# ДЕКЛАРАТИВНОЕ ПРОГРАММИРОВАНИЕ

## Лекция 7. Рекурсивные структуры данных в F#

Традиционное императивное программирование по своей идеологии близко к архитектуре современных ЭВМ. Поэтому для работы со значительными объемами одинаковых данных используется естественная структура данных – массив, аналог последовательной области памяти ЭВМ, адресация к которой производится указанием индекса массива. В функциональном программировании используется другой подход к оперированию структурами данных на основе некоторого «конструктора», позволяющего рекуррентным образом порождать последовательности элементов.

Простейшей структурой данных является список – конечная последовательность элементов одного типа. Вообще говоря, для построения списков не требуется специальная поддержка языка – их можно описать следующим образом:

```
type 't sequence = Nil | Cons of 't*'t sequence
```

Здесь `Cons` называется конструктором списка, `Nil` обозначает так называемый пустой список. С использованием такого конструктора список целых чисел 1,2,3 будет записываться как `Cons(1,Cons(2,Cons(3,Nil)))` и иметь тип `int sequence`.

Таким образом, для присоединения каждого элемента к списку используется конструктор Cons. Первый элемент списка называется его головой (head), а весь оставшийся список – хвостом (tail). Для отделения головы и хвоста легко описать соответствующие функции:

```
let head (Cons(u,v)) = u
let tail (Cons(u,v)) = v
```

Структура данных называется рекурсивной, поскольку в ее описании используется сама же структура. Действительно, приведенное выше описание sequence может быть прочитано следующим образом: список типа T – это либо пустой список Nil, либо элемент типа T (голова) и присоединенный к нему список типа T (хвост).

Для обработки такой рекурсивной структуры вполне естественно использовать рекурсивные функции. Например, для вычисления длины списка (количества элементов) можно описать функцию `len` следующим образом:

```
let rec len l =
  if l = Nil then 0
  else 1+len(tail l)
```

На самом деле библиотека F# уже содержит определение списков, которое очень похоже на приведенное выше, только в качестве пустого списка используется константа [], а конструктор списков обозначается оператором (::):

```
let 't list = [] | (::) of 't * 't list
```

С использованием такого конструктора список из чисел 1,2,3 можно записать как 1::2::3::[], или [1;2;3], а определение функции `len` будет иметь вид:

```
let rec len l =
  if l = [] then 0
  else 1+len (List.tail l)
```

На самом деле модуль `List` содержит в себе определения множества полезных функций работы со списками, многие их которых мы рассмотрим в этой главе. В частности, там содержится определение функций `head` и `tail`, а также функции `length`.

# Сопоставление с образцом

В соответствии с описанием списка каждый список может представлять из себя либо константу Nil/[], либо конструктор списка с двумя аргументами. Для распознавания того, что же представляет из себя список, мы использовали условный оператор `if`, однако еще удобнее использовать для этого сопоставление с образцом (*pattern matching*). Используя сопоставление с образцом, функция вычисления длины запишется следующим образом:

```
let rec len l =
   match l with
      [] -> 0
    | h::t -> 1+len t
```

После конструкции match следует один или более вариантов, разделенных |, на каждый из которых описывается свое поведение. В данном случае мы используем всего два варианта сопоставления, хотя их может быть больше. Напомним, что, помимо простого сопоставления, можно также использовать более сложные условные выражения, как в примере ниже, в котором мы описываем функцию суммирования положительных элементов списка:

```
let rec sum_positive l =
   match l with
      [] -> 0


    | h::t when h>0 -> h+sum_positive t
    | _::t -> sum_positive t
```

Этот пример демонстрирует также две особенности оператора match. Во-первых, шаблоны сопоставления проверяются в порядке следования, поэтому с последним шаблоном будут сопоставлены только случаи, в которых голова списка меньше или равна 0. Во-вторых, если значение какой-то части шаблона нам не важно, можно использовать символ подчеркивания _ для обозначения анонимной переменной.

Сопоставление с образцом работает не только в конструкции match, но и внутри сопоставления имен `let` и в конструкции описания функциональных констант `function`. Конструкция `function` аналогична `fun`, в отличие от нее, позволяет описывать только функции одного аргумента, но поддерживает сопоставление с образцом. При описании функций обработки рекурсивных структур данных часто удобно использовать `function`, например:

```
let rec len = function
   [] -> 0
 | _::t -> 1+len t
```

Модуль List содержит основные функции для работы со списками, в частности описанную нами функцию определения длины списка List.length. Из других функций, заслуживающих внимания, стоит отметить функцию конкатенации списков List.append, которая также может обозначаться как @, например:

```
List.append [1;2] [3;4]
[1;2]@[3;4]
```

Традиционно функция конкатенации определяется следующим образом:

```
let rec append l r =
  match l with
    [] -> r
  | h::t -> h::(append t r)
```

Из этого определения видно, что функция является рекурсивной по первому аргументу, и, значит, для объединения списков длины $L_1$ и $L_2$ элементов потребуется $O(L_1)$ операций. Такая сложная оценка операции конкатенации является следствием способа представления списков с помощью конструктора, в результате чего для составления списка-результата конкатенации мы вынуждены разбирать первый список поэлементно и затем присоединять эти элементы ко второму списку поочередно.

Для доступа к произвольному элементу списка по номеру может использоваться функция List.nth, которую также можно вызывать с помощью специального синтаксиса индексатора. Для доступа ко второму элементу списка (который имеет номер 1, поскольку нумерация идет с 0) можно использовать любое из следующих выражений:

```
List.nth [1;2;3] 1
[1;2;3].Item(1)
[1;2;3].[1]
```

Следует опять же помнить, что сложность такой операции – $O(n)$, где $n$ – номер извлекаемого элемента.

# Функции высших порядков

Рассмотрим основные операции, которые обычно применяются к спискам. Подавляющее большинство сложных операций обработки сводятся к трем базовым операциям: отображения, фильтрации и свертки. Поскольку такие функции в качестве аргументов принимают другие функции, работающие над каждым из элементов списка, то они называются *функционалами*, или функциями высших порядков.

Операция отображения map применяет некоторую функцию к каждому элементу некоторого списка, возвращая список результирующих значений. Если функция-обработчик имеет тип 'a->'b, то map применяется к списку типа 'a list и возвращает 'b list. Соответственно, сама функция map имеет тип ('a->'b) -> 'a list -> 'b list. Определена она может быть[1] следующим образом (естественно, модуль List определяет соответствующую функцию List.map):

```
let rec map f = function
    [] -> []
  | h::t -> (f h)::(map f t)
```

Спектр использования функции map очень широк. Например, если необходимо умножить на 2 все элементы целочисленного списка, это можно сделать одним из следующих способов:

```
map (fun x -> x*2) [1;2;3]
map ((*)2) [1;2;3]
[ for x in [1;2;3] -> x*2 ]
```

Другой пример – пусть нам необходимо загрузить содержимое нескольких веб-сайтов из Интернета, например с целью дальнейшего поиска. Предположим, у нас определена функция http, которая по адресу странички сайта (URL) возвращает ее содержимое[1]. Тогда осуществить загрузку всех сайтов из Интернета можно будет следующим образом:

```
["http://www.bing.com";"http://www.yandex.ru"] |> List.map http
```

Напоминаем, что здесь мы используем операцию последовательного применения функций |> (*pipeline*), которая позволяет последовательно применять цепочки

Иногда бывает полезно, чтобы функции обработки передавался номер обрабатываемого элемента списка. Для этого предусмотрена специальная функция List.mapi, которая принимает функцию обработки с двумя аргументами, один из которых – номер элемента в списке, начиная с 0. Например, если нам надо получить из списка строк пронумерованный список, это можно сделать так:

```
["Говорить";"Читать";"Писать"]
    |> List.mapi (fun i x -> (i+1).ToString()+". "+x)
```

В качестве чуть более сложного примера рассмотрим функцию «наивного» перевода списка цифр в некоторой системе счисления в число. Например, число 1000 можно представить как [1;0;0;0], и в двоичной системе оно будет обозначать 1  103 + 0  102 + 0  101 + 0  100 = 8. Таким образом, чтобы получить значение числа, нам надо умножать каждую цифру на основание системы счисления в степени, равной позиции цифры от конца числа. Для достижения этого проще всего сначала перевернуть (записать в обратном порядке) список цифр, после чего применить `mapi` для умножения цифр на возведенное в степень основание, далее сложить результат с помощью функции List.sum:

```
let conv_to_dec b l =
   List.rev l |>
   List.mapi (fun i x -> x*int(float(b)**float(i))) |>
   List.sum
```

Также в библиотеке определены функции попарного отображения двух списков с получением одного результата List.map2, на основе которых легко определить, например, сложение векторов, представленных списками:

```
List.map2 (fun u v -> u+v) [1;2;3] [4;5;6]
List.map2 (+) [1;2;3] [4;5;6]
```

С помощью map2 также можно определить функцию conv_to_dec:

```
let conv_to_dec b l =
  [ for i = (List.length l)-1 downto 0 do yield int(float(b)**float(i)) ]
  |> List.map2 (*) l |> List.sum
```

В этом случае мы сначала явно порождаем список из степеней основания системы счисления, а потом попарно умножаем его на цифры числа, затем суммируя результат.

Также в библиотеке определены функции для «тройного» отображения List.map3, двойного отображения с индексацией List.mapi2 и др.

Бывают ситуации, когда для каждого элемента списка нам полезно рассмотреть несколько альтернатив. Например, пусть у нас есть список сайтов, для которых мы хотим получать странички contact.html и about.html. Первая попытка реализовать это будет выглядеть следующим образом:

```
[ "http://site1.com"; "http://site2.com"; "http://site3.com" ] |>
  List.map (fun url ->
[ http (url+"/about.html"); http (url+"/contact.html")])
```

Однако в этом случае в результате будет получен не список содержимого всех страничек (что было бы удобно для реализации поисковой системы), а список списков – для каждого сайта будет возвращен список из двух страничек. Чтобы объединить все результирующие списки в один, придется использовать в явном виде функцию конкатенации списка списков:

```
[ "http://site1.com"; "http://site2.com"; "http://site3.com" ] |>
  List.map (fun url ->
[ http (url+"/about.html"); http (url+"/contact.html")])
  |> List.concat
```

Однако намного более эффективно сразу использовать вместо `map` функцию `collect`, которая применяет заданную функцию к каждому элементу исходного списка и затем объединяет вместе возвращаемые этими функциями списки:

```
[ "http://site1.com"; "http://site2.com"; "http://site3.com" ] |>
 List.collect (fun url ->
     [ http (url+"/about.html"); http (url+"/contact.html")])
```

Фильтрация позволяет нам оставить в списке только элементы, удовлетворяющие заданной функции-фильтру. Например, для выделения только четных элементов списка чисел от 1 до 10 можно использовать:

```
[1..10] |> List.filter (fun x -> x%2=0)
```

Если `filter` применяется к списку типа `'t list`, то функция фильтрации должна иметь тип 't -> bool. В результирующий список типа 't list попадают только элементы, для которых функция принимает истинное значение. Функция фильтрации может быть реализована следующим образом:

```
let rec filter f = function
    [] -> []
  | h::t when (f h) -> h::(filter f t)
  | _::t -> filter f t
```

Вот как можно легко использовать `filter` для реализации простейшей поисковой системы в фиксированном множестве сайтов:

```
["http://www.bing.com";"http://www.yandex.ru"] |>
   List.map http |>
   List.filter (fun s -> s.IndexOf("bing")>0)
```

В качестве более сложного примера рассмотрим вычисление простых чисел в интервале от 2 до некоторого числа *N*. Для этого используется алгоритм, известный как **решето Эратосфена**. Он состоит в следующем: выписываем все числа от 2 до *N*, после чего применяем к ним многократно одинаковую процедуру: объявляем первое из написанных чисел простым, а из оставшихся вычеркиваем все числа, кратные данному. После чего процедура повторяется. В результате в списке у нас остаются только простые числа.

Для реализации этого алгоритма опишем функцию `primes`, которая применяется к списку от 2 до *N*. Эта функция будет рекурсивно реализовывать каждый шаг алгоритма Эратосфена:

```
let rec primes = function
    [] -> []
  | h::t -> h::primes(filter (fun x->x%h>0) t)
```

На каждом шаге первое число в списке h объявляется простым (то есть входит в результирующий список), а к остальным применяется функция фильтрации, которая вычеркивает из списка оставшихся чисел все, кратные h.

Еще один интересный пример – быстрая сортировка Хоара. Алгоритм быстрой сортировки состоит в том, что на каждом шаге из списка выбирается некоторый элемент и список разбивается на две части – элементы, меньшие или равные выбранному и большие выбранного. Затем сортировка рекурсивно применяется к обоим частям списка. С использованием `filter`, выбирая первый элемент списка в качестве элемента для сравнения, мы получим следующую реализацию:

```
let rec qsort = function
   [] -> []

| h::t ->
    qsort(List.filter ((>)h) t) @ [h] @
    qsort(List.filter ((<=)h) t)
```

Эта же реализация может быть записана более наглядно, с использованием конструктора списков для фильтрации:

```
let rec qsort = function
   [] -> []
 | h::t ->
    qsort([for x in t do if x<=h then yield x]) @ [h]
  @ qsort([for x in t do if x>h then yield x])
```

Мы видим, что в данном случае мы, по сути, с помощью операции фильтрации разбиваем список на две части в соответствии с некоторым предикатом – при этом две операции фильтрации требуют двух проходов по списку. Чтобы сократить число проходом, можно воспользоваться функцией partition, возвращающей пару списков – из элементов, удовлетворяющих предикату фильтрации и всех остальных:

```
List.partition ((>)0) [1;-3;0;4;3]   => ([-3],[1;0;4;3])
```

С учетом этой функции быстрая сортировка запишется следующим образом:

```
let rec qsort = function
   [] -> []
 | h::t ->
     let (a,b) = List.partition ((>)h) t
     qsort(a) @ [h] @ qsort(b)
```

Еще одной альтернативой функции `filter`, объединяющей ее с отображением `map`, является функция `choose`, которая для каждого элемента возвращает опциональный тип и собирает только те результаты, которые не равны `None`. В частности:

```
let filter p = List.choose (fun x -> if p x then Some(x) else None)
let map f = List.choose (fun x -> Some(f x))
```

Операция свертки применяется тогда, когда необходимо получить по списку некоторый интегральный показатель – минимальный или максимальный элемент, сумму или произведение элементов и т. д. Свертка является заменой циклической обработки списка, в которой используется некоторый аккумулятор, на каждом шаге обновляющийся в результате обработки очередного элемента.

Поскольку в функциональном программировании нет переменных, то традиционное решение с аккумулятором невозможно. Вместо этого используется в явном виде передаваемое через цепочку функций значение – состояние. Функция свертки будет принимать на вход это значение и очередной элемент списка, а возвращать – новое состояние. Таким образом, функция `fold`, примененная к списку $[a_1;...;a_n]$, будет вычислять $f(...f(f(s_0,a_1),a_2),...,a_n)$, где $s_0$ – начальное значение аккумулятора.

В качестве аккумулятора для вычисления суммы элементов списка будет выступать обычное числовое значение, которое мы будем складывать с очередным элементом:

```
let sum L = List.fold (fun s x -> s+x) 0 L
```

Поскольку функция, передаваемая fold, представляет собой обычное сложение, то мы можем записать то же самое короче:

```
let sum = List.fold (+) 0
let product = List.fold (*) 1
```

Здесь мы также определяем функцию произведения элементов списка. Для вычисления минимального и максимального элементов списка за один проход мы можем использовать состояние в виде пары:

```
let minmax L =
   let a0 = List.head L in
   List.fold (fun (mi,ma) x ->
               ((if mi>x then x else mi),
                (if ma<x then x else ma)))
             (a0,a0) L
let min L = fst (minmax L)
let max L = snd (minmax L)
```

Описанная нами операция свертки называется также левой сверткой, поскольку применяет операцию к элементам списка слева направо. Также имеется операция правой, или обратной, свертки `List.foldBack`, которая вычисляет $f(a_1 f(a_2,\ldots,f(a_n,s_0)\ldots)$.

Наш пример с функцией-минимаксом с помощью обратной свертки запишется так:

```
let minmax L =
   let a0 = List.head L in
   List.foldBack (fun x (mi,ma) ->
               ((if mi>x then x else mi),
                (if ma<x then x else ma)))
            L (a0,a0)
```

Обратите внимание, что функция *f* в данном случае имеет тип 't→State→State (то есть сначала идет элемент списка, а затем – состояние) и что порядок аргументов у функции `foldBack` другой. Типы функций `fold` и `foldBack` следующие:

- fold: (State→'t →State) →State→'T list→State
- foldBack: ('t→State→State) →'T list→State→State

Для вычисления минимального и максимального элементов нам приходилось в явном виде использовать первый элемент списка в качестве начального состояния, из-за чего функция получилась несколько громоздкой. Альтернативно, если нам необходимо определить лишь функцию минимального или максимального элемента, мы можем воспользоваться редукцией списка `List.reduce`, которая применяет некоторую редуцирующую функцию $f$ типа 'T→'T→'T попарно сначала к первым двум элементам списка, затем к результату и третьему элементу списка и так далее до конца, вычисляя $f(f(...f(a_3 f(a_1,a_2))...)$. С помощью редуцирования мы получим простое определение:

```
let min : int list -> int = List.reduce (fun a b -> Math.Min(a,b))
```

Здесь нам пришлось описать тип функции min в явном виде, поскольку иначе система вывода типов не может правильно выбрать необходимый вариант полиморфной функции Min из библиотеки .NET. Если же функция минимума определена как каррированная F#-функция, то определение будет еще проще:

```
let minimum a b = if a>b then b else a
let min L = List.reduce minimum L
```

В библиотеке F# есть также множество функций, которые используются не так часто, как рассмотренные выше, но которые полезно упомянуть. Для простого итерирования по списку могут использоваться функция `iter` и ее разновидности `iteri` и `iter2`, например:

```
List.iteri (fun n x -> printf "%d. %s\n" (n+1) x) ["Раз";"Два";"Три"]
List.iter2 (fun n x -> printf "%d. %s\n" n x) [1;2;3] ["Раз";"Два";"Три"]
```

Для поиска элемента в списке по некоторому предикату используются функции `find` и `tryFind`. Первая из них возвращает найденный элемент и генерирует исключение, если элемент не найден; вторая возвращает опциональный тип, то есть `None`, в случае если элемент не найден. Аналогичные функции `findIndex` и `tryFindIndex` возвращают не сам элемент, а его порядковый номер.

Из других функций, работающих с предикатами, упомянем `exists` и `forall` (а также их варианты `exists2` и `forall2`), проверяющие, соответственно, истинность предиката на хотя бы одном или на всех элементах списка. Эти функции могут быть легко определены через свертку:

```
let exists p = List.fold (fun a x -> a || (p x)) false
let for_all p = List.fold (fun a x -> a && (p x)) true
```

Функции `zip/unzip` позволяют объединять два списка в список пар значений и, наоборот, из списка пар получать два списка. Есть их версии `zip3/unzip3` для троек значений.

Имеется также целый спектр функций для сортировки списка. Обычная `sort` сортирует список в соответствии с операцией сравнения, определенной на его элементах. Если необходимо сортировать элементы в соответствии с некоторым другим критерием, то можно либо задать этот критерий явно (`sortWith`), либо задать функцию генерации по каждому элементу некоторого индекса, в соответствии с которым осуществлять сортировку (`sortBy`). Вот как с помощью этих функций можно отсортировать список слов по возрастанию длины:

```
["One";"Two";"Three";"Four"] |> List.sortBy(String.length)
["One";"Two";"Three";"Four"] |>
   List.sortWith(fun a b -> a.Length.CompareTo(b.Length))
```

Из арифметических операций над списками определены операции min/max (minBy/maxBy), sum/sumBy и average/averageBy – например, вот так мы могли бы найти строку с максимальной длиной и среднюю длину строк в списке:

```
["One";"Two";"Three";"Four"] |> List.maxBy(String.length)
["One";"Two";"Three";"Four"] |> List.averageBy(fun s -> float(s.Length))
```

В заключение упомянем функцию permute, которая умеет применять к списку перестановку, заданную целочисленной функцией:

```
List.permute (function 0->0 | 1->2 | 2->1) [1;2;3]
```

Мы уже ранее встречались с конструкциями, которые позволяли создавать список путем задания диапазона элементов, или некоторой функции генерации списка. Все эти конструкции укладываются в единообразный синтаксис генераторов списков, начинающийся с квадратной скобки. Списки могут задаваться:

- явным перечислением элементов: `[1;2;3]`;
- заданием диапазона значений: `[1..10]`. В этом случае для создания списка на самом деле вызывается оператор диапазона (..). Его можно использовать и в явном виде, например:

```
let integers n = (..) 1
```

- заданием диапазона и шага инкремента: `[1.1..0.1..1.9]` или `[10..-1..1]`;
- заданием генерирующей функции: `[ for x in 0..8 -> 2**float(x) ]`. Этот пример можно также записать в виде явного вызова функции List.init следующим образом: `List.init 9 (fun x -> 2.0**float(x))`, либо же в виде отображения `[0..8] |> List.map (fun x -> 2.0**float(x))`;

- заданием более сложного алгоритма генерации элементов списка. В этом случае внутри скобок могут использоваться любые комбинации из операторов `for`, `let`, `if` и др., а для возврата элементов используется конструкция `yield`:

```
[ for a in -3.0..3.0 do
    for b in -3.0..3.0 do
       for c in -3.0..3.0 do
         let d = b*b-4.*a*c
         if a<>0.0 then
           if d<0.0 then yield (a,b,c,None,None)
           else yield
                (a,b,c,
                 Some((-b-Math.Sqrt(d))/2./a),
                 Some((-b+Math.Sqrt(d))/2./a)) ]
```

# Хвостовая рекурсия

Вернемся снова к рассмотрению простейшей функции вычисления длины списка:

```
let rec len = function
   [] -> 0
 | h::t -> 1+len t
```

Посмотрим на то, как эта функция вычисляется, например для списка [1;2;3]. Вначале от списка отделяется хвост, и рекурсивно вызывается len [2;3] – происходит рекурсивное погружение. На следующем уровне рекурсии вызывается len [3] и наконец len [], которая возвращает 0, – после чего происходит «всплывание» из рекурсии, для вычисления len [3] к 0 прибавляется 1, затем еще 1, и наконец вызванная функция завершается, возвращая результат – 3. Схематически процесс рекурсивного вызова len [1;2;3] изображен на рис. 2.1.

На каждом уровне рекурсии для рекурсивного вызова необходимо запомнить в стеке адрес возврата, параметры функции и возвращаемый результат – то есть такое определение len требует для своей работы $O(n)$ ячеек памяти. С другой стороны, очевидно, что для вычисления длины списка при императивном программировании не требуются дополнительные расходы памяти. Если бы на функциональном языке было невозможно совершать такие простые итеративные операции без расхода памяти, это было бы крайне негативной стороной, сводящей на нет многие преимущества.

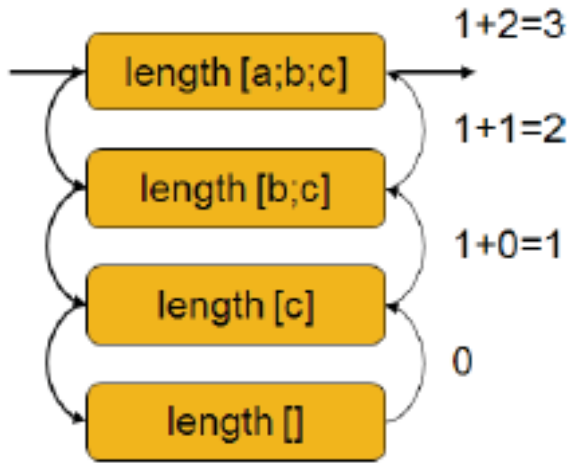


*Рис. 2.1*

К счастью, алгоритмы, реализуемые в императивных языках при помощи итерации, могут быть эффективно вычислены в функциональном подходе при помощи так называемой *хвостовой рекурсии* (tail recursion). Суть хвостовой рекурсии сводится к тому, что в процессе рекурсии не выделяется дополнительная память, а рекурсивный вызов является последней операцией в процессе вычисления функции. В этом случае возможно сразу после вычисления функции перейти к следующему рекурсивному вызову без запоминания адреса возврата, то есть компилятор может распознать такую рекурсию и преобразовать ее к обычному циклу – сохранив при этом все преимущества рекурсивного определения.

Чтобы вычисление длины списка производилось без дополнительных расходов памяти, преобразуем функцию таким образом, чтобы прибавление единицы к длине происходило до рекурсивного вызова. Для этого введем счетчик – текущую длину списка – и каждый рекурсивный вызов будет сначала увеличивать счетчик и потом вызывать функцию с увеличенным счетчиком в качестве параметра:

```
let rec len a = function
   [] -> a
 | _::t -> len (a+1) t
```

При этом рекурсивный вызов располагается в конце вызова функции – поэтому адрес возврата не запоминается, а совершаются по сути циклические вычисления, как показано на рис. 2.2.

Чтобы вычислить длину списка, надо вызывать функцию с нулевым значением счетчика:

```
len 0 [1;2;3]
```

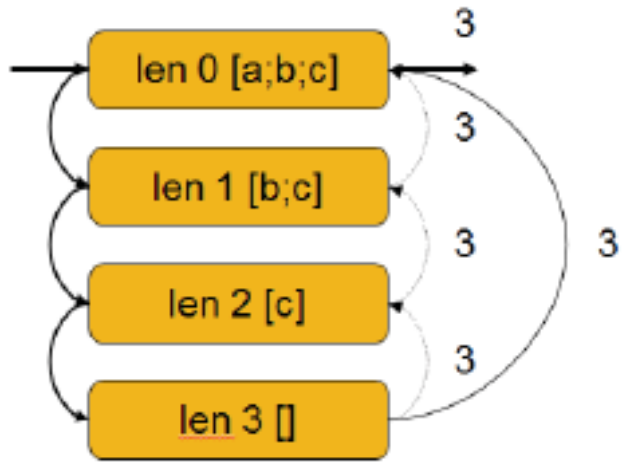


*Рис. 2.2*

Понятно, что программисту, использующему функцию `len`, нет нужды знать про внутреннее устройство функции, поэтому правильно будет спрятать особенности реализации внутрь вложенной функции:

```
let len l =
  let rec len_tail a = function
     [] -> a
   | _::t -> len_tail (a+1) t
  len_tail 0 l
```

Другим примером, когда хвостовая рекурсия позволяет сильно оптимизировать решение, является реверсирование списка. Исходя из декларативных соображений, простейший вариант реверсирования может быть записан следующим образом:

```
let rec rev = function
  [] -> []
| h::t -> (rev t)@[h]
```

Если задуматься над сложностью такого алгоритма, то окажется, что она равна $O(n^2)$, поскольку в реверсировании используется операция `append`, имеющая линейную сложность. С другой стороны, алгоритм реверсирования может быть легко сформулирован итерационно: необходимо отделять по одному элементу с начала исходного списка и добавлять в начало результирующего – в этом случае последний элемент исходного списка как раз окажется первым в результате.

Для реализации такого алгоритма мы определим функцию `rev_tail`, которая в качестве первого аргумента будет принимать результирующий список, а вторым аргументом будет исходный список. На каждом шаге мы будет отделять голову исходного списка и добавлять его в начало первого аргумента при рекурсивном вызове. Когда исходный список исчерпается и станет пустым – мы вернем первый аргумент в качестве результата. При ближайшем рассмотрении также можно увидеть, что функция `rev_tail` использует хвостовую рекурсию.

```
let rev L =
   let rec rev_tail s = function
     [] -> s
   | h::t -> rev_tail (h::s) t in
   rev_tail [] L
```

Таким образом, мы видим, что декларативные реализации функций часто оказываются очень наглядными, но с точки зрения эффективности не самыми лучшими. Поэтому разработчику, пишущему код в функциональном стиле, стоит взять в привычку задумываться над особенностями выполнения кода, в частности постараться, где возможно, использовать хвостовую рекурсию. Основным сигналом к тому, что хвостовая рекурсия возможна, является итерационный алгоритм обработки, который при императивной реализации не требовал бы дополнительной памяти. При некотором навыке преобразование таких рекурсивных алгоритмов к хвостовой рекурсии станет механическим упражнением.