Функциональное программирование

Ярон Мински
Анил Мадхавапедди
Джейсон Хикки

Программирование на языке OCaml

ДМК
издательство

УДК 004.6
ББК 32.973.26
М57

Мински Я., Мадхавапедди А., Хикки Дж.
М57 Программирование на языке OCaml / пер. с анг.л А. Н. Киселева. – М.: ДМК Пресс, 2014. – 536 с.: ил.

ISBN 978-5-97060-539-6

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

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

УДК 004.6
ББК 32.973.26

ISBN 978-1-449-32391-2 (анг.)

ISBN 978-5-97060-539-6 (рус.)

Содержание

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.

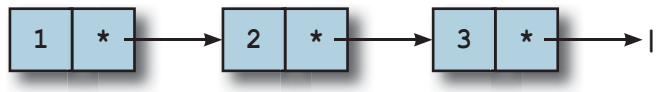

Рис. 3.1 ❖ Односвязный список как структура данных

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 :: l;;
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 образцом в инструкции match. Переменные, связанные таким способом, можно использовать в выражении справа от стрелки, следующей за текущим образцом.

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 подъязык, 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 приступать 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

OCaml utop (part 22)
https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript

```
# let s = "." ^ "." ^ "." ^ "." ^ "." ^ "." ^ ".";;
val s : string = "......."
```

создаст строки с длиной 2, 3, 4, 5, 6 и 7 символов, тогда как следующий код

OCaml utop (part 23)
https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript

```
# let s = String.concat [".";".";".";".";".";".";"."];;
val s : string = "......."
```

создаст единственную строку длиной 7 символов, а также список с 7 элементами. При таких небольших размерах разница в производительности оказывается небольшой, но при сборке длинных строк применение оператора может существенно снизить скорость выполнения программы.

Теперь нам нужно реализовать вывод одной строки в таблице. Сначала напишем функцию с именем `pad` для дополнения строки пробелами до указанной длины плюс по одному пробелу с каждой стороны:

OCaml utop (part 24)
https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript

```
# let pad s length =
    " " ^ s ^ String.make (length - String.length s + 1) ' '
  ;;
val pad : string -> int -> string = <fun>
# pad "hello" 10;;
- : string = " hello "
```

Сконструировать строку таблицы можно путем объединения отдельных строк в колонках. И снова воспользуемся функцией `List.map2_exn` для объединения списка с данными в строке таблицы со списком размеров колонок:

OCaml utop (part 25)
https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript

```
# let render_row row widths =
    let padded = List.map2_exn row widths ~f:pad in
    "|" ^ String.concat ~sep:"|" padded ^ "|"
  ;;
val render_row : string list -> int list -> string = <fun>
# render_row ["Hello";"World"] [10;15];;
- : string = "| Hello | World |"
```

Теперь можно объединить все вместе в одну функцию, осуществляющую отображение таблицы:

OCaml utop (part 26)
https://github.com/realworldocaml/examples/blob/v1/code/lists-and-patterns/main.topscript

```
# let render_table header rows =
    let widths = max_widths header rows in
```

```
    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.topscript"; "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.topscript

```
# 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.topscript

```
# [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.topscript

```
# 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.topscript

```
# 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
    | _ :: tl -> 1 + length tl
  ;;
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?).
```