Оглавление

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`: