuyer P. Efficient and portable combined random number generators. *Comm ACM*, 31(6), Jun 1988, pp. 742—749.

[15] McBride C., Paterson R. Applicative Programming with Effects. In *Journal of Functional Programming*. (В интернете доступно по адресу: http://www.soi.city.ac.uk/ ross/papers/Applicative.html).

[16] Morrison D. R. PATRICIA — Practical Algorithm To Retrieve Information Coded In Alphanumeric, Journal of the ACM, 15(4), October 1968, pp. 514—534.

[17] Nievergelt J., Reingold E. M. Binary search trees of bounded balance, In *SIAM journal of computing*, 2(1), March 1973.

[18] Okasaki C. Breadth-First Numbering: Lessons from a Small Exercise in Algorithm Design. ICFP'00.

[19] Okasaki C., Gill A., Fast Mergeable Integer Maps, Workshop on ML, September 1998, pp. 77—86. (В интернете доступно по адресу: http://www.cse.ogi.edu/ andy/pub/finite.htm).

[20] Park S. K., Miller K. W. Random number generators — good ones are hard to find. *Comm ACM*, 31(10), Oct 1988, pp. 1192—1201.

[21] Paterson R. A New Notation for Arrows. In ICFP 2001, Firenze, Italy, pp. 229—240.

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС-КНИГА» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Internet-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. **(495) 258-91-94, 258-91-95**; электронный адрес **books@alians-kniga.ru**.

Душкин Роман Викторович

Справочник по языку Haskell

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Корректор *Кикава Л. В.*
Верстка *Душкин Р. В.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 02.11.2007. Формат $70\times100^{1}/_{16}$.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 51. Тираж 1500 экз.

Издательство ДМК Пресс.
Web-сайт издательства: www.dmk-press.ru
Internet-магазин: www.abook.ru