

Функциональное программирование

Ярон Мински
Анил Мадхавапедди
Джейсон Хикки

Программирование

на языке OCaml

УДК 004.6
ББК 32.973.26
М57

Мински Я., Мадхавapedди А., Хикки Дж.
М57 Программирование на языке OSaml / пер. с англ. А. Н. Киселева. – М.: ДМК
Пресс, 2014. – 536 с.: ил.

ISBN 978-5-97060-539-6

Эта книга введет вас в мир OSaml, надежный язык программирования, обладающий большой выразительностью, безопасностью и быстродействием. Пройдя через множество примеров, вы быстро поймете, что OSaml – это превосходный инструмент, позволяющий писать быстрый, компактный и надежный системный код.

Вы познакомитесь с основными понятиями языка, узнаете о приемах и инструментах, помогающих превратить OSaml в эффективное средство разработки практических приложений. В конце книги вы сможете углубиться в изучение тонких особенностей инструментов компилятора и среды выполнения OSaml.

УДК 004.6
ББК 32.973.26

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 978-1-449-32391-2 (анг.)

ISBN 978-5-97060-539-6 (рус.)

© 2014 Yaron Minsky, Anil Madhavapeddy,
Jason Hickey

© Оформление, перевод, ДМК Пресс, 2014

Содержание

Вступление	14
Часть I. Основы языка	22
Глава 1. Введение	23
OSaml как калькулятор.....	23
Функции и автоматический вывод типов.....	25
Автоматический вывод типов.....	27
Автоматический вывод обобщенных типов.....	28
Кортежи, списки, необязательные значения и сопоставление с образцом.....	30
Кортежи.....	30
Списки.....	31
Необязательные значения.....	38
Записи и варианты.....	40
Императивное программирование.....	42
Массивы.....	42
Изменяемые поля записей.....	43
Ссылки.....	45
Циклы for и while.....	46
Законченная программа.....	48
Компиляция и запуск.....	48
Что дальше.....	49
Глава 2. Переменные и функции	50
Переменные.....	50
Сопоставление с образцом и let.....	53
Функции.....	54
Анонимные функции.....	55
Функции нескольких аргументов.....	57
Рекурсивные функции.....	59
Префиксные и инфиксные операторы.....	60
Объявление функций с помощью ключевого слова function.....	65
Аргументы с метками.....	66
Необязательные аргументы.....	69
Глава 3. Списки и образцы	77
Основы списков.....	77
Использование сопоставления с образцом для извлечения данных из списка.....	78
Ограничения (и благословения) сопоставления с образцом.....	80
Производительность.....	81
Определение ошибок.....	83

Эффективное использование модуля List	84
Другие полезные функции из модуля List	88
Хвостовая рекурсия	91
Компактность и скорость сопоставления с образцом	93
Глава 4. Файлы, модули программы	98
Программы в единственном файле.....	98
Программы и модули из нескольких файлов.....	101
Сигнатуры и абстрактные типы.....	103
Конкретные типы в сигнатурах.....	106
Вложенные модули	107
Открытие модулей.....	109
Подключение модулей	111
Типичные ошибки при работе с модулями	113
Несовпадение типов	113
Отсутствие определений.....	114
Несоответствие определений типов	114
Циклические зависимости	115
Проектирование с применением модулей.....	117
Старайтесь не экспортировать конкретные типы.....	117
Продумывайте синтаксис вызовов.....	117
Создавайте однородные интерфейсы.....	118
Определяйте интерфейсы до реализации	119
Глава 5. Записи	120
Сопоставление с образцом и полнота	122
Уплотнение полей.....	124
Повторное использование имен полей.....	125
Функциональные обновления.....	129
Изменяемые поля	131
Поля первого порядка	132
Глава 6. Варианты	137
Универсальные образцы и рефакторинг.....	139
Объединение записей и вариантов	141
Варианты и рекурсивные структуры данных.....	145
Полиморфные варианты	149
Пример: и снова о цветных терминалах	151
Когда следует использовать полиморфные варианты.....	157
Глава 7. Обработка ошибок	159
Типы возвращаемых значений с признаком ошибки	159
Кодирование ошибок в результате	160
Error и Or_error	161

Функция bind и другие идиомы обработки ошибок.....	163
Исключения.....	165
Вспомогательные функции для возбуждения исключений.....	167
Обработчики исключений.....	169
Восстановление работоспособности после исключений.....	169
Перехват определенных исключений.....	170
Трассировка стека.....	172
От исключений к типам с информацией об ошибках и обратно.....	174
Выбор стратегии обработки ошибок.....	175
Глава 8. Императивное программирование.....	177
Пример: императивные словари.....	177
Элементарные изменяемые данные.....	182
Данные в формах, подобных массивам.....	182
Изменяемые поля записей и объектов и ссылочные ячейки.....	183
Внешние функции.....	184
Циклы for и while.....	184
Пример: двусвязные списки.....	186
Изменение списка.....	188
Итеративные функции.....	189
Отложенные вычисления и другие благоприятные эффекты.....	190
Мемоизация и динамическое программирование.....	192
Ввод и вывод.....	200
Терминальный ввод/вывод.....	200
Форматированный вывод с помощью printf.....	202
Файловый ввод/вывод.....	204
Порядок вычислений.....	207
Побочные эффекты и слабый полиморфизм.....	209
Ограничение значений.....	210
Частичное применение и ограничение значения.....	211
Ослабление ограничения значений.....	212
В заключение.....	215
Глава 9. Функторы.....	216
Простейший пример.....	216
Более практичный пример: вычисления с применением интервалов.....	218
Создание абстрактных функторов.....	222
Совместно используемые ограничения.....	223
Деструктивная подстановка.....	225
Использование нескольких интерфейсов.....	227
Расширение модулей.....	231
Глава 10. Модули первого порядка.....	235
Приемы работы с модулями первого порядка.....	235

Пример: фреймворк обработки запросов	241
Реализация обработчика запросов	242
Диспетчеризация запросов по нескольким обработчикам	244
Загрузка и выгрузка обработчиков запросов.....	248
Жизнь без модулей первого порядка	252
Глава 11. Объекты	253
Объекты OSaml	253
Полиморфизм объектов.....	255
Неизменяемые объекты	257
Когда следует использовать объекты.....	258
Подтипизация	259
Подтипизация в ширину	259
Подтипизация в глубину	260
Вариантность	261
Сужение	265
Подтипизация и рядный полиморфизм	267
Глава 12. Классы	269
Классы в OSaml	269
Параметры класса и полиморфизм.....	270
Типы объектов и интерфейсы.....	272
Функциональные итераторы.....	274
Наследование	276
Типы классов	277
Открытая рекурсия	278
Скрытые методы.....	280
Бинарные методы.....	281
Виртуальные классы и методы.....	285
Создание простых фигур	285
Инициализаторы.....	288
Множественное наследование.....	288
Как выполняется разрешение имен	289
Примеси	290
Отображение анимированных фигур.....	293
Часть II. Инструменты и технологии	295
Глава 13. Отображения и хэш-таблицы	296
Отображения	297
Создание отображений с компараторами	298
Деревья.....	301
Полиморфные компараторы.....	302
Множества	304

Соответствие интерфейсу Comparable.S	304
Хэш-таблицы	307
Соответствие интерфейсу Hashable.S	310
Выбор между отображениями и хэш-таблицами.....	311
Глава 14. Анализ командной строки.....	315
Простейший анализ командной строки	315
Анонимные аргументы	316
Определение простых команд.....	317
Выполнение простых команд.....	317
Типы аргументов	319
Определение собственных типов аргументов	320
Необязательные аргументы и аргументы по умолчанию.....	321
Последовательности аргументов	324
Добавление поддержки передачи именованных флагов в командной строке.....	325
Группировка подкоманд	327
Расширенное управление парсингом.....	329
Типы в основе Command.Spec	330
Объединение фрагментов спецификаций	331
Интерактивный запрос ввода.....	333
Добавление аргументов с метками в функции обратного вызова.....	335
Автодополнение командной строки средствами Bash	336
Создание фрагментов автодополнения	336
Установка фрагмента автодополнения	337
Альтернативные парсеры командной строки.....	338
Глава 15. Обработка данных JSON	339
Основы JSON	339
Парсинг данных в формате JSON с помощью Yojson	340
Выборка значений из структур JSON	343
Конструирование значений JSON	346
Использование нестандартных расширений JSON	348
Автоматическое отображение JSON в типы OCaml.....	350
Основы ATD.....	350
Аннотации ATD	351
Компиляция спецификаций ATD в код на OCaml.....	352
Пример: запрос информации об организации в GitHub	353
Глава 16. Парсинг с помощью OCamllex и Menhir	357
Лексический анализ и парсинг	358
Определение парсера.....	360
Описание грамматики.....	360
Парсинг последовательностей	362
Определение лексического анализатора.....	364

Вступление.....	364
Регулярные выражения.....	365
Лексические правила	366
Рекурсивные правила	367
Объединяем все вместе.....	368
Глава 17. Сериализация данных с применением s-выражений	371
Основа использования.....	372
Преобразование типов OCaml в s-выражения	374
Формат Sexp	376
Сохранение инвариантов	377
Вывод информативных сообщений об ошибках.....	380
Директивы sexp-преобразований	382
sexp_oraque.....	383
sexp_list	384
sexp_option.....	385
Определение значений по умолчанию.....	385
Глава 18. Конкурентное программирование с помощью Async	388
Основа Async	389
Ivar и cprop	393
Примеры: эхо-сервер.....	395
Усовершенствование эхо-сервера	399
Пример: поиск определений с помощью DuckDuckGo.....	401
Обработка URI.....	402
Парсинг строк JSON.....	402
Выполнение запроса HTTP	403
Обработка исключений.....	406
Мониторы.....	408
Пример: обработка исключений при работе с DuckDuckGo.....	410
Тайм-ауты, отмена и выбор.....	413
Работа с системными потоками	416
Защищенность данных в потоках и блокировки.....	419
Часть III. Система времени выполнения	421
Глава 19. Интерфейс внешних функций	422
Пример: интерфейс к терминалу	423
Простые скалярные типы языка C.....	427
Указатели и массивы.....	429
Выделение памяти для указателей.....	430
Использование представлений для отображения составных значений.....	431

Структуры и объединения	432
Определение структуры	432
Добавление полей в структуры	433
Незавершенные определения структур	433
Определение массивов	437
Передача функций в код на С	438
Пример: быстрая сортировка в командной строке	439
Дополнительная информация о взаимодействии с кодом на С	441
Организация структур в памяти	442
Глава 20. Представление значений в памяти	444
Блоки и значения ОСaml	445
Различение целых чисел и указателей во время выполнения	445
Блоки и значения	447
Целые числа, символы и другие простые типы	448
Кортежи, записи и массивы	448
Вещественные числа и массивы	449
Варианты и списки	450
Полиморфные варианты	452
Строковые значения	453
Нестандартные блоки памяти	454
Управление внешней памятью средствами Varray	454
Глава 21. Сборка мусора	456
Алгоритм сборки мусора	456
Сборка мусора с разделением на поколения	457
Быстрая вспомогательная куча	457
Выделение памяти во вспомогательной куче	458
Основная куча долгоживущих блоков	459
Выделение памяти в основной куче	460
Стратегии распределения памяти	461
Маркировка и сканирование кучи	462
Компактификация кучи	463
Указатели между поколениями	464
Подключение функций-финализаторов к значениям	467
Глава 22. Компиляторы: парсинг и контроль типов	470
Обзор инструментов компилятора	470
Парсинг исходного кода	472
Синтаксические ошибки	473
Автоматическое оформление отступов в исходном коде	473
Автоматическое создание документации на основе интерфейсов	475
Препроцессинг исходного кода	477
Использование Camlp4 в интерактивной оболочке	479

Запуск Camlp4 из командной строки	480
Препроцессинг сигнатур модулей	482
Дополнительные источники информации о Camlp4	483
Статическая проверка типов	483
Демонстрация типов, выводимых компилятором	484
Вывод типов	486
Модули и отдельная компиляция	491
Упаковка модулей вместе	493
Сокращение путей к модулям в сообщениях об ошибках	495
Типизированное синтаксическое дерево	496
Использование <code>ocp-index</code> для поддержки автодополнения	496
Непосредственное исследование типизированного синтаксического дерева	497
Глава 23. Компиляторы: байт-код и машинный код	501
Нетипизированная <code>lambda</code> -форма	501
Оптимизация сопоставлений с образцом	501
Оценка производительности сопоставления с образцом	504
Переносимый байт-код	506
Компиляция и компоновка байт-кода	507
Выполнение байт-кода	508
Встраивание байт-кода OCaml в программы на C	509
Компиляция быстрого машинного кода	511
Исследование ассемблерного кода	511
Отладка двоичных выполняемых файлов	515
Профилерование машинного кода	519
Встраивание машинного кода в программы на C	521
Сводка по расширениям имен файлов	522
Алфавитный указатель	523

Глава 3

Списки и образцы

В этой главе мы сосредоточимся на двух элементах, чаще других используемых при программировании на языке OCaml: списках и сопоставлении с образцом. Оба они обсуждались в главе 1, но здесь мы исследуем их более подробно, используя их совместно, чтобы проиллюстрировать особенности друг друга.

ОСНОВЫ СПИСКОВ

Списки в языке OCaml являются неизменяемыми, конечными последовательностями элементов одного типа. Как вы уже знаете, списки в OCaml можно создавать с помощью квадратных скобок и точек с запятой:

OCaml utop

<https://github.com/realworldocaml/examples/tree/v1/code/lists-and-patterns/main.topscript>

```
# [1;2;3];;  
- : int list = [1; 2; 3]
```

и с использованием эквивалентной нотации :::

OCaml utop (part 1)

<https://github.com/realworldocaml/examples/tree/v1/code/lists-and-patterns/main.topscript>

```
# 1 :: (2 :: (3 :: [])) ;;  
- : int list = [1; 2; 3]  
# 1 :: 2 :: 3 :: [] ;;  
- : int list = [1; 2; 3]
```

Как видите, оператор :: является правоассоциативным, а это означает, что списки можно конструировать без применения круглых скобок. Пустой список [] используется здесь для обозначения конца списка. Отметьте, что пустой список является полиморфным, то есть может использоваться с элементами любого типа, например:

OCaml utop (part 2)

<https://github.com/realworldocaml/examples/tree/v1/code/lists-and-patterns/main.topscript>

```
# let empty = [] ;;  
val empty : 'a list = []  
# 3 :: empty ;;  
- : int list = [3]  
# "three" :: empty ;;  
- : string list = ["three"]
```

Способ, каким оператор `::` присоединяет элементы в начало списка, отражает тот факт, что списки в языке OCaml являются односвязными. На рис. 3.1 схематически изображено, как выглядит список `1 :: 2 :: 3 :: []` в виде структуры данных. Заключительная стрелка (из прямоугольника с числом 3) указывает на пустой список.

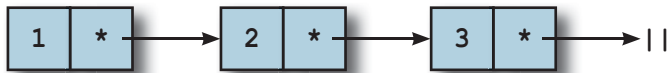


Рис. 3.1 ❖ Односвязный список как структура данных

Каждый оператор `::` фактически добавляет новый блок в цепочку. Каждый блок состоит из двух компонентов: ссылки на данные, составляющие элемент списка, и ссылки на оставшуюся часть списка. Это объясняет, как оператор `::` может наращивать список, не изменяя его; новый элемент списка сохраняется отдельно, без изменения существующих:

OCaml utop (part 3)

<https://github.com/realworldocaml/examples/tree/v1/code/lists-and-patterns/main.topscript>

```
# let l = 1 :: 2 :: 3 :: [];;
val l : int list = [1; 2; 3]
# let m = 0 :: 1;;
val m : int list = [0; 1; 2; 3]
# l;;
- : int list = [1; 2; 3]
```

Использование сопоставления с образцом для извлечения данных из списка

Читать данные из списка можно с помощью инструкции `match`. Ниже приводится простой пример рекурсивной функции, определяющей сумму всех элементов списка:

OCaml utop (part 4)

<https://github.com/realworldocaml/examples/tree/v1/code/lists-and-patterns/main.topscript>

```
# let rec sum l =
  match l with
  | [] -> 0
  | hd :: tl -> hd + sum tl
;;
val sum : int list -> int = <fun>
# sum [1;2;3];;
- : int = 6
# sum [];;
- : int = 0
```

Этот код следует соглашению об использовании имени `hd` для представления первого элемента (или головы (`head`)) списка и имени `tl` для представления оставшейся части (или хвоста (`tail`)) списка.

Инструкция `match` в функции `sum` в действительности выполняет две операции: во-первых, она действует как инструмент выбора, выделяя различные возможные варианты, и, во-вторых, дает возможность присвоить имена отдельным элементам исходной структуры данных. В данном случае переменные `hd` и `tl` связываются вторым образом в инструкции `match`. Переменные, связанные таким способом, можно использовать в выражении справа от стрелки, следующей за текущим образом.

Тот факт, что инструкцию `match` можно использовать для связывания новых переменных, может быть источником недопонимания. Давайте посмотрим, как. Представьте, что нам требуется написать функцию, отфильтровывающую из списка элементы, равные некоторому значению. У вас наверняка появится желание написать код, как показано ниже, но при попытке скомпилировать его компилятор сразу же выдаст предупреждение:

OCaml utop (part 5)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# let rec drop_value l to_drop =
  match l with
  | [] -> []
  | to_drop :: tl -> drop_value tl to_drop
  | hd :: tl -> hd :: drop_value tl to_drop
;;
```

Characters 114-122:

Warning 11: this match case is unused.

```
val drop_value : 'a list -> 'a -> 'a list = <fun>
(Предупреждение 11: этот образец не используется)
val drop_value : 'a list -> 'a -> 'a list = <fun>
```

Более того, функция будет работать неправильно, отфильтровывая все элементы, а не только те, что равны указанному значению:

OCaml utop (part 6)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# drop_value [1;2;3] 2;;
- : int list = []
```

Что не так?

Здесь важно понимать, что присутствие `to_drop` во втором образце не подразумевает сравнения первого элемента списка со значением аргумента `to_drop`, переданного функции `drop_value`. Вместо этого просто создается новая переменная `to_drop`. Она будет связана с первым элементом списка и скроет прежнее определение `to_drop`. Третий образец окажется неиспользуемым, потому что фактически тот же самый шаблон используется во втором образце.

Лучший выход из этой ситуации – вообще не использовать сопоставление с образцом для сравнения первого элемента списка с аргументом `to_drop`, а применить обычную инструкцию `if`:

OCaml utop (part 7)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# let rec drop_value l to_drop =
  match l with
  | [] -> []
  | hd :: tl ->
    let new_tl = drop_value tl to_drop in
    if hd = to_drop then new_tl else hd :: new_tl
;;
val drop_value : 'a list -> 'a -> 'a list = <fun>
# drop_value [1;2;3] 2;;
- : int list = [1; 3]
```

Обратите внимание, что в случае, когда требуется отбросить конкретное литеральное значение (а не значение, переданное в переменной), можно использовать нечто, подобное первоначальной версии `drop_value`:

OCaml utop (part 8)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# let rec drop_zero l =
  match l with
  | [] -> []
  | 0 :: tl -> drop_zero tl
  | hd :: tl -> hd :: drop_zero tl
;;
val drop_zero : int list -> int list = <fun>
# drop_zero [1;2;0;3];;
- : int list = [1; 2; 3]
```

Ограничения (и благословения) сопоставления с образцом

Предыдущий пример наглядно демонстрирует, что сопоставление с образцом не может использоваться для выражения произвольных условий. Образцы могут описывать организацию структуры данных и даже включать литералы, как в примере реализации функции `drop_zero`, но на этом круг их возможностей ограничивается. С помощью образца можно проверить, содержит ли список два элемента, но нельзя проверить равенство первых двух элементов.

Образцы можно рассматривать как специализированный подязык, на котором можно выразить ограниченное (но достаточно богатое) множество условий. Ограниченность языка образцов одновременно является и благословением, упрощая их поддержку и оптимизацию в компиляторе. В частности, эффективность

инструкций `match` и способность компилятора обнаруживать ошибки во многом обусловлены ограниченной природой образцов.

Производительность

По простоте душевной можно было бы подумать, что инструкция `match` последовательно проверяет все образцы, чтобы найти совпадение. В ситуациях, когда образцы снабжаются произвольным ограничивающим кодом, так и есть. Но чаще компилятор OCaml оказывается в состоянии сгенерировать машинный код, сразу выполняющий переход к нужной ветви, опираясь на множество простых проверок во время выполнения.

Например, взгляните на следующие, немного дурацкие функции, увеличивающие целое число на единицу. Первая реализована на основе инструкции `match`, а вторая – в виде последовательности инструкций `if`:

OCaml utop (part 9)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# let plus_one_match x =
  match x with
  | 0 -> 1
  | 1 -> 2
  | 2 -> 3
  | _ -> x + 1

let plus_one_if x =
  if x = 0 then 1
  else if x = 1 then 2
  else if x = 2 then 3
  else x + 1
;;
val plus_one_match : int -> int = <fun>
val plus_one_if : int -> int = <fun>
```

Обратите внимание на образец `_` в первой функции. Это шаблонный образец – он совпадает с любым значением, но не связывает совпавшее значение ни с каким именем переменной.

Если проверить производительность этих функций, можно заметить, что `plus_one_if` выполняется намного медленнее, чем `plus_one_match`, причем превосходство возрастает с увеличением возможных вариантов. Следующий фрагмент определяет производительность этих функций с помощью библиотеки `core_bench`, которую можно установить командой `opam install core_bench`, выполнив ее в командной строке:

OCaml utop (part 10)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# #require "core_bench";;
# open Core_bench.Std;;
```

```
# let run_bench tests =
  Bench.bench
    ~ascii_table:true
    ~display:Textutils.Ascii_table.Display.column_titles
    tests
  ;;
val run_bench : Bench.Test.t list -> unit = <fun>
# [ Bench.Test.create ~name:"plus_one_match" (fun () ->
  ignore (plus_one_match 10))
  ; Bench.Test.create ~name:"plus_one_if" (fun () ->
  ignore (plus_one_if 10)) ]
|> run_bench
;;
Estimated testing time 20s (change using -quota SECS).
```

Name	Time (ns)	% of max
plus_one_match	46.81	68.21
plus_one_if	68.63	100.00

```
- : unit = ()
```

Вот вам другой, менее искусственный пример. Мы можем реализовать функцию `sum`, о которой писалось выше в этой главе, с использованием инструкции `if` вместо `match`, и применить в ней функции `is_empty`, `hd_exn` и `tl_exn` из модуля `List` для деконструкции списка:

OCaml utop (part 11)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# let rec sum_if l =
  if List.is_empty l then 0
  else List.hd_exn l + sum_if (List.tl_exn l)
;;
val sum_if : int list -> int = <fun>
```

Давайте еще раз проверим производительность, чтобы увидеть разницу:

OCaml utop (part 12)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# let numbers = List.range 0 1000 in
[ Bench.Test.create ~name:"sum_if" (fun () -> ignore (sum_if numbers))
; Bench.Test.create ~name:"sum" (fun () -> ignore (sum numbers)) ]
|> run_bench
;;
Estimated testing time 20s (change using -quota SECS).
```

Name	Time (ns)	% of max
sum_if	110_535	100.00
sum	22_361	20.23

```
- : unit = ()
```


На этот раз преимущество реализации на основе инструкции `match` оказалось намного разительнее. Большая разница обусловлена необходимостью выполнять одну и ту же работу много раз, так как при каждом вызове функции приходится повторно анализировать первый элемент списка, чтобы определить, является ли он пустой ячейкой. В инструкции `match` эта работа выполняется точно один раз на элемент списка.

Вообще говоря, сопоставление с образцом действует эффективнее любого другого кода, который вы только сможете придумать. Известным исключением является сопоставление со строками, которые проверяются последовательно, из-за чего инструкция `match` с длинной последовательностью строк может проигрывать приему на основе хэш-таблицы. Но в большинстве других случаев сопоставление с образцом выходит победителем.

Определение ошибок

Однако более важной особенностью, чем производительность, является способность инструкции `match` обнаруживать ошибки. Мы уже видели пример способности языка OCaml обнаруживать проблемы в операциях сопоставления с образцом: в ошибочной реализации `drop_value`, когда компилятор OCaml предупредил нас, что последний образец является избыточным. Не существует алгоритмов, которые могли бы определить избыточность предиката, написанного на универсальном языке, но в контексте образцов эта задача решается достаточно надежно.

Компилятор OCaml также проверяет инструкции `match` на полноту. Взгляните, что произойдет, если в реализации `drop_zero` удалить одну из ветвей:

OCaml utop (part 13)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# let rec drop_zero l =
  match l with
  | [] -> []
  | 0 :: tl -> drop_zero tl
;;
```

Characters 26-84:

Warning 8: this pattern-matching is not exhaustive.

Here is an example of a value that is not matched:

```
1::_
```

(Предупреждение 8: это сопоставление не является исчерпывающим.

Ниже следует пример значения, не соответствующего сопоставлению:

```
1::_)
```

```
val drop_zero : int list -> 'a list = <fun>
```

Как видите, компилятор сгенерировал предупреждение, сообщив, что мы пропустили возможный вариант, а также привел пример вероятного образца.

Даже в таких простых ситуациях, как эта, проверка на полноту может оказаться весьма полезной. Но, как будет показано в главе 6, ценность этой проверки увеличивается еще больше с ростом сложности примеров, особенно когда в работу включаются типы данных, определяемые пользователем. Помимо вылавливания

ошибок, эти проверки могут играть роль своеобразного инструмента рефакторинга, направляя вас к фрагментам кода, где требуется изменить код, чтобы привести его в соответствие с изменением типов.

Эффективное использование модуля List

К настоящему моменту мы написали немало кода, использующего сопоставление с образцом и рекурсивные функции для обработки списков. Но в реальной жизни вы чаще будете использовать для этих целей модуль List, который битком набит функциями, реализующими наиболее типичные операции со списками.

Давайте разберем конкретный пример и посмотрим, как применять этот модуль на практике. Напишем функцию `render_table`, принимающую список заголовков столбцов и список строк, которая будет выводить их в виде отформатированной таблицы:

OCaml utop (part 69)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# printf "%s\n"
(render_table
  ["language";"architect";"first release"]
  [ ["Lisp" ;"John McCarthy" ;"1958"] ;
    ["C" ;"Dennis Ritchie";"1969"] ;
    ["ML" ;"Robin Milner" ;"1973"] ;
    ["OCaml";"Xavier Leroy" ;"1996"] ;
  ]));
| language | architect      | first release |
|-----+-----+-----|
| Lisp     | John McCarthy  | 1958         |
| C        | Dennis Ritchie | 1969         |
| ML       | Robin Milner   | 1973         |
| OCaml    | Xavier Leroy   | 1996         |
- : unit = ()
```

Первое, что нужно сделать, – написать функцию, определяющую максимальную ширину каждой колонки. Для этого можно преобразовать заголовок и все строки данных в список целых чисел, отражающих длину, а затем выбрать самый большой элемент. Реализовать весь необходимый код вручную будет довольно трудоемкой задачей, но мы можем избавить себя от лишних сложностей, воспользовавшись тремя функциями из модуля List: `map`, `map2_exn` и `fold`.

Описать действие функции List.map проще простого. Она принимает список и функцию для преобразования элементов этого списка и возвращает новый список с преобразованными элементами. То есть мы можем записать:

OCaml utop (part 14)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# List.map ~f:String.length ["Hello"; "World!"];
- : int list = [5; 6]
```

Функция `List.map2_exn` похожа на `List.map`, за исключением того, что она принимает два списка и функцию для их объединения. То есть мы можем записать:

OCaml utop (part 15)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# List.map2_exn ~f:Int.max [1;2;3] [3;2;1];;
- : int list = [3; 2; 3]
```

Суффикс `_exn` здесь обозначает, что функция возбудит исключение, если списки будут иметь разную длину:

OCaml utop (part 16)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# List.map2_exn ~f:Int.max [1;2;3] [3;2;1;0];;
Exception: (Invalid_argument "length mismatch in rev_map2_exn: 3 <> 4").
```

Функция `List.fold` самая сложная из этой троицы. Она принимает три аргумента: обрабатываемый список, начальное значение аккумулятора и функцию для обновления аккумулятора. Функция `List.fold` выполняет обход элементов списка слева направо, на каждом шаге обновляет аккумулятор и в конце возвращает получившееся значение аккумулятора. Кое-что из этого можно увидеть, взглянув на сигнатуру функции `fold`:

OCaml utop (part 17)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# List.fold;;
- : 'a list -> init:'accum -> f:( 'accum -> 'a -> 'accum) -> 'accum = <fun>
```

Функцию `List.fold` можно использовать как своеобразный сумматор для списка:

OCaml utop (part 18)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# List.fold ~init:0 ~f:(+) [1;2;3;4];;
- : int = 10
```

Этот пример получился таким простым потому, что аккумулятор и элементы списка принадлежат одному типу. Но функция `fold` не ограничена такими ситуациями. Функцию `fold` можно, к примеру, задействовать для перестановки элементов списка в обратном порядке, в этом случае аккумулятор сам будет являться списком:

OCaml utop (part 19)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# List.fold ~init:[] ~f:(fun list x -> x :: list) [1;2;3;4];;
- : int list = [4; 3; 2; 1]
```

Давайте объединим эти три функции для поиска максимальной ширины каждой колонки:

OCaml utop (part 20)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# let max_widths header rows =
  let lengths l = List.map ~f:String.length l in
  List.fold rows
    ~init:(lengths header)
    ~f:(fun acc row ->
      List.map2_exn ~f:Int.max acc (lengths row))
  ;;
val max_widths : string list -> string list list -> int list = <fun>
```

С помощью `List.map` мы определили функцию `lengths`, которая преобразует список строк в список целочисленных длин. Затем мы использовали `List.fold` для обхода строк и с помощью `map2_exn` определили максимальное значение аккумулятора длин значений в каждой строке таблицы, при этом аккумулятор инициализировался длинами заголовков.

Теперь, когда ширина каждой колонки известна, можно приступить к коду, генерирующему строку, отделяющую заголовок от остальной части таблицы. Мы сделаем это с помощью `String.make`, отобразив целочисленные значения в строки из дефисов соответствующей длины. Затем объединим эти строки дефисов с помощью `String.concat`, которая выполняет конкатенацию строк через необязательный разделитель, и с помощью оператора `^` добавим внешние ограничители колонок:

OCaml utop (part 21)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# let render_separator widths =
  let pieces = List.map widths
    ~f:(fun w -> String.make (w + 2) '-')
  in
  "|" ^ String.concat ~sep:"+" pieces ^ "|"
  ;;
val render_separator : int list -> string = <fun>
# render_separator [3;6;2];;
- : string = "|-----+-----+-----|"
```

Обратите внимание, что строку дефисов мы сделали на два символа длиннее, чтобы обеспечить наличие не менее одного пробела с каждой стороны записи в таблице.

Производительность `String.concat` и `^`

В предыдущем примере мы объединяли строки двумя разными способами: с помощью функции `String.concat`, которая принимает список строк, и оператора `^`, осуществляющего объединение пары строк. Старайтесь избегать использования оператора `^` для объединения большого числа строк, потому что каждый раз он размещает в памяти новую строку. То есть следующий код


```

String.concat ~sep:"\n"
  (render_row header widths
   :: render_separator widths
   :: List.map rows ~f:(fun row -> render_row row widths)
  )
);;
val render_table : string list -> string list list -> string = <fun>

```

Другие полезные функции из модуля List

В предыдущем примере мы использовали всего три функции из модуля List. Мы не можем позволить себе привести здесь полное описание всего интерфейса модуля (для этого вам следует обратиться к электронной документации), но некоторые особенно полезные функции отметим.

Объединение элементов списка с помощью List.reduce

Функция List.fold, описанная выше, — очень мощная и универсальная функция. Но иногда желательно иметь инструмент попроще. Одним из таких инструментов является функция List.reduce. По сути, это специализированная версия функции List.fold, не требующая явно указывать начальное значение. Она создает аккумулятор, тип которого соответствует типу элементов обрабатываемого списка.

Вот как выглядит ее сигнатура:

OCaml utop (part 27)

```

https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript
# List.reduce;;
- : 'a list -> f:('a -> 'a -> 'a) -> 'a option = <fun>

```

Функция reduce имеет необязательное возвращаемое значение, то есть для пустого списка она вернет None.

Теперь посмотрим, как действует функция reduce:

OCaml utop (part 28)

```

https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript
# List.reduce ~f:(+) [1;2;3;4;5];;
- : int option = Some 15
# List.reduce ~f:(+) [];;
- : int option = None

```

Фильтрация с помощью List.filter и List.filter_map

Очень часто при обработке списков бывает необходимо сосредоточить внимание лишь на подмножестве элементов списка. Такую возможность дает функция List.filter:

OCaml utop (part 29)

```

https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript
# List.filter ~f:(fun x -> x mod 2 = 0) [1;2;3;4;5];;
- : int list = [2; 4]

```

Обратите внимание, что `mod` используется здесь в качестве инфиксного оператора, как описывалось в главе 2.

Иногда может потребоваться одновременно преобразовать и отфильтровать элементы списка. В этом случае вам поможет функция `List.filter_map`. Функция, передаваемая в вызов `List.filter_map`, должна возвращать необязательное значение, а `List.filter_map` отбросит все элементы, для которых будет получено значение `None`.

Следующее выражение составляет список расширений имен файлов, присутствующих в текущем каталоге, и передает его функции `List.dedup` для удаления повторяющихся значений. Обратите внимание, что в этом примере используются также функции из других модулей, включая `Sys.ls_dir`, возвращающую список содержимого каталога, и `String.rsplit2`, разбивающую строку по самому правому указанному символу:

OCaml utop (part 30)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# List.filter_map (Sys.ls_dir ".") ~f:(fun fname ->
  match String.rsplit2 ~on:'.' fname with
  | None | Some ("",_) -> None
  | Some (_,ext) ->
    Some ext)
|> List.dedup
;;
- : string list = ["ascii"; "ml"; "mli"; "topscript"]
```

Предыдущий код может также служить примером применения ИЛИ-шаблона, позволяющего указывать несколько образцов в одной ветви инструкции `match`. В данном случае таким шаблоном является образец `None | Some ("",_)`. Как будет показано далее, ИЛИ-шаблоны могут присутствовать в любом месте в больших образцах.

Деление списков с помощью `List.partition_tf`

Еще одной полезной операцией, тесно связанной с фильтрацией, является операция деления списка. Функция `List.partition_tf` принимает список и функцию, возвращающую логическое значение для каждого элемента списка, и создает два списка. Суффикс `tf` в имени служит напоминанием для забывчивых пользователей, что элементы, для которых указанная функция вернула `true`, помещаются в первый из возвращаемых списков, а элементы, получившие `false`, – во второй. Например:

OCaml utop (part 31)

<https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript>

```
# let is_ocaml_source s =
  match String.rsplit2 s ~on:'.' with
  | Some (_,("ml"|"mli")) -> true
  | _ -> false
```

```
;;
val is_ocaml_source : string -> bool = <fun>
# let (ml_files, other_files) =
    List.partition_tf (Sys.ls_dir ".") ~f:is_ocaml_source;;
val ml_files : string list = ["example.mli"; "example.ml"]
val other_files : string list = ["main.topscrip"; "lists_layout.ascii"]
```

Комбинирование списков

Другая распространенная операция над списками – конкатенация. В действительности модуль `List` позволяет выполнять эту операцию несколькими разными способами. Во-первых, существует функция `List.append`, выполняющая объединение двух списков:

OCaml utop (part 32)

<https://github.com/realworldocaml/examples/tree/v1/code/lists-and-patterns/main.topscrip>

```
# List.append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Имеется также оператор `@`, действующий подобно функции `List.append`:

OCaml utop (part 33)

<https://github.com/realworldocaml/examples/tree/v1/code/lists-and-patterns/main.topscrip>

```
# [1;2;3] @ [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Конкатенацию списков в списке можно выполнить с помощью функции `List.concat`:

OCaml utop (part 34)

<https://github.com/realworldocaml/examples/tree/v1/code/lists-and-patterns/main.topscrip>

```
# List.concat [[1;2];[3;4;5];[6];[]];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Ниже приводится пример использования функции `List.concat` в паре с `List.map` для создания рекурсивного списка дерева каталогов:

OCaml utop (part 35)

<https://github.com/realworldocaml/examples/tree/v1/code/lists-and-patterns/main.topscrip>

```
# let rec ls_rec s =
    if Sys.is_file_exn ~follow_symlinks:true s
    then [s]
    else
        Sys.ls_dir s
        |> List.map ~f:(fun sub -> ls_rec (s ^/ sub))
        |> List.concat
;;
val ls_rec : string -> string list = <fun>
```


Обратите внимание на инфиксный оператор `^/` из библиотеки `Core`. Он добавляет новый элемент в строку пути к файлу. Этот оператор является эквивалентом функции `Filename.concat` из библиотеки `Core`.

Комбинация функций `List.map` и `List.concat`, представленная выше, используется настолько часто, что специально была создана функция `List.concat_map`, объединяющая эти две функции в одну, но действующая более эффективно:

OCaml utop (part 36)

<https://github.com/realworldocaml/examples/tree/v1/code/lists-and-patterns/main.topscript>

```
# let rec ls_rec s =
  if Sys.is_file_exn ~follow_symlinks:true s
  then [s]
  else
    Sys.ls_dir s
    |> List.concat_map ~f:(fun sub -> ls_rec (s ^/ sub))
;;
val ls_rec : string -> string list = <fun>
```

Хвостовая рекурсия

Единственный способ найти длину списка в языке OCaml – выполнить обход списка от начала до конца. Как результат продолжительность такой операции находится в линейной зависимости от длины списка. Ниже приводится пример реализации функции, выполняющей эту операцию:

OCaml utop (part 37)

<https://github.com/realworldocaml/examples/tree/v1/code/lists-and-patterns/main.topscript>

```
# let rec length = function
  | [] -> 0
  | _ :: t1 -> 1 + length t1
;;
val length : 'a list -> int = <fun>
# length [1;2;3];;
- : int = 3
```

Она выглядит достаточно незамысловатой, но, применив ее к очень длинному списку, вы можете столкнуться с проблемой, как показано в следующем примере:

OCaml utop (part 38)

<https://github.com/realworldocaml/examples/tree/v1/code/lists-and-patterns/main.topscript>

```
# let make_list n = List.init n ~f:(fun x -> x);;
val make_list : int -> int list = <fun>
# length (make_list 10);;
- : int = 10
# length (make_list 10_000_000);;
Stack overflow during evaluation (looping recursion?).
```