

Справочник по языку Haskell

**Синтаксис языка, описание
стандартных библиотек,
методы правильного
программирования**



Душкин Р. В.

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 части 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

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 5-94074-410-9

© Душкин Р. В., 2008
© Оформление ДМК Пресс, 2008

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 <code>Logic</code>	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 <code>Applicative</code>	172
7.2. Модуль <code>Arrow</code>	176
7.3. Модуль <code>Concurrent</code>	181
7.3.1. Модуль <code>Chan</code>	186
7.3.2. Модуль <code>MVar</code>	188
7.3.3. Модуль <code>QSem</code>	192
7.3.4. Модуль <code>QSemN</code>	193
7.3.5. Модуль <code>SampleVar</code>	194
7.4. Модуль <code>Exception</code>	197
7.5. Модуль <code>Monad</code>	211
7.5.1. Модуль <code>Fix</code>	220
7.5.2. Модуль <code>Instances</code>	222
7.5.3. Модуль <code>ST</code>	222
7.6. Модуль <code>Parallel</code>	224
8. Пакет модулей Data	226
8.1. Модуль <code>Array</code>	226
8.1.1. Модуль <code>Base</code>	231
8.1.2. Модуль <code>Diff</code>	231
8.1.3. Модуль <code>IArray</code>	233
8.1.4. Модуль <code>IO</code>	234
8.1.5. Модуль <code>MArray</code>	237
8.1.6. Модуль <code>ST</code>	241
8.1.7. Модуль <code>Storable</code>	243

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

8.31. Модуль <code>Typeable</code>	408
8.32. Модуль <code>Unique</code>	414
8.33. Модуль <code>Version</code>	415
8.34. Модуль <code>Word</code>	416
9. Пакет модулей <code>Debug</code>	419
9.1. Модуль <code>Trace</code>	419
10. Пакет модулей <code>Foreign</code>	421
10.1. Модуль <code>C</code>	422
10.1.1. Модуль <code>Error</code>	423
10.1.2. Модуль <code>String</code>	429
10.1.3. Модуль <code>Types</code>	436
10.2. Модуль <code>ForeignPtr</code>	439
10.3. Модуль <code>Marshal</code>	444
10.3.1. Модуль <code>Alloc</code>	445
10.3.2. Модуль <code>Array</code>	447
10.3.3. Модуль <code>Error</code>	453
10.3.4. Модуль <code>Pool</code>	455
10.3.5. Модуль <code>Utils</code>	459
10.4. Модуль <code>Ptr</code>	462
10.5. Модуль <code>StablePtr</code>	466
10.6. Модуль <code>Storable</code>	468
11. Пакет модулей <code>System</code>	471
11.1. Модуль <code>Cmd</code>	471
11.2. Модуль <code>CPUTime</code>	472
11.3. Модуль <code>Directory</code>	473
11.3.1. Модуль <code>Internals</code>	482
11.4. Модуль <code>Environment</code>	482
11.5. Модуль <code>Exit</code>	484
11.6. Модуль <code>Info</code>	485
11.7. Модуль <code>IO</code>	486
11.7.1. Модуль <code>Error</code>	497
11.7.2. Модуль <code>Unsafe</code>	507
11.8. Модуль <code>Locale</code>	507

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

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, являющимися 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 кюз, то есть одно выражение, определяющее значение функции. В функциональном программировании принято, что функция может быть определена несколькими кюзами, каждый из которых характеризуется определённым набором образцов. Например, вот функции для оперирования со списками — их определения состоят из двух кюзов:

```
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 кюзов в определении функции является естественным способом ветвления алгоритма в функциональном программировании. При этом в разных языках функционального программирования используется различный способ обработки кюзов. В языке Haskell весьма важен порядок кюзов, так как транслятор просматривает набор кюзов 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 кюз. Если список состоит из более чем одного элемента, транслятор пропускает первый кюз и выбирает второй. Если бы кюзы в этих функциях стояли наоборот, то первый кюз 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 кюзов 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 обзвать 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 вычислительным 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. Прodelать 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 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 $(n + k)$

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 клон 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 кюз 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 «подязык», в рамках которого функции с определённым типом могли выполнять действия (вызывать побочный эффект изменения внешнего окружения — устройств вывода) и быть недетерминированными. Однако вне этого подязыка 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 обернуты контейнером