

# Функциональное программирование на языке Haskell

Лучшая  
практическая реализация  
функциональной парадигмы  
программирования



**На CD содержится:**

- трансляторы Haskell
- библиотеки и утилиты
- примеры программ

Душкин Р. В.

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. Изучение прикладных основ показано на примере языка Haskell, на сегодняшний день являющегося самым мощным и развитым инструментом функционального программирования.

Издание можно использовать в качестве учебника по функциональному программированию, и в качестве самостоятельного учебного пособия по смежным дисциплинам, в первую очередь по комбинаторной логике и  $\lambda$ -исчислению.

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

К книге прилагается 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.

**ISBN 5-94074-335-8**

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

# Оглавление

<b>Содержание</b>	<b>5</b>
<b>Введение</b>	<b>19</b>
<b>1 Основы функционального программирования</b>	<b>27</b>
1.1 История функционального программирования . . . . .	28
1.2 Основные свойства функциональных языков . . . . .	44
1.3 Типовые задачи, решаемые методами функционального программирования . . . . .	58
1.4 Конструирование функций . . . . .	69
1.5 Доказательство свойств функций . . . . .	86
<b>2 Базовые принципы языка Haskell</b>	<b>99</b>
2.1 Списки — основа функциональных языков . . . . .	100
2.2 Функции как описания процессов вычисления . . . . .	118
2.3 Типизация данных и функций . . . . .	131
2.4 Элементы программирования . . . . .	142
2.5 Модули и абстрактные типы данных . . . . .	153
<b>3 Классы и их экземпляры</b>	<b>164</b>
3.1 Параметрический полиморфизм данных . . . . .	165
3.2 Классы в языке Haskell как способ абстракции действительности .	170
3.3 Наследование и реализация . . . . .	180
3.4 Стандартные классы языка Haskell . . . . .	192
3.5 Сравнение с другими языками программирования . . . . .	207

<b>4</b>	<b>Монады — последовательное выполнение действий в функциональной парадигме</b>	<b>213</b>
4.1	Монада как тип-контейнер . . . . .	214
4.2	Последовательное выполнение действий . . . . .	222
4.3	Операции ввода/вывода в языке Haskell . . . . .	239
4.4	Стандартные монады языка Haskell . . . . .	252
4.5	Разработка собственных монад . . . . .	262
<b>5</b>	<b>Комбинаторная логика и <math>\lambda</math>-исчисление</b>	<b>273</b>
5.1	Основы комбинаторной логики . . . . .	274
5.2	Абстракция функций как вычислительных процессов . . . . .	288
5.3	$\lambda$ -исчисление как теоретическая основа функционального программирования . . . . .	298
5.4	Кодирование данных в $\lambda$ -исчислении . . . . .	307
5.5	Редукция и вычисления в функциональных языках . . . . .	315
<b>6</b>	<b>Трансляторы программ</b>	<b>331</b>
6.1	Математическая лингвистика . . . . .	331
6.2	Краткое введение в теорию построения трансляторов . . . . .	346
6.3	Реализация трансляторов на языке Haskell . . . . .	360
6.4	Библиотеки для создания трансляторов . . . . .	372
6.5	Частичные вычисления, трансформация программ и суперкомпиляция . . . . .	381
<b>7</b>	<b>Функциональное программирование и искусственный интеллект</b>	<b>395</b>
7.1	Основные задачи искусственного интеллекта . . . . .	396
7.2	Нечеткая математика и функциональное программирование . . . . .	407
7.3	Логический вывод на знаниях . . . . .	429
7.4	Общение с компьютером на естественном языке . . . . .	443
7.5	Перспективы функционального программирования . . . . .	454
	<b>Заключение</b>	<b>463</b>
	<b>Ответы на задачи для самостоятельного решения</b>	<b>465</b>
	Решения задач из главы 1 . . . . .	465

Решения задач из главы 2 . . . . .	467
Решения задач из главы 3 . . . . .	474
Решения задач из главы 4 . . . . .	477
Решения задач из главы 5 . . . . .	483
Решения задач из главы 6 . . . . .	488
<b>A Функциональные языки программирования и Интернет-ресурсы по функциональному программированию</b>	<b>496</b>
Функциональные языки программирования . . . . .	496
Русские Интернет-ресурсы . . . . .	502
Иностранные Интернет-ресурсы . . . . .	503
<b>B Опции различных сред разработки на языке Haskell</b>	<b>506</b>
Интегрированная среда разработки HUGS 98 . . . . .	506
Компилятор GHC . . . . .	510
Компилятор NHC . . . . .	523
Компилятор компиляторов Happy . . . . .	528
<b>C Описание стандартного модуля Prelude</b>	<b>531</b>
Функции . . . . .	531
Описание некоторых операторов языка Haskell . . . . .	586
<b>D Краткий словарь терминов из области функционального программирования</b>	<b>589</b>
<b>Литература</b>	<b>599</b>
Общая литература по функциональному программированию . . . .	599
Книги, руководства и статьи по языку Haskell . . . . .	601
Комбинаторная логика и $\lambda$ -исчисление . . . . .	602
Математическая лингвистика и теория построения трансляторов .	603
Искусственный интеллект . . . . .	604

# Содержание

<b>Введение</b> .....	<b>19</b>
Краткая биография автора .....	21
О пользовании книгой .....	22
Состав и структура представления информации .....	24
Благодарности .....	26
Контактная информация .....	26
 <b>1 Основы функционального программирования</b> .....	<b>27</b>
<i>В этой главе рассматриваются основополагающие принципы функционального программирования как отдельного направления в математической науке и технологии создания программного обеспечения. Приводится история развития функционального программирования, описываются предпосылки его развития. В главе рассматривается больше теоретического материала, нежели практического, поэтому пока изложение ведется без рассмотрения синтаксиса языка Haskell. Однако для приведения примеров используется именно этот язык наряду с математическими формулами. Изложение знаний о языке Haskell начинается с главы 2.</i>	
 <b>1.1 История функционального программирования</b> .....	<b>28</b>
<i>Краткая история развития теории функционального программирования в мире и в России. Разработка функциональных языков для подтверждения теоретических выкладок. Проблемы, с которыми столкнулись исследователи при разработке функциональных языков программирования. Современное состояние теории функционального программирования. Стандарт Haskell 98 как результат унификации и стандартизации процессов развития функционального программирования.</i>	
 Предпосылки создания функционального программирования .....	34
История языка Haskell .....	38

Заключительные слова .....	42
<b>1.2 Основные свойства функциональных языков .....</b>	<b>44</b>
<i>Описание основных свойств функциональных языков программирования. Краткость и простота, строгая типизация, модульность, функциональные значения и объекты, чистота и отложенные вычисления. Понимание свойств функциональных языков на примере языка Haskell.</i>	
Краткость и простота .....	44
Строгая типизация .....	48
Модульность .....	50
Функции — это значения и объекты вычисления .....	51
Чистота (отсутствие побочных эффектов и детерминированность) .....	53
Отложенные (ленивые) вычисления .....	55
<b>1.3 Типовые задачи, решаемые методами ФП .....</b>	<b>58</b>
<i>Краткое описание семи типовых задач, которые решаются методами функционального программирования. Получение остаточной процедуры. Построение математического описания функций. Определение формальной семантики языка программирования. Описание динамических структур данных. Автоматическое построение «значительной» части программы по описанию структур данных, которые обрабатываются создаваемой программой. Доказательство наличия некоторого свойства программы. Эквивалентная трансформация программ.</i>	
Получение остаточной процедуры .....	60
Построение математического описания функций .....	62
Определение формальной семантики языка программирования .....	64
Описание динамических структур данных .....	64
Автоматическое построение функций по описанию структур данных .....	66
Доказательство наличия некоторого свойства программы .....	67
Эквивалентная трансформация программ .....	68
<b>1.4 Конструирование функций .....</b>	<b>69</b>
<i>Описание метода конструирования функций, предложенного Ч. Хоаром (синтаксически ориентированное конструирование). Метаязык для конструирования функций. Примеры определения типов и функций для обработки этих типов.</i>	
Декартово произведение .....	70
Размеченное объединение .....	71



Примеры определения типов данных .....	72
<b>1.5 Доказательство свойств функций .....</b>	<b>86</b>
<i>Задача доказательства свойств функций. Описание процесса доказательства свойств функций в зависимости от типа области определения функций. Примеры доказательства свойств функций.</i>	
Область определения $D$ — линейно-упорядоченное множество .....	88
Множество $D$ определяется как индуктивный класс .....	89
Рассмотрение некоторых примеров доказательства свойств функций .....	91
Вопросы для самоконтроля .....	95
Задачи для самостоятельного решения .....	97
<b>2 Базовые принципы языка Haskell .....</b>	<b>99</b>
<i>Глава посвящена введению в основные положения языка Haskell, приводится описание синтаксиса для решения основных задач по созданию отдельных функций и законченных модулей. Рассматриваются базовые объекты «список» и «функция» для изучения в рамках функционального программирования, а также их реализация на языке Haskell.</i>	
<b>2.1 Списки — основа функциональных языков .....</b>	<b>100</b>
<i>Понятие списка в функциональном программировании. Списки как основная структура для работы с функциональными языками. Базисные операции для работы со списками. Списки и списочные структуры. Программная реализация списков в функциональных языках. Списки в языке Haskell. Генераторы списков и математические последовательности. Бесконечные списки и другие структуры данных. Кorteжи.</i>	
Проекция списков в язык Haskell .....	101
Несколько слов о программной реализации .....	104
Примеры .....	106
Определители списков и математические последовательности .....	110
Кorteжи .....	117
<b>2.2 Списки — основа функциональных языков .....</b>	<b>118</b>
<i>Функция — основной объект изучения функционального программирования. Соглашения по именованию объектов в языке Haskell. Описание и определение функций на языке Haskell. Образцы и клозы. Передача параметров и возвращение значений функциями. Инфиксный способ записи функций. Функция как объект</i>	

для передачи в другие функции. Программа на языке *Haskell* — функция, описывающая процесс вычисления.

<i>Соглашения по именованию</i> .....	118
<i>Общий вид определения функции</i> .....	119
<i>Образцы и клозы</i> .....	119
<i>Вызовы функций</i> .....	125
<i>Использование <math>\lambda</math>-исчисления</i> .....	126
<i>Инфиксный способ записи функций</i> .....	127
<i>Несколько слов о функциях высшего порядка</i> .....	131
 2.3 Списки — основа функциональных языков .....	131
<i>Структуры и типы данных. Типы функций. Каррированные и некаррированные функции. Язык Haskell и его механизмы для организации каррированных и некаррированных функций. Описание типов функций на языке Haskell. Частичное применение. Ленивые (отложенные) вычисления на языке Haskell.</i>	
<i>Структуры данных и их типы</i> .....	132
<i>Синонимы типов</i> .....	136
<i>Типы функций в функциональных языках</i> .....	137
<i>Полиморфные типы</i> .....	140
 2.4 Списки — основа функциональных языков .....	142
<i>Охраняющие выражения и конструкции. Локальные переменные для оптимизации кода на функциональном языке и на языке Haskell. Использование накапливающего параметра (аккумулятора) для оптимизации процесса вычислений. Принципы построения определений функций с накапливающим параметром. Головная и хвостовая рекурсия.</i>	
<i>Охрана</i> .....	143
<i>Ветвление алгоритма</i> .....	145
<i>Локальные переменные</i> .....	146
<i>Двумерный синтаксис</i> .....	149
<i>Накапливающий параметр — аккумулятор</i> .....	150
<i>Принципы построения определений с накапливающим параметром</i> .....	152
 2.5 Списки — основа функциональных языков .....	153
<i>Модули как способы структуризации и организации программ на языке Haskell. Импорт и экспорт данных при помощи модулей. Скрытие данных. Абстрактные типы данных и интерфейсы. Иные аспекты использования модулей.</i>	

Абстрактные типы данных .....	156
Другие аспекты использования модулей .....	157
Литературный код .....	158

Вопросы для самоконтроля .....	160
Задачи для самостоятельного решения .....	161

### 3 Классы и их экземпляры ..... 164

*Эта глава посвящена рассмотрению симбиоза парадигм функционального и объектно-ориентированного программирования. Большинство современных функциональных языков поддерживают механизмы и методы, разработанные в рамках объектно-ориентированного программирования, в том числе и такие базовые концепты, как «наследование», «инкапсуляция» и «полиморфизм». Не обошел своим вниманием этот аспект и язык Haskell, в котором имеются достаточные средства для программирования в объектно-ориентированном стиле.*

#### 3.1 Параметрический полиморфизм данных ..... 165

*Понятие класса и его реализации в языке Haskell. Чистый (параметрический) полиморфизм на языке Haskell. Примеры параметрического полиморфизма в императивных и функциональных языках, а также в языке Haskell.*

#### 3.2 Классы в языке Haskell как способ абстракции действительности ..... 170

*Расширенное описание понятия класса в языке Haskell. Класс как высшая абстракция данных и методов для их обработки. Методы класса — шаблоны функций для реализации обработки данных. Минимальное описание методов класса и связь методов.*

#### Модель типизации Хиндли-Милнера ..... 171

#### Определение классов ..... 176

#### 3.3 Наследование и реализация ..... 180

*Наследование классов и наследование методов. Экземпляры классов — реализация интерфейсов, предоставляемых реализуемым классом. Реализация методов для обработки данных. Класс — шаблон типа, реализация класса — тип данных.*

#### Наследование ..... 180

#### Реализация ..... 182

#### Реализация для существующих типов ..... 186

#### Сорта типов ..... 187

#### Дополнительные возможности при определении типов данных ..... 189

### 3.4 Стандартные классы языка Haskell .....192

*Краткое описание всех стандартных классов, разработанных для облегчения программирования на языке Haskell. Дерево наследования стандартных классов. Типичные способы использования стандартных классов языка Haskell. Реализация стандартных классов — типы в языке Haskell.*

Класс <i>Bounded</i> .....	192
Класс <i>Enum</i> .....	193
Класс <i>Eq</i> .....	195
Класс <i>Floating</i> .....	195
Класс <i>Fractional</i> .....	196
Класс <i>Functor</i> .....	197
Класс <i>Integral</i> .....	198
Класс <i>Ix</i> .....	199
Класс <i>Monad</i> .....	199
Класс <i>Num</i> .....	200
Класс <i>Ord</i> .....	201
Класс <i>Read</i> .....	202
Класс <i>Real</i> .....	203
Класс <i>RealFloat</i> .....	203
Класс <i>RealFrac</i> .....	204
Класс <i>Show</i> .....	206

### 3.5 Сравнение с другими языками программирования .....207

*Более или менее полное сравнение понятий «класс» и «реализация класса» в языке Haskell с объектно-ориентированными языками программирования (на примере языков C++ и Java, а также некоторых других языков). Глобальные отличия понятия «класс» в функциональных и объектно-ориентированных языках.*

Окончательные замечания .....	209
-------------------------------	-----

### Вопросы для самоконтроля .....210

### Задачи для самостоятельного решения .....211

## 4 Монады — последовательное выполнение действий в функциональной парадигме ..... 213

*Глава описывает такое незаурядное понятие, введенное в функциональной парадигме программирования, как «монада». Монады, основанные на математической теории категорий, позволяют внедрить в функциональный подход опре-*

деленные структуры для выполнения императивных действий, как, например, операции ввода/вывода, обработка исключений, хранение состояний в процессе вычислений, и многие другие действия, связанные с побочными эффектами. Вместе с тем монады позволяют обернуть императивные действия в функциональную оболочку, спрятав все от «императивного мира» внутри монады.

#### 4.1 Монада как тип-контейнер ..... 214

*Описание монады как типа-контейнера. Использование монад в функциональных языках. Свойства монадических типов. Операции связывания с передачей и без передачи результата выполнения операции на предыдущем шаге. Правила построения монад.*

##### *Определение понятия «монада» ..... 215*

##### *Нотация `do` ..... 218*

##### *Правила построения монад ..... 220*

#### 4.2 Последовательное выполнение действий ..... 222

*Действие — элемент функциональной парадигмы. Императивный код внутри функционального. Выполнение действий и возвращение результата. Сокращенный способ записи последовательности действий. Списки действий. Программирование при помощи действий.*

##### *Класс `Computations` ..... 224*

##### *Монада `State` ..... 227*

#### 4.3 Операции ввода/вывода в языке Haskell ..... 239

*Более или менее полное описание базовых операций ввода/вывода в языке Haskell. Монада `IO`. Обработка исключений. Использование файлов, каналов и обработчиков. Нарушение теоретических принципов функционального программирования в монаде `IO`.*

##### *Действия ввода/вывода ..... 240*

##### *Программирование при помощи действий ..... 244*

##### *Обработка исключений ..... 245*

##### *Файлы и потоки ..... 247*

##### *Окончательные замечания ..... 251*

#### 4.4 Стандартные монады языка Haskell ..... 252

*Подробное описание монадических типов в стандартной библиотеке языка Haskell. Назначение и применимость монадических типов. Примеры использования стандартных монадических типов (кроме списков и монады `IO`). Модуль `M Monad`. Монады `Glasgow Haskell Compiler`.*

Модуль <i>Monad</i> .....	253
Стандартные монады .....	257
4.5 Разработка собственных монад .....	262
<i>Критерии возможности и необходимости разработки собственного монадического типа. Комбинирование монадических вычислений. Преобразователи монад. Примеры преобразования.</i>	
Комбинирование монадических вычислений .....	263
Преобразователи монад .....	264
Пример с преобразователем <i>StateT</i> .....	267
Окончательные замечания .....	268
Вопросы для самоконтроля .....	269
Задачи для самостоятельного решения .....	270
<b>5 Комбинаторная логика и <math>\lambda</math>-исчисление .....</b>	<b>273</b>
<i>В главе рассматриваются основополагающие теоретические формализмы, которые стояли у истоков функционального программирования, а именно комбинаторная логика и <math>\lambda</math>-исчисление, разработанные в качестве расширений формальной логики и теории множеств в начале XX в. Приводятся самые основы этих направлений дискретной математики, достаточные для понимания сути того, что в свое время стояло за парадигмой функционального программирования.</i>	
5.1 Основы комбинаторной логики .....	274
<i>Введение в комбинаторную логику. Поверхностное описание принципов комбинаторной логики. Комбинаторы и вычисления при помощи комбинаторов. Базисы в комбинаторной логике. Использование базисных комбинаторов для выражения любых вычислительных процессов. Числа и иные математические объекты в виде комбинаторов.</i>	
Базовые комбинаторы .....	274
Комбинатор неподвижной точки .....	281
Нумералы и арифметические операции .....	283
Заключительные слова .....	286
5.2 Абстракция функций как вычислительных процессов .....	288
<i>Функция — объект математического исследования. Вычислительный процесс — функция. Описание функций как <math>\lambda</math>-выражений. Свободные и связанные иденти-</i>	

фикаторы. Применение (аппликация) значений $\kappa$ $\lambda$ -выражениям. Непрерывная точка функций и теорема о непрерывной точке.	
«Наивное» определение $\lambda$ -исчисления .....	289
Связь с комбинаторной логикой .....	292
Редукция .....	293
Тезис Черча-Тьюринга .....	294
5.3 $\lambda$ -исчисление как теоретическая основа функционального программирования .....	298
Предположение о том, что любая функция представима в виде $\lambda$ -выражения.	
Интенционал и экстенционал функций. Формальная система. Построение формальной системы для обоснования теории функционального программирования.	
Правила вывода. Соответствия между вычислениями функциональных программ и редукцией $\lambda$ -выражений.	
Построение формальной системы .....	299
Функциональное программирование как формальная система .....	303
Теорема Черча-Россера .....	305
5.4 Кодирование данных в $\lambda$ -исчислении .....	307
Механизм кодирования данных в $\lambda$ -исчислении. $\lambda$ -исчисление — достаточный формализм для представления значений истинности, упорядоченных пар, натуральных чисел, списков, а также базовых операций над этими объектами.	
Булевские значения .....	308
Упорядоченные пары .....	309
Натуральные числа .....	311
Списки .....	314
5.5 Редукция и вычисления в функциональных языках .....	315
Понятие редукции. Частичные вычисления с точки зрения редукции $\lambda$ -выражений. Различные редукционные стратегии и их свойства.	
Стратегия редукции и стратегия вычислений .....	315
Ленивая редукция .....	321
Вопросы для самоконтроля .....	327
Задачи для самостоятельного решения .....	329
6 Трансляторы программ .....	331

*В главе 6 представлено краткое описание теории построения трансляторов для интерпретации и компиляции языков программирования. Теория рассматривается на примерах, написанных на языке Haskell. Кроме того, рассматриваются способы построения парсеров, а также готовые библиотеки для синтаксического анализа. Суперкомпиляция.*

<b>6.1</b>	<b>Математическая лингвистика</b> .....	<b>331</b>
	<i>Краткое введение в математическую лингвистику. Обзор методов и принципов математической лингвистики. Классификация языков и грамматик. Конечные автоматы и контекстно-свободные языки. Грамматики типа <math>LL(k)</math>. Контекстно-зависимые грамматики.</i>	
	<i>Базовые понятия</i> .....	<i>332</i>
	<i>Расширенная нотация Бэкуса — Наура</i> .....	<i>335</i>
	<i>Классификация грамматик</i> .....	<i>337</i>
	<i>Конечные лингвистические автоматы</i> .....	<i>338</i>
	<i>Синтаксический анализ контекстно-свободных языков</i> .....	<i>343</i>
<b>6.2</b>	<b>Краткое введение в теорию построения трансляторов</b> .....	<b>346</b>
	<i>Трансляторы: определения, типы и классификация, применимость для тех или иных типов языков и грамматик. Интерпретаторы и компиляторы. Компиляторы компиляторов. Методы построения трансляторов. Автоматическое построение транслятора по грамматике языка (для ограниченного множества языков). Понятие трансформационной грамматики.</i>	
	<i>Классификация трансляторов и их типовые структуры</i> .....	<i>347</i>
	<i>Трансформационные грамматики</i> .....	<i>354</i>
	<i>Автоматическое построение анализатора для отдельных типов языков</i> .....	<i>358</i>
<b>6.3</b>	<b>Реализация трансляторов на языке Haskell</b> .....	<b>360</b>
	<i>Функциональные языки, как естественный инструмент реализации трансляторов. Синтаксические анализаторы. Методы написания трансляторов на языке Haskell. Примеры синтаксических анализаторов для различных форматов данных. Пример вычисления арифметических выражений, записанных в различной нотации.</i>	
	<i>Простейшие парсеры</i> .....	<i>361</i>
	<i>Комбинаторы синтаксического анализа</i> .....	<i>363</i>
	<i>Дополнительные комбинаторы синтаксического анализа</i> .....	<i>366</i>
	<i>Анализ нотации Бэкуса — Наура</i> .....	<i>368</i>



6.4	Библиотеки для создания трансляторов .....	372
	<i>Описание имеющихся библиотек для языка Haskell, предназначенных для создания трансляторов. Монадическая библиотека <b>Parsec</b> для самостоятельного создания трансляторов. Компилятор компиляторов <b>Happy</b>.</i>	
	Монадическая библиотека парсеров <b>Parsec</b> .....	372
	Компилятор компиляторов <b>Happy</b> .....	377
6.5	Частичные вычисления, трансформация программ и суперкомпиляция .....	381
	<i>Задача трансформации программ. Частичные вычисления, как инструмент для трансформации программ. Частичный вычислитель — инструмент для получения остаточного кода заданной функциональной программы. Интерпретатор, компилятор и компилятор компиляторов, их связь. Проекция Футамуры-Турчина. Суперкомпиляция.</i>	
	Частичные вычисления и трансляция программ .....	382
	Проекция Футамуры — Турчина .....	385
	Трансформация программ .....	387
	Вопросы для самоконтроля .....	392
	Задачи для самостоятельного решения .....	394
7	ФП и искусственный интеллект .....	395
	<i>Заключительная глава книги рассматривает такую область человеческого знания, как искусственный интеллект, то есть методы решения слабоформализованных задач, для которых не существует алгоритмического решения либо такое решение слишком сложно. Такой интерес связан с тем, что именно парадигма функционального программирования нашла свое непосредственное применение в рамках искусственного интеллекта.</i>	
7.1	Основные задачи искусственного интеллекта .....	396
	<i>Историческая справка о развитии искусственного интеллекта, как области научного исследования. Введение в базовые понятия искусственного интеллекта. Место функционального программирования в искусственном интеллекте. Функциональное и логическое программирования. Задачи искусственного интеллекта, которые могут быть решены при помощи методов и средств функционального программирования.</i>	
	История развития искусственного интеллекта .....	398
	Различные подходы к построению систем искусственного интеллекта .....	402
	Место функционального программирования в искусственном интеллекте .....	405

7.2	Нечеткая математика и функциональное программирование .....	407
	<i>Небольшой экскурс в нечеткую математику. Функции принадлежности и лингвистические переменные. Базовые операции над функциями принадлежности. Кусочно-линейные функции принадлежности. Операции сравнения, арифметические и логические операции над кусочно-линейными функциями принадлежности. Использование языка Haskell для реализации методов обработки кусочно-линейных функций принадлежности.</i>	
	<i>Базовые концепты нечеткой логики .....</i>	<i>407</i>
	<i>От нечеткой логики к нечеткой математике .....</i>	<i>409</i>
	<i>Функции принадлежности как способ описания нечетких значений .....</i>	<i>411</i>
	<i>Нечеткие и лингвистические переменные .....</i>	<i>417</i>
	<i>Операции над функциями принадлежности .....</i>	<i>418</i>
	<i>Пример модуля для обработки кусочно-линейных функций принадлежности .....</i>	<i>423</i>
7.3	Логический вывод на знаниях .....	429
	<i>Знания и данные. Модели представления знаний. Понятие логического вывода на знаниях. Стратегии вывода на знаниях. Машины вывода. Эволюция машинного вывода на знаниях. Интерпретаторы функциональных языков как естественные машины вывода. Язык Haskell и его возможности в логическом выводе на знаниях. Универсальный вывод на продукционной модели знания.</i>	
	<i>Знания и данные .....</i>	<i>430</i>
	<i>Вывод на знаниях .....</i>	<i>434</i>
	<i>Прямой нечеткий вывод .....</i>	<i>437</i>
	<i>Обратный нечеткий вывод .....</i>	<i>439</i>
	<i>Некоторые окончательные замечания о машинном выводе .....</i>	<i>442</i>
7.4	Общение с компьютером на естественном языке .....	443
	<i>Принципы общения с компьютерными системами на естественном языке. Ограниченный естественный язык, деловая проза. Трансляция фраз на естественном языке во внутренний язык представления смысла. Понимание текстов на естественном языке. Использование методов функционального программирования.</i>	
	<i>Обобщенная схема интеллектуальных диалоговых систем .....</i>	<i>444</i>
	<i>Схема анализа входного текста .....</i>	<i>447</i>
	<i>Некоторые окончательные замечания .....</i>	<i>453</i>
7.5	Перспективы функционального программирования .....	454

*Описание видения будущего функционального программирования, функциональных языков и языка Haskell. Значение функциональной парадигмы для технологии программирования вообще.*

Вопросы для самоконтроля .....	460
Задачи для самостоятельного решения .....	461

<b>Заключение .....</b>	<b>463</b>
-------------------------	------------

<b>Ответы на задачи для самостоятельного решения .....</b>	<b>465</b>
--	------------

Решения задач из главы 1 .....	465
Решения задач из главы 2 .....	467
Решения задач из главы 3 .....	474
Решения задач из главы 4 .....	477
Решения задач из главы 5 .....	483
Решения задач из главы 6 .....	488

<b>А Функциональные языки программирования и Интернет-ресурсы по функциональному программированию .....</b>	<b>496</b>
---	------------

*Список наиболее известных и широко используемых функциональных языков программирования с краткой аннотацией. Список Интернет-ресурсов, посвященных функциональному программированию как в русском сегменте Интернета, так и во всем остальном мире.*

Функциональные языки программирования .....	496
Русские Интернет-ресурсы .....	502
Иностранные Интернет-ресурсы .....	503

<b>В Опции различных сред разработки на языке Haskell .....</b>	<b>506</b>
---	------------

*Описание типичных настроек интегрированных сред разработки и компиляторов на примере продуктов HUGS 98 и GHC для полноценной работы и выполнения различных задач на языке Haskell. Список команд, ключей командной строки и внутренних директив интерпретатора HUGS 98. Список параметров командной строки для компилятора GHC. Другие функциональные средства разработки.*

Интегрированная среда разработки HUGS 98 .....	506
Компилятор GHC .....	510
Компилятор NHC .....	523
Компилятор компиляторов Hapru .....	528

<b>С</b>	<b>Описание стандартного модуля Prelude .....</b>	<b>531</b>
	<i>Список функций из стандартного модуля языка Haskell Prelude с более или менее подробным описанием.</i>	
	Функции .....	531
	Описание некоторых операторов языка Haskell .....	586
<b>Д</b>	<b>Краткий словарь терминов из области функционального программирования .....</b>	<b>589</b>
	<i>Краткий словарь терминов, имеющих отношение к функциональному программированию. для каждого термина даются ссылки на страницы, где он описан, а также переводы на некоторые иностранные языки.</i>	
	<b>Литература .....</b>	<b>599</b>
	Общая литература по функциональному программированию .....	599
	Книги, руководства и статьи по языку Haskell .....	601
	Комбинаторная логика и $\lambda$ -исчисление .....	602
	Математическая лингвистика и теория построения трансляторов .....	603
	Искусственный интеллект .....	604

# 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<sup>1</sup> 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  $\lambda$ -исчислению<sup>2</sup>. 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.

---

<sup>1</sup> 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.

<sup>2</sup> Основы комбинаторной логики и  $\lambda$ -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,  $\lambda$ -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:  $\langle \rangle$ ,  $\{ \}$  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  $\lambda$ -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  $\text{\LaTeX}$  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, вносявших 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. Тупо́та, 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  $C_1, \dots, C_n$  — это типы, а  $C$  — это тип, состоящий из множества кортежей вида  $\langle c_1, \dots, c_n \rangle, c_i \in C_i, i = \overline{1, n}$ , то говорится, что  $C$  — декартово произведение типов  $C_1, \dots, C_n$  и обозначается как:

$$C = C_1 \times \dots \times C_n. \quad (1.11)$$

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:

$$c = \text{constructor } C, \quad (1.12)$$

$$s_1, \dots, s_n = \text{selectors } C. \quad (1.13)$$

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

$$\forall c_i \in C_i, i = \overline{1, n} : c \ c_1 \dots c_n = \langle c_1, \dots, c_n \rangle. \quad (1.14)$$

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

$$\forall x \in C : \text{constructor } C (s_1x) \dots (s_nx) = x. \quad (1.15)$$

Либо в симметричной записи:

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

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  $C_1, \dots, C_n$  — это некоторые типы, а  $C$  — это тип, состоящий из объединения типов  $C_1, \dots, C_n$ , при условии выполнения «размеченности», то  $C$  называется размеченным объединением типов  $C_1, \dots, C_n$ . Обозначается этот факт как  $C = C_1 + \dots + C_n$ . 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  $P_1, \dots, P_n$  таких, что:

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

Размеченное объединение гарантирует наличие таких предикатов. Этот факт указывается записью:  $P_1, \dots, P_n = \text{predicates } C$ . 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  $N_1, \dots, N_n = \text{parts } C$ . 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* = constructor  $List(A)$

*head, tail* = selectors  $List(A)$

*isNil, isNonNil* = predicates  $List(A)$

*nil, nonNil* = 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 клона<sup>10</sup>. Первый обрабатывает  $NIL$ , второй —  $nonNIL$  соответ-

<sup>10</sup> Под клоном 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<sup>11</sup> `[]` и `(x:xs)`. Два клоза можно объединить в один с использованием технологии охраны<sup>12</sup>. 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.
```

<sup>11</sup> Определение того, что понимается под словом «образец», приведено на стр. 121 — см. определение 2.4.

<sup>12</sup> Охрана, или охранный 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 вернуть 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 вернуть 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 С). 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 = constructor ListStructure(A);
head, tail = selectors ListStructure(A);
isAtom, isNonAtom = predicates ListStructure(A);
atom, nonAtom = 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-

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

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

Как видно, вспомогательный тип  $\text{Forest}(A)$  — 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  $\text{Tree}(A)$  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  $\text{Tree}(A)$  и с помощью селекторов и частей для типа  $\text{Forest}(A)$ . 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  $\text{Forest}(A)$  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.

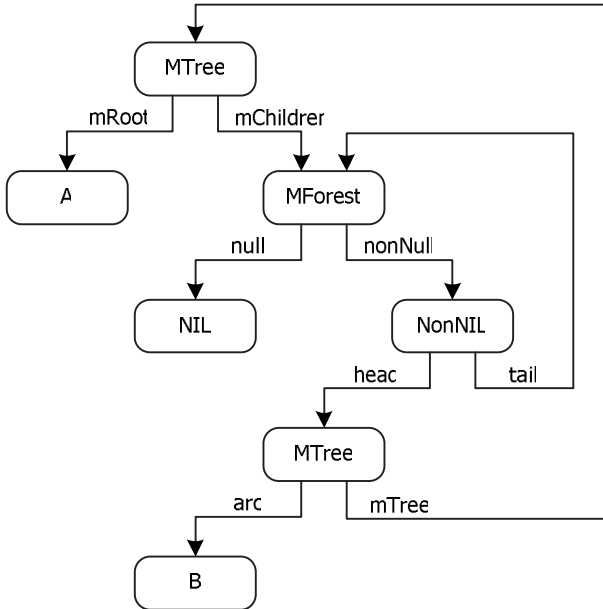
Метка на вершине хранит некоторую информацию, связанную с этой вершиной. Пусть эта информация имеет базовый тип  $A$ . Метка на дуге по существу хранит тип этой дуги, который зависит от решаемой задачи. Пусть такая метка имеет базовый тип  $B$ . Тогда тип  $MTree(A, B)$  можно определить следующим образом:

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

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 приводят-ся шаблонные функции (для списков эти функции приведены в примере 1.5).

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

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

Далее приводится определение шаблонных функций, которые позволяют об-рабатывать значения типа  $MTree(A, B)$  для любых целей. Все остальные кон-кретные 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 обозначе-ний. 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 листевой 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 листевой 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 — листевой 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
```