Функциональное программирование на языке Haskell

Лучшая практическая реализация функциональной парадигмы программирования

На CD содержится:

- трансляторы Haskell
- библиотеки и утилиты
- примеры программ

Душкин Р. В.

ДМК ПРЕСС
издательство

Душкин Р. В.

Функциональное программирование на языке Haskell

Москва, 2006

УДК 004.4
ББК 32.973.26-018.2
Д86

Душкин Р. В.
Д86 Функциональное программирование на языке Haskell. М.: ДМК Пресс, 2006. 608 с., ил.

ISBN 5-94074-335-8

Данная книга является первым в России изданием, рассматривающая функциональное программирование в полном объеме, досточном для понимания новичку и для использования книги в качестве справочного пособия теми, кто уже использует парадигму функционального программирования в своей практике. Изучсние прикладных основ показано на примере языка Haskell, на сегодняшний день являющегося самым мощным и развитым инструментом функционального программирования.

Издание можно использоватьи в качестве учебника по функциональному программированию, и в качестве самостоятельного учебного пособия по смежным дисциплинам, в первую очередь по комбинаторной логике и λ-исчислению.

Также книга будет интересна тем, кто всерьез занимается изучением новых компьютерных технологий, искусственного интеллекта и экспертных систем.

К книге придагается компакт-диск с транслятором Haskell, а также различными библиотеками к нему, дополнительными утилитами и рабочими примерами программ, рассмотренных в книге.

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.

© Душкин Р. В., 2006
© Оформление ДМК Пресс, 2006

ISBN 5-94074-335-8

Оглавление

Содержание

Введение

Данная книга является первым в России изданием, рассматривающим функциональное программирование в полном объеме, достаточном и для понимания новичку, и для использования книги в качестве справочного пособия теми, кто уже использует парадигму[1] функционального программирования в своей практике. Эту книгу можно использовать и в качестве учебника по функциональному программированию, и в качестве вспомогательного учебного пособия по смежным дисциплинам, в первую очередь по комбинаторной логике и λ-исчислению[2]. Также книга будет интересна тем, кто всерьез занимается изучением новых компьютерных технологий, искусственного интеллекта и синергетическим слиянием научных направлений человеческой деятельности.

Необходимость написания подобной книги в первую очередь вызвана тем, что на русском языке нет полноценного справочного и учебного пособия по языку Haskell, в то время как этот функциональный язык все больше и больше завоевывает сердца программистов по всему миру. Более того, этот язык уже используют и для написания полноценных программных систем, в том числе для коммерческого использования. Однако в России изучение языка Haskell проповедуется лишь энтузиастами, а в строгой академической школе довольствуются традиционным рассмотрением языка Lisp в качестве иллюстрации основ функционального программирования.

[1] Под парадигмой (от греч. παράδειγμα — пример, модель, образец) здесь и далее понимается общая концептуальная схема постановки проблем и методов их решения. В более узком смысле под парадигмой программирования будет пониматься не просто стиль написания программ, а способ мышления, который позволяет использовать тот или иной стиль при создании таких программ.

[2] Основы комбинаторной логики и λ-исчисление кратко рассматриваются в главе 5 этой книги.

Однако если рассматривать англоязычные источники, руководства и учебники по языку Haskell, то налицо сложности, с которыми сталкиваются те, кто начинает самостоятельно изучать этот функциональный язык. Во-первых, это скупой и формальный подход при описании синтаксиса языка — на этом принципе основаны все руководства по языку Haskell. Во-вторых, это отсутствие какой-либо системности, или даже присутствие антисистемности в изложении как основ функционального программирования, так и способов программирования на языке Haskell. Например, в учебнике «Haskell: The Craft of Functional Programming» автор перескакивает с одной темы на другую безо всяких переходов между ними. Начинает с описания списков, а продолжает принципами создания программного обеспечения вообще. Описывает систему классов, перескакивает на описание системы ввода/вывода практически без затрагивания темы монад. А это — один из основных учебников по языку Haskell на английском языке.

Книга рассчитана на всех, кто интересуется современными тенденциями развития компьютерной науки и технологии, а также искусственным интеллектом. Она может служить основным или дополнительным учебным пособием для студентов и аспирантов, обучающихся по специальности «прикладная математика» и специализации «искусственный интеллект». Также книга будет полезна в качестве справочного материала по языку Haskell всем тем, кто использует функциональную парадигму в целом и этот функциональный язык в частности в научных исследованиях или повседневной работе.

Для чтения книги необходимо обладать базовыми познаниями в дискретной математике, а также иметь представление о методах разработки алгоритмах и алгоритмическом решении задач (теория алгоритмов). В процессе написания книги полагалось, что читатель знаком с базовыми концептами дискретной математики, поэтому лишние математические определения в книге не приводятся. Сложные разделы математики, представленные в заключительных главах, требуют более серьезных знаний, поэтому их описание сопровождается небольшими экскурсами в теорию. Это касается таких разделов дискретной математики, как комбинаторная логика, λ-исчисление, математическая лингвистика и теория построения трансляторов, а также базовые принципы искусственного интеллекта.

Рассмотрение парадигмы функционального программирования производится на примере языка Haskell, на котором написаны все исходные коды, представленные в книге. Все представленные примеры, функции, модули и исходные тексты программ проверены в интегрированной среде разработки HUGS 98 (за исклю-

чением тех, которые предназначены для компиляции в среде GHC — Glasgow Haskell Compiler). Для удобства читателя, желающего проверить и закрепить свои знания на практике, все приведенные в книге программы и функции записаны на прилагаемом CD-диске, который выпущен в дополнение к книге ограниченным тиражом.

Книгу нельзя рассматривать в качестве скрупулезного введения в язык Haskell и программирования на нем, так как цель автора — не преподать некоторую догму относительно использования языка Haskell, но направить читателя на путь, следуя которому он сам сможет понять, что и как необходимо делать для получения правильных программ. В связи с этим в книге не рассматриваются вопросы о том, какие инструкции на языке необходимо написать, чтобы получить определенный эффект (или рассматриваются в самом минимальном виде), но предлагаются пути решения разных задач, встающих перед программистом. Именно поэтому главы, непосредственно посвященные языку Haskell, написаны сухим языком. Цель этих глав — преподнести читателю как можно больше информации о синтаксисе языка без лишнего упоминания о том, как использовать предлагаемые синтаксические конструкции. Для этого необходимо обратиться к последним главам книги, где, однако, упоминание о языке Haskell сведено к минимуму. Такой формат представления информации позволит (и даже принудит) читателю самостоятельно следовать по пути использования парадигмы функционального программирования.

Также на прилагаемом CD-диске можно найти текст данной книги в формате Adobe PDF (Portable Document Format) для использования его в личных нуждах.

Краткая биография автора

Душкин Роман Викторович. С 2001 г. читает лекции по функциональному программированию для студентов четвертого курса кафедры кибернетики (№ 22) факультета «К» Московского инженерно-физического института (МИФИ). Базисом для авторского курса по функциональному программированию послужил текст лекций Г. М. Сергиевского, читавшихся до 2001 г. Данные лекции были кардинально переработаны с учетом современных веяний в науке и технике, основа была переведена с языка Lisp на Haskell (вместе с полной переработкой курса лабораторных работ), а сами лекции были дополнены строгим научным обоснованием как самой парадигмы функционального программирования, так и ее прикладных аспектов.

Р. В. Душкин является автором множества научных публикаций по темам нечеткой математики, искусственного интеллекта и функционального программирования в российских и зарубежных научных изданиях. Участвовал во множестве национальных и международных научных конференций, проводимых под эгидой Российской Ассоциации Искусственного Интеллекта. Издал ряд методических пособий, в том числе и для выполнения лабораторных работ по функциональному программированию.

Р. В. Душкин работает в области создания автоматизированных систем управления на железнодорожном транспорте, на практике используя все методы создания программных средств, применяющиеся в составе парадигм функционального и объектно-ориентированного программирования, а также искусственного интеллекта. Входил в состав команды разработчиков и управлял самим процессом разработки множества автоматизированных систем, в том числе и реального времени, для информационной и технологической поддержки процессов управления в различных областях железнодорожной отрасли России.

О пользовании книгой

Для удобства читателей при наборе текста книги использовались шрифты различных начертаний для выделения тех или иных особенностей представленной информации.

Для написания имен функций, элементов формул и математических выражений, которые приводятся непосредственно в тексте книги, используется курсивное начертание. Например: $x \in C$, f_0, $List(A)$.

Примеры фрагментов программ на языке Haskell и изредка на абстрактном функциональном языке, который используется для объяснения теоретических аспектов применения парадигмы функционального программирования, в тексте книги приводятся при помощи моноширинной гарнитуры (машинописного шрифта):

```
length [] = 0
length (x:xs) = 1 + length xs
```

Упоминания об именах функций непосредственно в тексте также выделяются моноширинным начертанием символов: `length`, `append`. Кроме того, таким же образом выделяются наименования модулей, стандартных библиотек и дополнительных программных средств: `Prelude`, `Parsec`, `Happy`.

Иногда в тексте встречаются обозначения операций, наименования которых состоят из одного или более нелитеральных символов. Для того чтобы как-то отделять подобные обозначения от текста книги, такие операции заключаются в круглые скобки, а сами обозначения приводятся моноширинным шрифтом. Например: (+), (<@), (>>=). Сами скобки различного вида в круглые скобки не заключаются: {}, [] и т. д. в случае, если необходимо показать отдельную скобку, она заключается в кавычки: «(», «}» и т. д.

Листинги больших и законченных модулей на языке Haskell приводятся в виде выделенных блоков:

Листинг 0.1. Пример текста программы из внешнего файла

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--                                                                            --
-- Модуль INTROEXAMPLE - пример правильно описанного модуля на языке Haskell. --
--                                                                            --
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
module IntroExample
  (someFunction)
where

--------------------------------------------------------------------------------
-- Некоторая функция, выполняющая определённые действия.
--
-- Входные параметры:
--   n      - число, которое необходимо прибавить к каждому элементу списка.
--   (x:xs) - список, к каждому элементу которого необходимо прибавить число n.
--
-- Возвращаемое значение:
--   Список, составленный из сумм заданного числа n с элементами исходного
--   списка.

someFunction :: Int -> [Int] -> [Int]
someFunction n [] = []
someFunction n (x:xs) = (x + n) : someFunction n xs

--[ КОНЕЦ МОДУЛЯ ]------------------------------------------------------------
```

Еще раз необходимо заметить, что все приведенные в книге листинги читатель сможет найти на прилагаемом к книге CD-диске или на официальном сайте книги в Интернете.

Состав и структура представления информации

Книга разбита на семь больших глав, каждая из которых посвящена отдельному аспекту парадигмы функционального программирования. Каждая глава разбита на пять разделов, которые логически делят главу на несколько отдельных областей рассмотрения.

Первая глава является своеобразным введением в проблематику — в ней приводится базовый методический материал, необходимый для понимания основ функционального программирования и дальнейшего чтения книги. По существу, первая глава — это пролог, который открывает путь в функциональный мир.

Вторая глава повествует о базовых принципах языка Haskell, в ней приводятся самые основные сведения о синтаксисе, структуре программ и модулей, взаимодействии функций, организации исходного кода и прочих аспектах использования нового языка программирования. Данную главу можно использовать в качестве учебного пособия для тех, кто впервые приступил к изучению языка Haskell.

Третья глава посвящена слиянию функциональной и объектно-ориентированной парадигм программирования, что должно представить язык Haskell в качестве современного средства разработки компьютерных приложений. Рассматривается такое базовое понятие объектно-ориентированного программирования, как «класс», и его применение в составе средств языка Haskell. Кроме того, приводится более или менее полное описание стандартных классов и их экземпляров, поставляемых в составе базового модуля `Prelude`.

Четвертая глава рассматривает такую незаурядную вещь, как «монада» — расширение парадигмы функционального программирования для включения в нее императивного подмножества структур языка Haskell для выполнения операций, связанных с использованием побочных эффектов (в первую очередь операций ввода/вывода). Так же как и в третьей главе, приводится подробное описание стандартных монад, которое поможет понять этот непростой объект и полноценно использовать язык Haskell для разработки сложных приложений.

По существу, для изучения самого языка Haskell достаточно прочитать первые четыре главы. С пятой главы начинается рассмотрение теории и услож-

ненного материала, который поможет углубить понимание функционального программирования и вывести читателя на новый уровень знаний.

Пятая глава рассматривает теоретические основы функционального программирования, из которых, собственно, данная парадигма и вышла в первой половине XX в. Приводится описание комбинаторной логики, разработанной Х. Карри, и λ-исчисления, введенного в математический аппарат А. Черчем. Приводятся основные аспекты данных математических формализмов, которые помогут понять смысл и суть работы интерпретаторов функциональных языков, в том числе и интерпретаторов языка Haskell.

Шестая глава посвящена математической лингвистике, которая является теоретическим механизмом для построения трансляторов различных языков программирования. В главе приводятся базовая теория, принципы построения трансляторов на языках программирования, а также применение теории к самому языку Haskell — на примерах показано написание интерпретатора языка Haskell на нем самом.

Седьмая глава повествует о такой области человеческого знания, как искусственный интеллект, в котором парадигма функционального программирования нашла самое непосредственное применение. Глава не ставит целью дать скрупулезное исследование темы искусственного интеллекта, но вкратце рассматривает несколько основных задач, которые традиционно решаются методами искусственного интеллекта. Естественно, что все рассмотрение происходит в канве изучения языка Haskell.

Также в книге имеются четыре приложения, где собран интересный материал, который может помочь в деле изучения стези функционального программирования, но который, однако, выходит за рамки общей канвы книги. Вдумчивый читатель найдет в этих приложениях интересные для себя вещи, которые позволят ему отправиться в свободное плавание по океану функционального программирования.

Материал представлен таким образом, что не каждую главу и не каждый раздел в любой главе необходимо читать для понимания и проникновения в суть отдельных моментов в парадигме функционального программирования. В начале каждой главы и каждого раздела внутри главы имеется краткая аннотация для соответствующего уровня рубрикации, по которой читатель сможет понять и осмыслить необходимость изучения главы или раздела.

Благодарности

Первая благодарность выражается В. А. Роганову за тот импульс, который был дан для того чтобы только сесть за написание этой книги. Без этого импульса идея книги продолжала бы лежать в ящике рабочего стола, так и не воплотившись в чем-то серьезном, пока не устарела бы морально. Без помощи А. Н. Преображенского и А. Ю. Фоменко верстка книги в системе LaTeX заняла бы слишком много времени, которое было бы отнято у непосредственного написания текста и изложения смысла.

Автор выражает благодарность всем своим студентам, обучавшимся на кафедре кибернетики в МИФИ, помогавшим делать курс лекций по функциональному программированию более адаптированным для понимания неподготовленными новичками. Особая благодарность всем тем, кто занимался технической работой по сбору, переводам, адаптации и верстке материалов, изданных на иностранных языках.

Автор благодарит всех людей, принявших участие в обсуждении черновиков книги и внесших свой посильный вклад в ее создание. Перечислить поименно всех таких людей ввиду их огромного количества не представляется возможным. Однако без их активного участия книга была бы неполной, пресноватой и лишенной того шарма, который привносит в любую книгу интерактивное обсуждение в процессе ее создания.

Особо необходимо отметить помощь следующих людей, бескорыстно занимавшихся чтением и правкой черновиков книги, вносивших дельные комментарии и предложения: Д. А. Антонюк, В. Г. Владимиров, М. А. Забелин, И. О. Кабанова, Ю. Д. Лобарев, В. Н. Назаров, М. П. Трескин, А. В. Тупóта, Д. А. Храпов.

Огромное сердечное спасибо выражается жене Елене и сыну Кириллу за то, что они есть, за их поддержку и полное понимание во время работы над рукописью.

Контактная информация

Автор будет рад получить от своих читателей любые замечания, комментарии и просто отзывы о данной книге по следующему адресу электронной почты: `darkus.14@gmail.com`

Официальный web-ресурс книги находится по адресу `fp.haskell.ru`.

1.4 Конструирование функций

Математические доказательства схематически могут рассматриваться как упражнение в комбинировании двух приспособлений, которые мы можем назвать интуицией и изобретательностью.

Алан Тьюринг

Рассмотренная в разделе 1.3 типовая задача по созданию динамических структур данных возникла, быть может, вместе с самой информатикой как наукой об обработке информации. С тех пор было предложено большое количество методов и формализмов для описания таких структур данных, а также для их последующей обработки.

В этом разделе книги рассматривается метод синтаксически ориентированного конструирования, предложенный британским математиком Чарльзом Хоаром. Он предложил использовать некоторый метаязык, который позволяет описывать структуру данных любой сложности, в том числе и определяемую рекурсивно через саму себя.

Кроме того, метод синтаксически ориентированного конструирования позволяет решить не только задачу по разработке динамических структур данных, но и по автоматическому созданию шаблонов функций для обработки этих данных. Более того, этот метод в том числе отчасти подходит и для решения третьей типовой задачи функционального программирования, а именно — доказательства свойств функций.

Рассмотрение метода необходимо начать с метаязыка, который используется для описания типов данных. Данный метаязык включает в себя две операции — декартово произведение и размеченное объединение, а также несколько служебных слов для описания различных аспектов проектируемых типов данных. Служебные слова приводятся по мере рассмотрения упомянутых операций.

Декартово произведение. Декартово произведение определяется стандартным для математики образом (равно как и обозначается стандартно). Если $C_1, \ldots, C_n$ — это типы, а C — это тип, состоящий из множества кортежей вида $\langle c_1, \ldots, c_n \rangle, c_i \in C_i, i = \overline{1,n}$, то говорится, что C — декартово произведение типов $C_1, \ldots, C_n$ и обозначается как:

$$C = C_1 \times \ldots \times C_n. \tag{1.11}$$

Таким образом, операция декартова произведения строит гиперпространство на заданных множествах значений (типах). Каждая точка такого гиперпространства определяется набором координат, соответствующих значениям определенного типа, из которых состоит само декартово произведение. Другая математическая аналогия — вектор.

В методике синтаксически ориентированного конструирования к самому декартову произведению прилагаются два типа операций (функций). Это конструктор (обозначается как «constructor») и множество селекторов (обозначаются как «selectors»). Функция-конструктор конструирует тип, являющийся декартовым произведением. Каждая функция-селектор выбирает из значения созданного типа соответствующее назначению селектора значение базового типа, из которых состоит декартово произведение.

Запись наличия конструктора и селекторов у типа C при помощи математических формул выглядит следующим образом:

$$c = \text{constructor } C, \tag{1.12}$$

$$s_1, \ldots, s_n = \text{selectors } C. \tag{1.13}$$

Вполне понятно, что конструктор — это функция, имеющая тип $c : (C_1 \times \ldots C_n) \to C$, или при записи в каррированном виде: $c : (C_1 \to \ldots (C_n \to C) \ldots)$. Таким образом, смысл конструктора полностью определяется следующей формулой:

$$\forall\, c_i \in C_i, i = \overline{1,n} : c\ c_1 \ldots c_n =< c_1, \ldots, c_n > . \tag{1.14}$$

В свою очередь, каждый селектор $s_i, i = \overline{1,n}$ имеет тип $s_i : C \to C_i$. Все селекторы связаны с конструктором простейшей взаимосвязью:

$$\forall\, x \in C : \text{constructor } C\ \ (s_1 x) \ldots (s_n x) = x. \tag{1.15}$$

Либо в симметричной записи:

$$s_i\ (\text{constructor } C\ \ c_1 \ldots c_n) = c_i. \tag{1.16}$$

Таким образом, имея операцию декартова произведения и связанные с ней функции для конструирования и выбора, можно описывать сложные типы данных на основе простых, создавать для них значения из значений базовых типов, а также выбирать любое значение базового типа из декартова произведения.

Размеченное объединение. Размеченное объединение, так же как и декартово произведение в данном случае, являет собой обычное объединение множеств (типы — это, по сути, множества), дополненное двумя наборами функций. То есть размеченное объединение определяется следующим образом.

Если $C_1, \ldots, C_n$ — это некоторые типы, а C — это тип, состоящий из объединения типов $C_1, \ldots, C_n$, при условии выполнения «размеченности», то C называется размеченным объединением типов $C_1, \ldots, C_n$. Обозначается этот факт как $C = C_1 + \ldots + C_n$. Условие размеченности обозначает, что если из C взять какой-нибудь элемент c_i, то однозначно определяется базовый тип этого элемента C_i.

Размеченность типа C определяется при помощи набора предикатов $P_1, \ldots, P_n$ таких, что:

$$\forall\, (x \in C)\ \wedge\ (x \in C_i) \Rightarrow (P_i(x))\ \wedge\ (\forall\, j \neq i,\ \overline{P_j(x)}). \tag{1.17}$$

Размеченное объединение гарантирует наличие таких предикатов. Этот факт указывается записью: $P_1, \ldots, P_n = \text{predicates } C$. Присутствие предикатов по сути позволяет всем значениям из типа C не терять своей идентичности и «помнить» о своем первоначальном типе C_i. Тем самым достигается то, что тип C как бы остается разделенным на части. Каждая часть — это исходный тип.

В связи с этим, для того чтобы выделить среди множества значений типа C определенную часть, имеется набор функций, обозначаемых как $N_1, \ldots, N_n = \text{parts}\, C$. Каждая такая функция в применении ко всему множеству значений типа C возвращает только множество значений соответствующего ей типа C_i.

Как видно, в представленном метаязыке используются две операции для конструирования типов: $(\times)$ и $(+)$. К этим операциям прилагаются наборы функций для доступа к различным элементам конструируемых типов, для определения их свойств (принадлежности), для разделения типов. Эти наборы функций определяются при помощи служебных слов constructor, selectors, predicates и parts.

Примеры определения типов данных

Далее рассматриваются несколько примеров определения новых типов данных. для каждого примера по возможности показывается вариант создания шаблонной функции для обработки, на основании которой можно построить любые другие функции. Этот процесс рассмотрен в разделе 1.3 в качестве одной из типовых задач, решаемых методами функционального программирования.

Пример 1.5. Формальное определение типа $List(A)$

$List(A) = NIL + (A \times List(A))$
$prefix = \text{constructor}\, List(A)$
$head, tail = \text{selectors}\, List(A)$
$isNil, isNonNil = \text{predicates}\, List(A)$
$nil, nonNil = \text{parts}\, List(A)$

Данное определение типа $List(A)$ по своей сути является определением индуктивного множества некоторых сложных значений, построенных на основе значений базового (атомарного) типа A. Такие индуктивные множества позволяют в том числе и проводить доказательство свойств функций, созданных для обработки значений этих типов. Рассмотрение этой задачи приводится в разделе 1.5.

Для такого определения становится простым построение типового внешнего вида функции, обрабатывающей значения типа $List(A)$. Типовой внешний вид можно определить при помощи шаблонной функции высшего порядка, которая принимает на вход другие функции, которые реализуют логику обработки значений. А сам перебор атомарных значений внутри типа $List(A)$ производится типовой функцией.

Каждая функция для обработки значений типа $List(A)$ должна содержать как минимум два клоза[10]. Первый обрабатывает NIL, второй — $nonNIL$ соответ-

[10] Под клозом понимается одно выражение в записи определения функции. Точное определение дано на стр. 120 — см. определение 2.3.

ственно. Этим двум частям типа $List(A)$ в языке Haskell обычно соответствуют образцы[11] `[]` и `(x:xs)`. Два клоза можно объединить в один с использованием технологии охраны[12]. В теле второго клоза (или второго выражения охраны) обработка элемента `xs` (или `tail l`) выполняется той же самой функцией.

В следующем примере исходного кода показан модуль, содержащий определения функций на языке Haskell для выполнения определенных действий над списками. Все эти функции являются аналогами существующих в стандартном модуле `Prelude`, однако основаны на рассмотренной технологии автоматического построения шаблонной функции по определению типа. Шаблонная функция описана в модуле первой и называется `lstTemplate`.

Листинг 1.1. Модуль с описанием шаблонной функции для обработки списков

```
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--                                                                           --
--  Модуль TEMPLATES - примеры построения функций для обработки списков на   --
--  основе шаблонного каркаса типовых функций.                               --
--                                                                           --
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
module Templates
  (lstLength, lstSumm, lstProduct, lstReverse, lstMap)
where

-------------------------------------------------------------------------------
--[ СЛУЖЕБНЫЕ ФУНКЦИИ ]--------------------------------------------------------

-------------------------------------------------------------------------------
-- Главная функция, определяющая шаблонный каркас для любой другой функции,
-- которая предназначена для обработки списка (при этом полагается, что такая
-- функция принимает только один аргумент - список для обработки.
--
-- Входные параметры:
--   f1 - функция для обработки пустого списка.
```

[11] Определение того, что понимается под словом «образец», приведено на стр. 121 — см. определение 2.4.

[12] Охрана, или охранное выражение, — это условное выражение, которое как бы «охраняет» часть кода от исполнения при условии своей ложности. Описание того, что представляет собой охрана и как это понятие используется в программировании, приведено на стр. 143 в разделе 2.4.

```
--   f2 - функция для сцепки результатов обработки головы и остатка непустого
--        списка.
--   f3 - функция для обработки головы непустого списка.
--   f4 - функция для обработки результата рекурсивного вызова для остатка
--        непустого списка.
--   f5 - функция для предварительной обработки остатка непустого списка перед
--        рекурсивным вызовом.
--   l  - список для обработки.
--
-- Возвращаемое значение:
--   Список, обработанный в соответствии с логикой, установленной функциями
--   f1, f2, f3, f4 и f5.

lstTemplate :: ([a] -> b) -> (c -> d -> b) -> (e -> c) ->
               (b -> d) -> ([e] -> [e]) -> [e] -> b
lstTemplate f1 _ _ _ _ []          = f1 []
lstTemplate f1 f2 f3 f4 f5 (x:xs)  = f2 (f3 x)
                                        (f4 (lstTemplate f1 f2 f3 f4 f5 (f5 xs)))

-------------------------------------------------------------------------------
-- Функция для возвращения заданной константы, независимо от второго аргумента.
-- Предназначена для определения констант. Аналог функции const из стандартного
-- модуля Prelude. Аналог комбинатора K.
--
-- Входные параметры:
--   n - то, что необходимо возвратить в качестве результата.
--   _ - то, что теряется.
--
-- Возвращаемое значение:
--   Значение входного параметра n.

lstConstant :: a -> b -> a
lstConstant n _ = n

-------------------------------------------------------------------------------
-- Функция для возвращения своего аргумента. Аналог функции id из стандартного
-- модуля Prelude. Аналог комбинатора I.
--
-- Входные параметры:
--   any - то, что необходимо возвратить в качестве результата.
--
-- Возвращаемое значение:
--   Значение входного параметра any.
```

```
lstId :: a -> a
lstId any = any

-------------------------------------------------------------------------------
-- Функция для перемены местами операндов у заданной операции. Аналог функции
-- flip из стандартного модуля Prelude. Аналог комбинатора C.
--
-- Входные параметры:
--   op - операция (бинарная функция), у которой необходимо поменять местами
--        операнды (входные аргументы).
--   x  - первый операнд операции op.
--   y  - второй операнд операции op.
--
-- Возвращаемое значение:
--   Значение операции op, вычисленное с аргументами, поменянными местами.

lstSwap :: (a -> b -> c) -> b -> a -> c
lstSwap op x y = op y x

-------------------------------------------------------------------------------
-- Функция для заключения своего аргумента в список. Аналог функции return,
-- реализованной для монады [].
--
-- Входные параметры:
--   x - то, что необходимо заключить в список.
--
-- Возвращаемое значение:
--   Входной аргумент x, заключённый в список.

lstList :: a -> [a]
lstList x = [x]

-------------------------------------------------------------------------------
--[ ОСНОВНЫЕ ФУНКЦИИ ]---------------------------------------------------------

-------------------------------------------------------------------------------
-- Функция для вычисления длины списка. Аналог функции length из стандартного
-- модуля Prelude.

lstLength :: [a] -> Integer
lstLength = lstTemplate (lstConstant 0) (+) (lstConstant 1) lstId lstId

-------------------------------------------------------------------------------
-- Функция для вычисления суммы элементов списка, состоящего из чисел. Аналог
```

```
-- функции sum из стандартного модуля Prelude.

lstSumm :: [Integer] -> Integer
lstSumm = lstTemplate (lstConstant 0) (+) lstId lstId lstId

-------------------------------------------------------------------------------
-- Функция для вычисления произведения элементов списка, состоящего из чисел.
-- Аналог функции product из стандартного модуля Prelude.

lstProduct :: [Integer] -> Integer
lstProduct = lstTemplate (lstConstant 1) (*) lstId lstId lstId

-------------------------------------------------------------------------------
-- Функция для обращения списка. Первый элемент становится последним, второй -
-- предпоследним и т. д. Аналог функции reverse из стандартного модуля Prelude.

lstReverse :: [a] -> [a]
lstReverse = lstTemplate lstId (lstSwap (++)) lstList lstId lstId

-------------------------------------------------------------------------------
-- Функция для применения к каждому элементу заданного списка определённой
-- функции. Аналог функции map из стандартного модуля Prelude.

lstMap :: (a -> b) -> [a] -> [b]
lstMap f = lstTemplate lstId (:) f lstId lstId

--[ КОНЕЦ МОДУЛЯ ]-------------------------------------------------------------
```

В этот модуль включен набор определений служебных функций, большая часть из которых также представлена в стандартном модуле `Prelude`, — `lstConstant`, `lstId`, `lstSwap` и `lstList`. Они приведены лишь Для того чтобы показать возможности языка Haskell и приучить читателя уже на данном этапе к синтаксису этого языка программирования.

Необходимо отметить, что в исходном тексте программ на языке Haskell любая подстрока, начинающаяся с символов «`--`», считается комментарием. Многострочные комментарии можно организовывать при помощи заключения их между последовательностями символов «`{-`» и «`-}`».

В качестве функций для обработки списков представлены:

1) функция `lstLength` — возвращает длину заданного списка;

2) функция `lstSumm` — возвращает сумму элементов списка;

3) функция `lstProduct` — возвращает произведение элементов списка;

4) функция `lstReverse` — обращает список;

5) функция `lstMap` — применяет к каждому элементу списка заданную функцию и возвращает список, полученный из результатов выполнения этой функции.

Последняя функция `lstMap` является примером того, что представленный шаблон типовых функций можно использовать и для определения функций с более чем одним аргументом.

На примере представленных функций видно, что в типовом шаблоне аргументы `f4` и `f5` практически никогда не используются. Во всех пяти примерах определений конкретных функций в качестве данных аргументов использовалась константная функция `lstId`, поэтому можно построить «облегченную» версию типового шаблона:

```
lstTemplate_ f1 f2 f3 []     = f1 []
lstTemplate_ f1 f2 f3 (x:xs) = f2 (f3 x) (lstTemplate_ f1 f2 f3 xs)
```

Если в этом определении ответственность за обработку пустого списка `[]` возложить на функцию `f2` (что вполне логично и не нарушает общности), то определение типового шаблона можно еще больше сократить (заодно и переименовав входные параметры):

```
lstTemplate_ f1 f2 []     = f2 []
lstTemplate_ f1 f2 (x:xs) = f1 (f2 (x:xs)) (lstTemplate_ f1 f2 xs)
```

Данное определение очень сильно напоминает функцию правой свертки `foldr`, описанную в стандартном модуле `Prelude`. Определение правой свертки выглядит так:

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

На первый взгляд может показаться, что общего в этих двух определениях немного, однако функция `foldr` написана с использованием накапливающего параметра (см. раздел 2.4), поэтому для нее задается только одна функция для обработки значений внутри списка. А в целом эти функции очень похожи.

В связи с этим остается отметить, что в стандартном модуле `Prelude` определены именно такие шаблонные функции для обработки списков в виде функций для левой и правой свертки разных видов (для каждого типа свертки имеется несколько функций, отличающихся некоторыми незначительными нюансами, — см. приложение C). То есть описанная здесь технология имеет самое непосредственное практическое применение.

Пример 1.6. Формальное определение типа $ListStructure(A)$

$ListStructure(A) = A + List(ListStructure(A))$;
$prefix = \text{constructor}\, ListStructure(A)$;
$head, tail = \text{selectors}\, ListStructure(A)$;
$isAtom, isNonAtom = \text{predicates}\, ListStructure(A)$;
$atom, nonAtom = \text{parts}\, ListStructure(A)$.

Функции, обрабатывающие данные типа $ListStructure(A)$, должны иметь, по крайней мере, следующие клозы:

```
f1 [] = ...
f1 a = if (isAtom a) then ...
                     else f2 a

f2 (x:xs) = if (isAtom x) then ...
                          else ...
```

Читателю предлагается самостоятельно разработать шаблонную функцию для обработки списочных структур (тип $ListStructure(A)$), а также конкретные функции для обработки таких данных, основанные на общей шаблонной функции.

Пример 1.7. Формальное определение деревьев с помеченными вершинами

Далее представлено формальное определение типа $Tree(A)$, представляющего дерево с помеченными вершинами, а также вспомогательного типа $Forest(A)$, представляющего лес (множество деревьев) с помеченными вершинами.

Данные типы можно использовать для представления иерархических структур данных, для которых не важен тип связи между родительскими и до-

черними элементами в иерархии. Для таких данных полагается важной только информация о вершине (метка), которая имеет базовый тип A.

$Tree(A) = A \times Forest(A)$;
$Forest(A) = List(Tree(A))$;
$node = \text{constructor}\, Tree(A)$;
$root, children = \text{selectors}\, Tree(A)$.

Как видно, вспомогательный тип $Forest(A)$ — это список (по своей сути, идентификатор `Forest` является просто синонимом, для укорачивания записи наименования типа), то есть на него распространяется то же самое определение, что и в примере 1.5, поэтому конструктор, селекторы, предикаты и части для этого типа не указаны. Предикаты и части для типа $Tree(A)$ не указаны по причине того, что в определении этого типа не используется операция размеченного объединения.

Таким образом, до любого элемента, хранящегося в таком дереве, можно добраться (найти) исключительно при помощи перечисленных функций — селекторов для самого типа $Tree(A)$ и с помощью селекторов и частей для типа $Forest(A)$. Реализовав конструктор `tree` и эти функции в конкретном применении (например, для языка Haskell), уже на их основе создаются описания прочих функций, работающих с деревьями.

Например, начать реализацию определения этого типа на языке Haskell можно следующим образом:

```
data Tree a = Node (a, [Tree a])

root (Node (a, _)) = a

children (Node (_, c)) = c
```

В этом примере показано определение типа данных `Tree a`, при этом проведена оптимизация кода и убран вспомогательный тип, который соответствовал бы определению $Forest(A)$ в теоретических выкладках. Конструктор типа `Node` используется для создания одной вершины со всеми потомками. Функции `root` и `children`, получающие на вход аргумент типа `Tree a`, являются селекторами этого типа.

Как видно, описание типов и базовых функций для их обработки на языке Haskell является делом несложным — метаязык Хоара переводится в синтаксис языка Haskell практически один в один.

Пример 1.8. Формальное определение деревьев с помеченными вершинами и дугами

Иногда тип, который представляет собой дерево, содержащее информацию в своих узлах, недостаточен для решения задач, стоящих перед разработчиком программного обеспечения. Такое случается, когда тип связи между родительскими и дочерними узлами в дереве важен. Для этих целей можно использовать дерево с помеченными вершинами и дугами.

Метка на вершине хранит некоторую информацию, связанную с этой вершиной. Пусть эта информация имеет базовый тип A. Метка на дуге по существу хранит тип этой дуги, который зависит от решаемой задачи. Пусть такая метка имеет базовый тип B. Тогда тип $MTree(A, B)$ можно определить следующим образом:

$MTree(A, B) = A \times MForest(A, B)$;
$MForest(A, B) = List(MArc(A, B))$;
$MArc(A, B) = B \times MTree(A, B)$;
$node = \text{constructor}\, MTree(A, B)$;
$mRoot, mChildren = \text{selectors}\, MTree(A, B)$;
$arc = \text{constructor}\, MArc(A, B)$;
$mArc, mTree = \text{selectors}\, MArc(A, B)$.

Абсолютно таким же образом, как и тип $Tree(A)$, определяется тип $MTree(A, B)$. Здесь конечно же необходимо иметь больше вспомогательных типов, так как дерево содержит внутри себя больше информации. Так, вспомогательный тип $MForest(A, B)$ является списком помеченных дуг, выходящих из вершины дерева, связанной со значением этого типа. В свою очередь, вспомогательный тип $MArc(A, B)$ является описанием дуги, на которой стоит пометка типа B.

Для типов $MTree(A, B)$ и $MArc(A, B)$ приведены конструкторы и селекторы (опять же, предикатов и частей для этих типов нет, так как размеченное объединение при их конструировании не используется). Для типа $MForest(A, B)$,

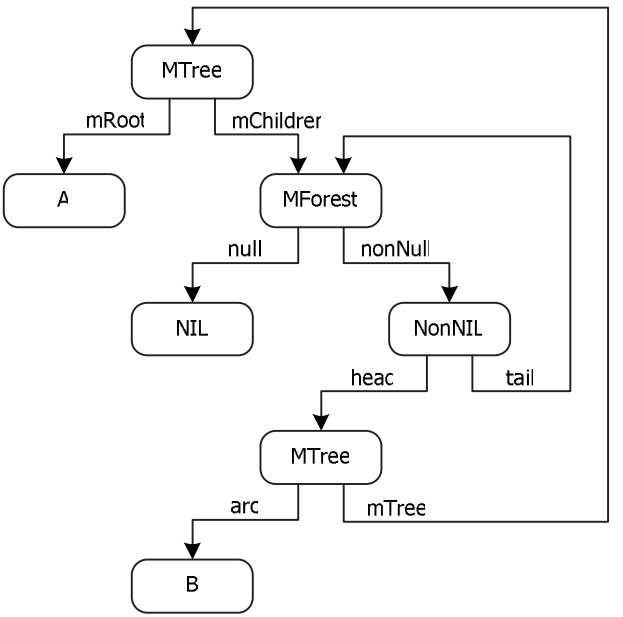

Рис. 1.4. Граф для представления типа $Tree(A, B)$

который является синонимом списка с элементами заданного типа, не приводятся шаблонные функции (для списков эти функции приведены в примере 1.5).

Способ обращения к частям описанного типа $MTree(A, B)$ схематично показан на рис. 1.4:

Таким образом, представленный набор базовых функций для обработки значений типа $MTree(A, B)$ является достаточным для обработки любого значения этого типа. Все остальные функции должны конструироваться на основе представленных.

Далее приводится определение шаблонных функций, которые позволяют обрабатывать значения типа $MTree(A, B)$ для любых целей. Все остальные конкретные функции для обработки деревьев с помеченными вершинами и дугами можно выразить через представленные шаблонные функции.

Для описания этих шаблонных функций, обрабатывающих структуры данных $MTree(A, B)$, необходимо ввести несколько дополнительных понятий и обозначений. Это делается для простоты описания, так как писать каждый раз сложный

конструктор типа с передачей ему аргументов слишком сильно загромождает и без того непростое определение.

Пусть начальная вершина дерева с помеченными вершинами и дугами, голова списка $MForest$ и вершина $MTree$, выходящая из $MArc$, обозначаются как S_0, S_1 и S_2 соответственно. Для обработки этих переменных необходимы три функции — `f0`, `f1` и `f2`, причем `f0` — это начальная функция, а две последние — рекурсивные.

Конструирование функции `f0` выглядит просто: у этой функции один параметр `tree`, который соответствует начальной вершине S_0. Две другие функции сконструировать несколько сложнее.

Функция `f1` получает следующие параметры:

1) `a` — метка текущей вершины;
2) `k` — параметр, содержащий результат обработки просмотренной части дерева;
3) `(x:xs)` — лес, который необходимо обработать в этой функции.

Определение этой функции может выглядеть следующим образом:

```
f1 a k []     = g1 a k
f1 a k (x:xs) = f1 a (g2 (f2 a arc children k) a arc k) xs
  where arc      = mArc x
        children = mTree xs
```

Как видно, эта функция организует режим просмотра дерева «сначала в глубину». При этом функции `g1` и `g2` являются вспомогательными, которые необходимы для изменения параметра `k` согласно логике задачи.

В свою очередь, функция `f2` получает следующие параметры (и это уже должно быть ясно из ее вызова во втором клозе функции `f1`):

1) `a` — метка текущей вершины;
2) `b` — метка текущей дуги;
3) `k` — результат обработки просмотренной части дерева;
4) `tree` — поддерево для последующей обработки.

Определение этой функции выглядит так:

```
f2 a b k tree = f1 (mRoot tree) (g3 a b k) (mChildren tree)
```

Опять же функция `g3` зависит от конкретных целей, поставленных перед разработчиком. Она необходима для получения очередного значения параметра `k`.

Теперь можно сконструировать и общий вид функции `f0`:

```
f0 tree = f1 (mRoot tree) k (mChildren tree)
```

В этом определении имеется неопределенная переменная `k`, которая должна быть означена в конкретном определении. Такая означенность просто-напросто полагает задание начального параметра переменной `k`.

Пример 1.9. Бинарное дерево и его обработка

Для более глубокого закрепления методики конструирования динамических структур данных и функций для их обработки (в том числе и шаблонных функций) можно рассмотреть конкретную реализацию работы с бинарными деревьями. Здесь под бинарным деревом имеется в виду дерево с помеченными вершинами, при этом из каждой вершины может выходить только ноль или две ветви. В случае, если из вершины не выходит ветвей, она называется листьевой вершиной (листом дерева).

Формально определить такой тип можно при помощи следующей формулы:

$$BTree(A) = Empty + A \times BTree(A) \times BTree(A).$$

Реализация примера будет проходить на языке Haskell, однако без детального описания используемого синтаксиса, так как сам синтаксис языка описывается в главе 2 и последующих. Однако записи определений типа $BTree(A)$ и функций для работы с ним настолько похожи на математические формулы, что понять их не составит особого труда.

Пусть тип $BTree(A)$, описывающий бинарное дерево с помеченными вершинами, определен на языке Haskell следующим образом:

```
data BTree a = Empty | Node (a, BTree a, BTree a)
```

В данной записи символ (`|`) можно понимать как размеченное объединение, а символ (`,`) — как декартово произведение. На самом деле обе эти операции из метаязыка Хоара могут проецироваться в любые иные операции

в конкретных реализациях, причем такая проекция определяется не языком программирования, а текущей задачей. Так, если в данном примере декартово произведение проецируется в символ (`,`), то в реализации списка на языке Haskell декартово произведение проецируется на операцию префиксации — (`:`).

Приведенную выше запись определения типа `BTree a` на языке Haskell можно прочитать как «Тип данных `BTree`, содержащий внутри себя значение некоторого типа `a`, являет собой пустое дерево `Empty` или (как раз значение символа (`|`) — или) узел (`Node`), при этом внутри узла содержатся значение типа `a` и два поддерева того же типа `BTree`».

Пусть для обработки значений этого типа описаны конструкторы, селекторы и предикаты:

1) `empty` — конструктор пустого дерева;
2) `node` — конструктор узловой вершины;
3) `leaf` — вспомогательный конструктор для листьевой вершины (такая вершина, оба потомка которой — пустые деревья);
4) `isEmpty` — предикат для пустого дерева;
5) `isNode` — предикат для узловой вершины;
6) `getRootValue` — селектор для выбора значения (метки), хранящегося внутри вершины (не важно — листьевой или узловой);
7) `getLeftBranch` — селектор для выбора левого поддерева (только для узловых вершин);
8) `getRightBranch` — селектор для выбора правого поддерева (только для узловых вершин).

Теперь остается написать пару функций для обработки таких бинарных деревьев в качестве примера.

Функция `insertBTree` предназначена для вставки заданного элемента внутрь дерева, при этом считается, что вставку необходимо проводить таким образом, чтобы вставляемый элемент балансировал само дерево (левее него должны быть только меньшие элементы, а правее — только большие).

Определение этой функции на языке Haskell выглядит так:

```
insertBTree :: Ord a => a -> BTree a -> BTree a
insertBTree x tree | isEmpty tree = leaf x
                   | x < root     = node root (insertBTree x left) right
                   | x > root     = node root left (insertBTree x right)
                   | otherwise    = tree
  where root  = getRootValue tree
        left  = getLeftBranch tree
        right = getRightBranch tree
```

Данная функция позволяет создавать сбалансированные бинарные деревья на базе типа `BTree a`, при этом она работает только с такими атомарными типами (то есть теми, что хранятся внутри дерева), значения которых являются сравнимыми величинами (на это указывает директива (`Ord a =>`) в начале описания типа функции `insertBTree`). При всей внешней привлекательности с точки зрения простоты и малого количества исходного кода, функция обладает немалым недостатком — при вставке нового элемента в дерево происходит полное воссоздание первоначального дерева с вставленным в него элементом. Как было уже показано в разделе 1.2, это основной недостаток всех функциональных языков программирования, в которых отсутствует деструктивное присваивание. В современных функциональных языках этот недостаток практически полностью нивелируется наличием эффективных сборщиков мусора.

Другая функция, которая может понадобиться для работы со сбалансированными бинарными деревьями, в какой-то мере имеет противоположный функции `insertBTree` эффект. Она необходима для поиска в заданном дереве некоторого поддерева, вершина которого соответствует заданному критерию поиска.

Определение такой функции выглядит так:

```
accessBTree :: Ord a => a -> BTree a -> BTree a
accessBTree x tree | isEmpty tree = Empty
                   | x < root     = accessBTree x (left)
                   | x > root     = accessBTree x (right)
                   | otherwise    = tree
  where root  = getRootValue tree
        left  = getLeftBranch tree
        right = getRightBranch tree
```