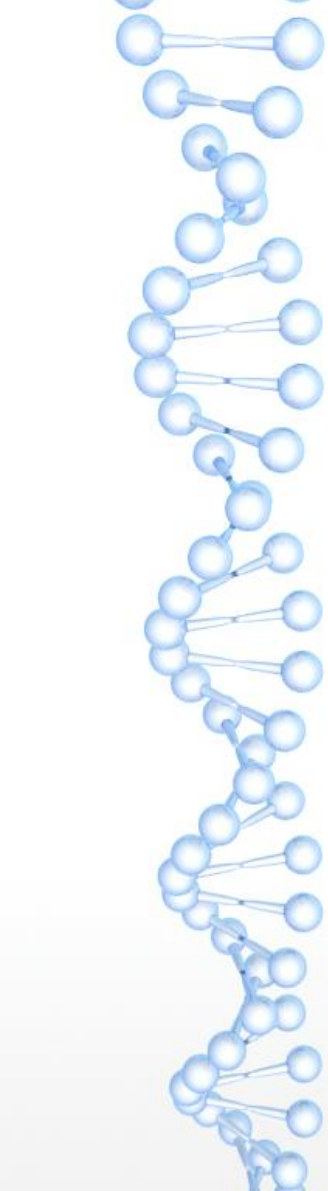
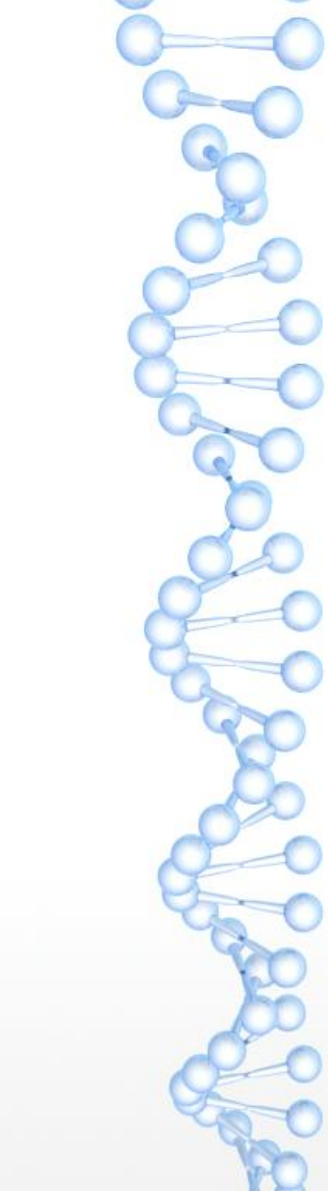


Функциональное и логическое программирование

Лекция 6. Функциональное программирование



Программирование с помощью функций

Язык, претендующий на звание «функционального», обычно должен обладать следующими ключевыми особенностями:

- Неизменяемость данных
- Возможность композиции функций
- Функции могут рассматриваться как данные
- Отложенные (lazy) вычисления
- Сопоставления с образцом (pattern matching)

Программирование с помощью функций

В основе функционального программирования лежит представление о коде в терминах математических функций. Рассмотрим две функции, f и g:

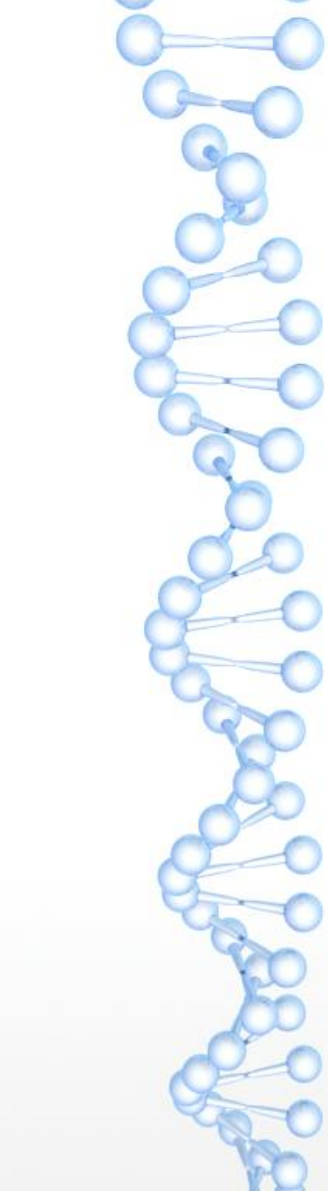
```
f(x) = x^2 + x
g(x) = x + 1
```

Из этого следует, что:

```
f(2) = (2)^2 + (2)
g(2) = (2) + 1
```

И если теперь объединить эти две функции, получим следующее:

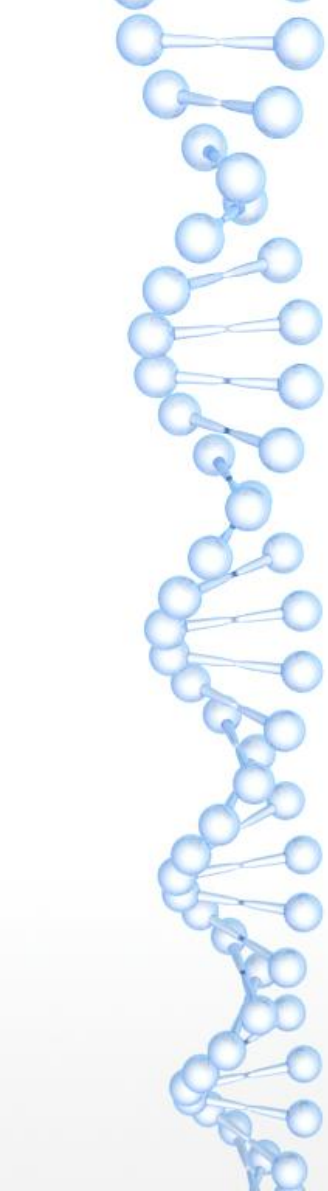
```
f g (2) = f(g(2))
        = (g(2))^2 + (g(2))
        = (2+1)^2 + (2+1)
        = 12
```



Программирование с помощью функций

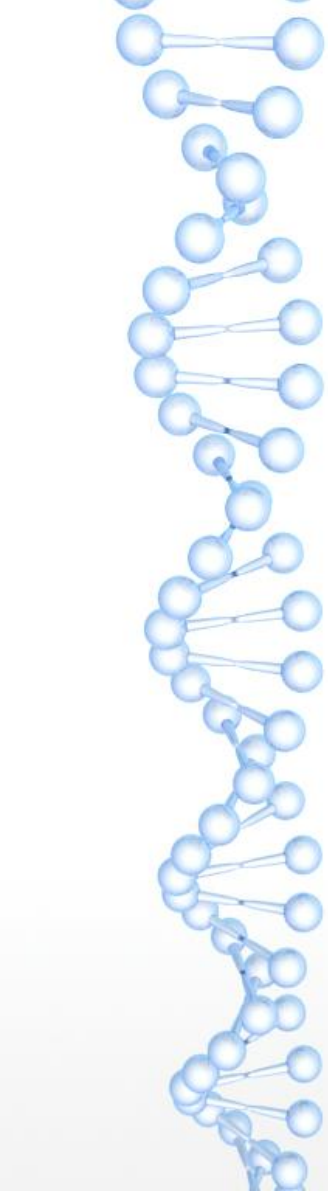
То, что этот код напоминает математическую нотацию, не случайное совпадение. По сути функциональное программирование как раз и заключается в представлении вычислений подобным абстрактным способом. Напомню еще раз, что в функциональном программировании программист описывает, *что* требуется сделать, а не *как* это сделать.

Вы можете даже представить себе всю программу целиком, как единую функцию, которая на входе принимает сигналы от мыши и клавиатуры, а на выходе возвращает код завершения приложения. Когда программирование начинает рассматриваться с этой точки зрения, уходят многие сложности, присущие обычным моделям программирования.



Программирование с помощью функций

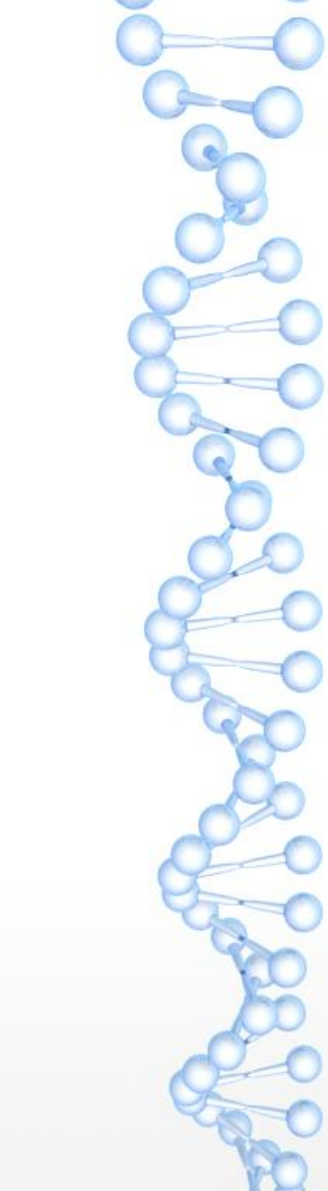
Во-первых, если представить себе программу как последовательность функций, то не придется тратить все свое время, чтобы во всех подробностях описать каждый шаг, необходимый для решения задачи. Функции просто принимают входные данные и возвращают некоторый результат. Во-вторых, алгоритмы обычно выражаются в терминах функций, а не классов или объектов, поэтому их проще выражать в функциональном стиле.



Неизменяемость

Вероятно, вы заметили, что ранее я нигде не использовал термин *переменная*, и вместо него я использовал термин *значение*. Причина заключается в том, что в функциональном программировании имена, которые объявляются в программе, по умолчанию являются неизменяемыми, в том смысле, что их нельзя изменить.

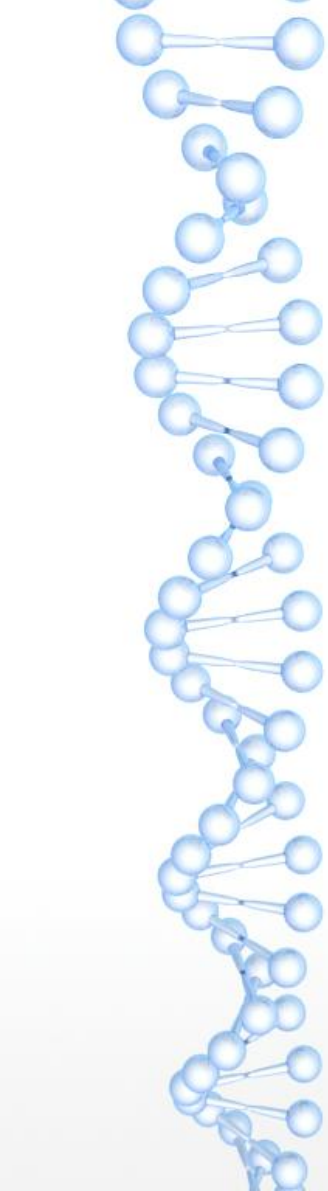
Если функция каким-то образом изменяет состояние программы, например записывает данные в файл или изменяет значение глобальной переменной в памяти, это называют *побочным эффектом*. Например, функция `printfn` возвращает результат типа `unit`, но она содержит побочный эффект в виде вывода текста на экран. Аналогично если функция изменяет некоторое значение в памяти, это также является побочным эффектом – тем, что делает функция помимо вычисления возвращаемого значения.



Неизменяемость

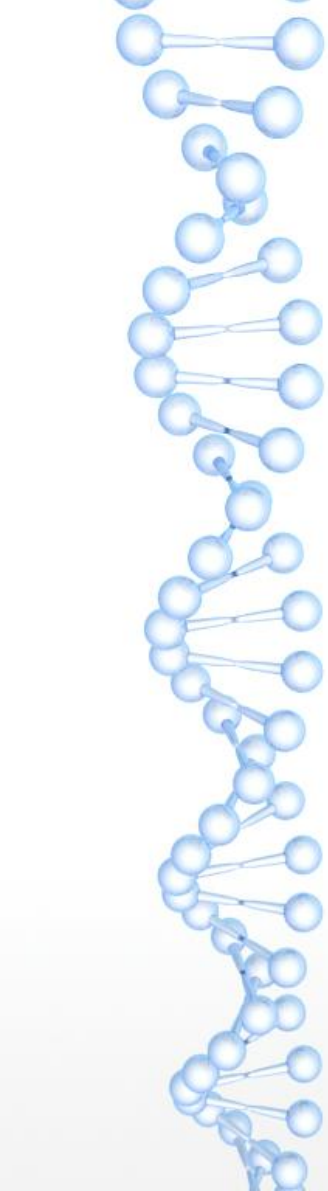
Побочные эффекты – это не всегда плохо, но непредусмотренные побочные эффекты являются источником многих ошибок. Даже с самыми благими намерениями можно допустить ошибку, если не задумываться о побочных эффектах функции. Неизменяемые значения помогают писать надежный код, потому что нельзя испортить то, что невозможно изменить.

Если у вас есть опыт использования императивных языков программирования, отсутствие переменных может показаться вам весьма неудобным. Тем не менее неизменяемость имеет свои преимущества. Рассмотрим пример 3.1, в котором обе функции просто суммируют квадраты списка чисел, но одна функция написана в императивном стиле и изменяет данные, а другая написана в функциональном стиле. Императивная функция использует *изменяемую* переменную, то есть значение `total` изменяется в процессе выполнения функции `imperativeSum`.



Неизменяемость

```
let square x = x * x

let imperativeSum numbers =
    let mutable total = 0
    for i in numbers do
        let x = square i
        total <- total + x
    total

let functionalSum numbers =
    numbers
    |> Seq.map square
    |> Seq.sum
```

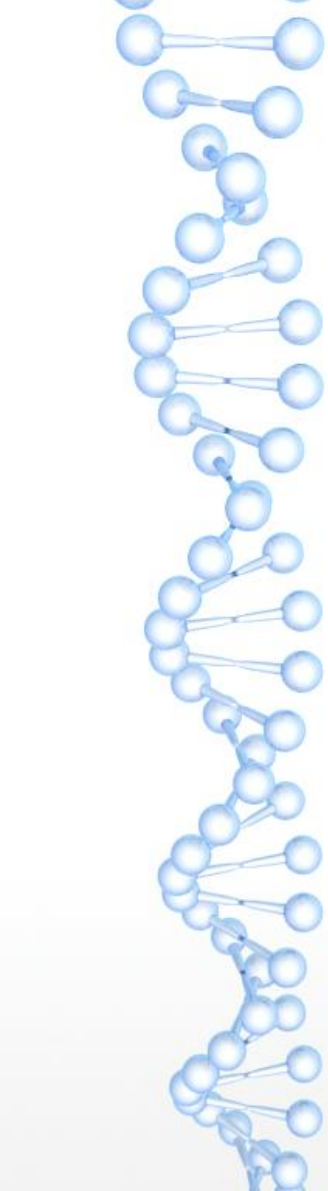


Функции как значения

В примере 3.2 определяется функция `negate`, которая изменяет знак одного целого числа. Если передать ее в функцию `List.map` в виде параметра, она будет применена ко всему списку и изменит знак каждого его элемента.

Пример 3.2. Пример функций высшего порядка

```
> let negate x = -x;;

val negate : int -> int

> List.map negate [1 .. 10];;
val it : int list = [-1; -2; -3; -4; -5; -6; -7; -8; -9; -10]
```



Функции как значения

Использовать функции как значения очень удобно, но в конечном результате вам придется написать множество простых функций, которые сами по себе не имеют большой ценности. Например, наша функция `negate` из примера 3.2 едва ли будет использоваться еще для каких-либо целей, кроме как для изменения знаков элементов списка.

Вместо того чтобы давать имена всем этим мелким функциям, которые передаются в виде параметров, можно использовать *анонимные функции*, которые также известны как *лямбда-выражения*.

Чтобы создать лямбда-выражение, достаточно воспользоваться ключевым словом `fun`, за которым следуют параметры функции и стрелка `->`.

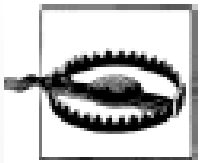
Функции как значения

В следующем фрагменте создается лямбда-выражение, которому в виде параметра передается значение 5. При вызове этой функции она увеличит значение параметра x на 3 и вернет 8:

```
> (fun x -> x + 3) 5;;
val it : int = 8
```

Теперь мы можем переписать наш пример, изменяющий знаки чисел в списке, с использованием лямбда-выражения, которое принимает единственный параметр i:

```
> List.map (fun i -> -i) [1 .. 10];;
val it : int list = [-1; -2; -3; -4; -5; -6; -7; -8; -9; -10]
```

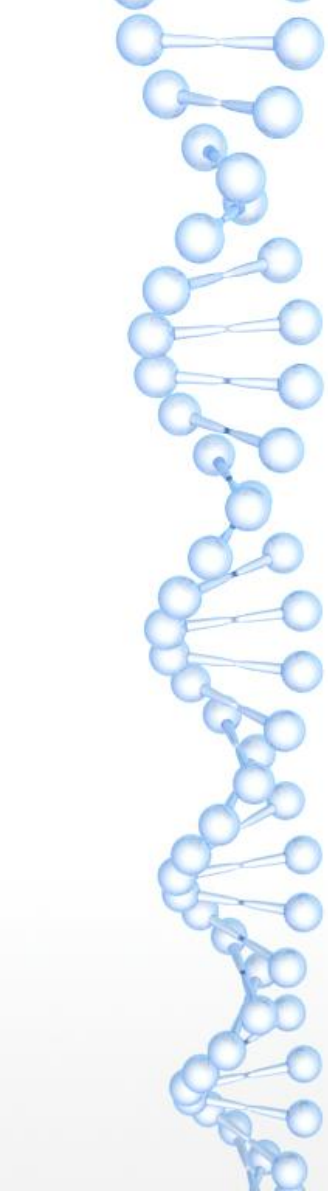


Старайтесь создавать лямбда-выражения максимально простыми. Чем длиннее лямбда-выражение вы напишете, тем сложнее его будет отлаживать. Это особенно справедливо когда вы начинаете копировать лямбда-выражения по всему своему коду.

Частичное применение функции

Еще одна характерная особенность функционального программирования – частично применимые функции, известные также как *каррированные функции (carrying)*. Начнем обсуждение этой темы с создания простой функции, которая дописывает текст в конец файла, используя существующие библиотеки .NET:

```
> // Добавить текст в конец файла
open System.IO

let appendFile (fileName : string) (text : string) =
    use file = new StreamWriter(fileName, true)
    file.WriteLine(text)
    file.Close();;

val appendFile : string -> string -> unit

> appendFile @"D:\Log.txt" "Processing Event X...";;
val it : unit = ()
```



Частичное применение функции

Функция `appendFile` выглядит достаточно просто, но что если нам потребуется многократно выполнять запись в один и тот же журнальный файл? Можно было бы сохранить путь к файлу и всегда передавать его в первом параметре. Однако было бы лучше создать версию функции `appendFile`, первый параметр которой имеет фиксированное значение `@"D:\Log.txt"`.

Создание частично применимой функции означает возможность задать значения некоторых параметров функции и создать новую функцию с фиксированными значениями параметров. Мы можем «каррировать» (зафиксировать) первый параметр функции `appendFile` и создать новую функцию, принимающую единственный параметр – текст сообщения для вывода в журнальный файл:

Частичное применение функции

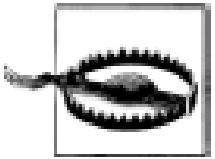
```
> // Каррировать appendFile, зафиксировав значение первого параметра
let curriedAppendFile = appendFile @"D:\Log.txt";;

val curriedAppendFile : (string -> unit)

> // Добавить текст в файл 'D:\Log.txt'
curriedAppendFile "Processing Event Y...";;
val it : unit = ()
```

Карринг объясняет наличие в типе функции стрелки между ее аргументами. Функция `appendFile` имела тип:

```
string -> string -> unit
```

так что после передачи первого параметра типа `string` получается функция, которая принимает параметр типа `string` и возвращает значение типа `unit`, то есть `string -> unit`.

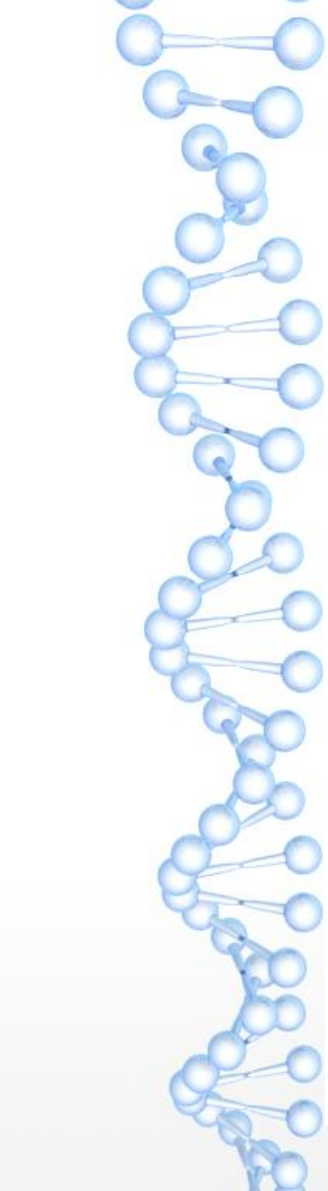
Частичное применение функции



Каррированные функции могут упростить код, но они также могут усложнить его отладку. Старайтесь не злоупотреблять каррингом, чтобы не сделать программу более сложной, чем это необходимо.

Все, что необходимо для каррирования функции, – это указать подмножество параметров; в результате вы получите функцию, которой требуется передать только параметры, которые не были указаны до этого. По этой причине каррировать параметры можно только слева направо.

Карринг и частично применимые функции не выглядят особенно мощным инструментом, но они могут существенно повысить элегантность вашего кода. Представьте вызов функции `printf`, которой передается строка формата и набор данных, соответствующих этой строке. Если создать каррированную версию, первый параметр которой имеет фиксированное значение `"%d"`, в результате мы получим частично применимую функцию, которая принимает целое число и выводит его на экран.

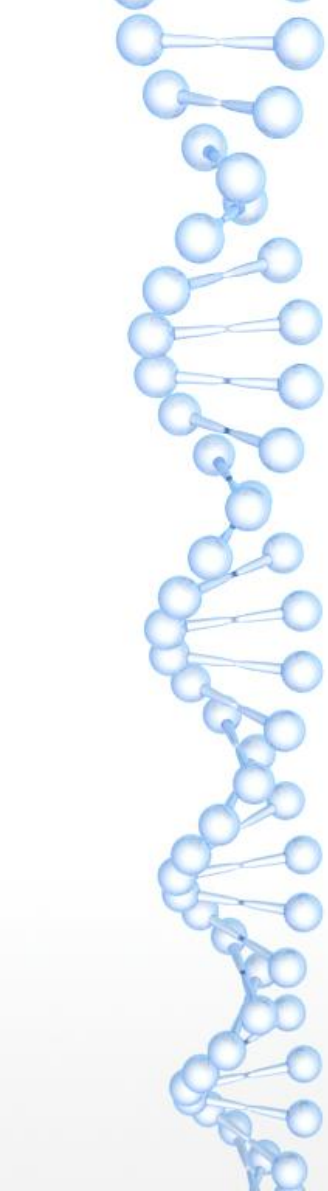


Частичное применение функции

В следующем примере демонстрируется, как можно передать частично применимую версию функции `printf`, чтобы избежать необходимости создавать лямбда-выражение:

```
> // Некаррированная версия
List.iter (fun i -> printfn "%d" i) [1 .. 3];;
1
2
3
val it : unit = ()
> // Использование каррированной версии printfn
List.iter (printfn "%d") [1 .. 3];;
1
2
3
val it : unit = ()
```

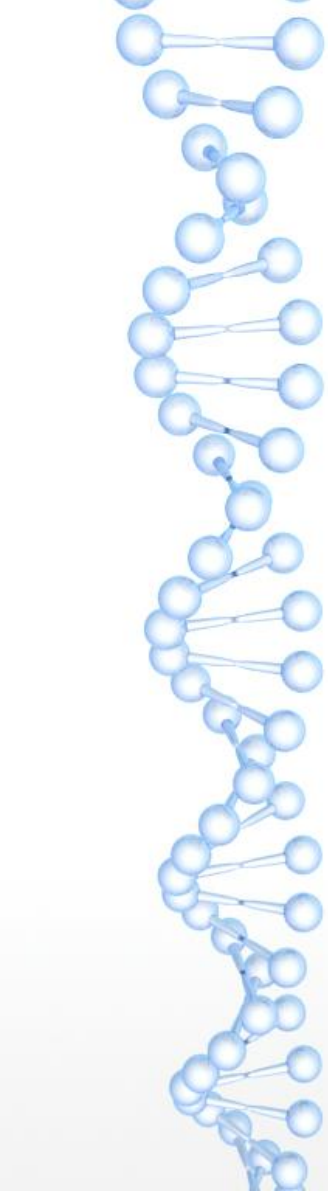
Как полностью использовать все преимущества частично применимых функций, вы увидите, когда мы будем обсуждать композицию функций и конвейерный оператор (pipe-forward operator).



Функции, возвращающие функции

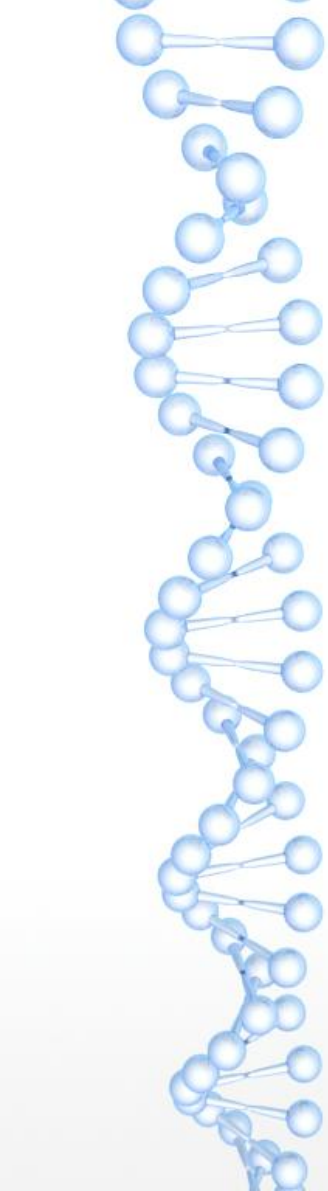
Благодаря тому что в функциональном программировании функции интерпретируются как данные, функции могут возвращать другие функции как обычные значения. Это может приводить к весьма интересным результатам, если рассматривать эти значения с их областями видимости.

В примере 3.3 определяется функция `generatePowerOfFunc`, возвращающая функцию, которая в свою очередь вычисляет степень заданного числа. Далее определяются две функции, `powerOfTwo` и `powerOfThree`, которые возводят двойку и тройку в заданную степень.



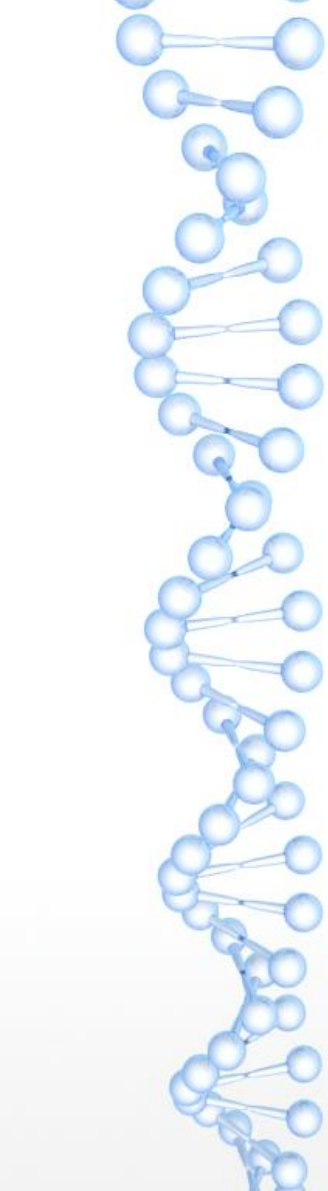
Функции, возвращающие функции

```
> // Функции, возвращающие функции
let generatePowerOfFunc baseValue = (fun exponent -> baseValue ** exponent);;

val generatePowerOfFunc : float -> float -> float

> let powerOfTwo = generatePowerOfFunc 2.0;;

val powerOfTwo : (float -> float)

> powerOfTwo 8.0;;
val it : float = 256.0
> let powerOfThree = generatePowerOfFunc 3.0;;

val powerOfThree : (float -> float)

> powerOfThree 2.0;;
val it : float = 9.0
```



Рекурсивные функции

Используя рекурсию в комбинации с функциями высшего порядка, вы легко сможете смоделировать конструкции циклов, которые можно найти в императивных языках программирования, не используя изменяемые значения. В следующем примере демонстрируются функциональные версии циклов `for` и `while`. Обратите внимание, что в примере с циклом `for` измененное значение счетчика просто передается рекурсивному вызову в виде параметра:



Рекурсивные функции

```
> // Функциональная версия цикла for
let rec forLoop body times =
    if times <= 0 then
        ()
    else
        body()
        forLoop body (times - 1)

// Функциональная версия цикла while
let rec whileLoop predicate body =
    if predicate() then
        body()
        whileLoop predicate body
    else
        ();;

val forLoop : (unit -> unit) -> int -> unit
val whileLoop : (unit -> bool) -> (unit -> unit) -> unit

> forLoop (fun () -> printfn "Looping...") 3;;
Looping...
Looping...
Looping...
val it : unit = ()
```

```
> // Типичная рабочая неделя...
open System

whileLoop
    (fun () -> DateTime.Now.DayOfWeek <> DayOfWeek.Saturday)
    (fun () -> printfn "I wish it were the weekend...");;
I wish it were the weekend...
I wish it were the weekend...
I wish it were the weekend...
    * * * Так будет продолжаться несколько дней * * *
val it : unit = ()
```

Взаимная рекурсия

Две функции, вызывающие друг друга, называются *взаимно рекурсивными*. Взаимно рекурсивные функции представляют определенную сложность для механизма вывода типов в языке F#. Чтобы определить тип первой функции, необходимо знать тип второй функции, и наоборот.

В примере 3.4 происходит ошибка компиляции взаимно рекурсивных функций, потому что к моменту компиляции функции `isOdd` функция `isEven` еще не определена.

Пример 3.4. Взаимно рекурсивные функции

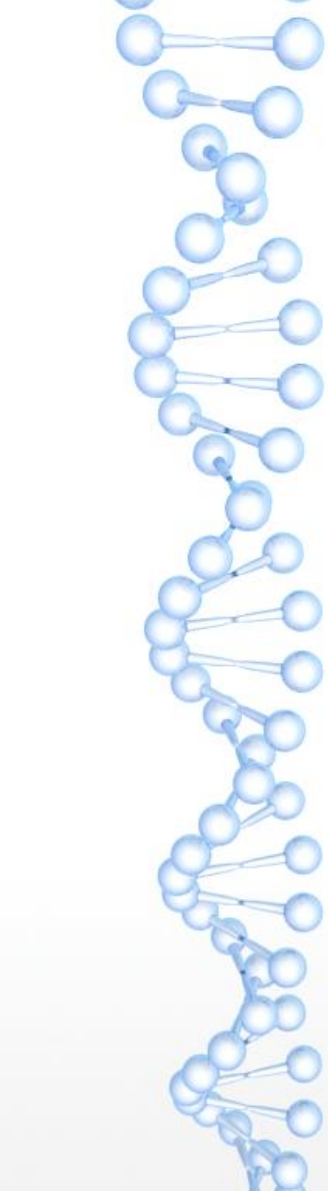
```
> // Ошибка: Невозможно определить isOdd без определения isEven, и наоборот
let isOdd  n = if n = 0 then false elif n = 1 then true else (isEven (n - 1))
let isEven n = if n = 0 then true elif n = 1 then false else (isOdd (n - 1));;

 let isOdd  n = if n = 0 then false elif n = 1 then true else (isEven (n - 1))
 ------------------------------------------------------------^^^^^^^

stdin(4,44): error FS0039: The value or constructor 'isEven' is not defined.
stopped due to error
```

(Значение или конструктор 'isEven' не определено. Остановлено из-за ошибки.)

Взаимная рекурсия

Чтобы определить взаимно рекурсивные функции, необходимо объединить их с помощью ключевого слова and, которое говорит компилятору F# выводить типы обеих функций одновременно:

```
> // Определение взаимно рекурсивных функций
let rec isOdd n = if n = 0 then false elif n = 1 then true else isEven (n - 1)
and   isEven n = if n = 0 then true elif n = 1 then false else isOdd (n - 1);;

val isOdd : int -> bool
val isEven : int -> bool

> isOdd 13;;
val it : bool = true
```



Рекурсия вверх и вниз

```
let rec cifrSum n =
    if n = 0 then 0
    else (n%10) + (cifrSum (n / 10))
```

```
let sumCifr n =
    let rec sumCifr1 n curSum =
        if n = 0 then curSum
        else
            let n1 = n / 10
            let cifr = n % 10
            let newSum  val n : int  um + n1
            sumCifr1 n1 newSum
    sumCifr1 n 0
```

Хвостовая рекурсия

Название «хвостовая рекурсия» связано с тем, что, как правило, рекурсивный вызов происходит в конце функции, то есть последняя строка («хвост») функции рекурсивно вызывает сама себя.

Преимущество хвостовой рекурсии состоит в том, что при компиляции аккумулятор можно заменить на переменную цикла, что позволяет генерировать оптимизированный код. (Ведь генерируемый машинный код является императивным. На уровне машинного кода цикл будет выполняться намного быстрее, чем рекурсивный вызов.)

Но проблема состоит в том, что далеко не каждый рекурсивный вызов можно преобразовать в хвостовую рекурсию, формального алгоритма для этого не существует.

Хвостовая рекурсия

```
///Рекурсивная (не хвостовая) функция для вычисления факториала
//аккумулятор отсутствует
let rec factorial1 n = if n<=1 then 1 else n*factorial(n-1)
```

Пример вызова функции для n=3.

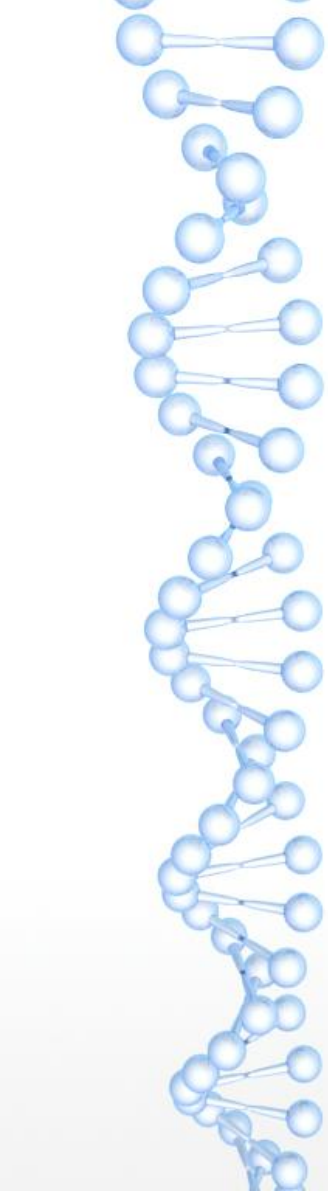
n	Возвращаемое значение	Накопленное вычисление
3	3*factorial(2)	3*factorial(2)
2	2*factorial(1)	3*2*factorial(1)
1	1	3*2*1

Хвостовая рекурсия

Для хранения накопленных вычислений, как правило, используется системный стек. После завершения цепочки рекурсивных вызовов значения 3*2*1 последовательно выбираются из стека и вычисляются.

Косвенным признаком отсутствия хвостовой рекурсии является то, что рекурсивно вычисляемая функция входит в выражение как операнд – `n*factorial(n-1)`.

В хвостовой рекурсии вычисления производятся, как правило, в операндах рекурсивно вызываемой функции:



Хвостовая рекурсия

```
    ///Функция для вычисления факториала на основе
хвостовой рекурсии
    let rec fact_tr(n:int, acc:int):int =
        if n=1 then acc
        else fact_tr(n-1, n*acc)

///Обертка для сокрытия хвостовой рекурсии
let rec factorial2 n = fact_tr(n,1)
//пример вызова
let q5 = factorial2(3)
```

Хвостовая рекурсия

Пример вызова функции `fact_tr` для n=3.

Входные параметры		Возвращаемое значение
n	**acc**	
3	1	fact_tr(3-1, 3*1) fact_tr(2, 3)
2	3	fact_tr(2-1, 3*2) fact_tr(1, 6)
1	6	6

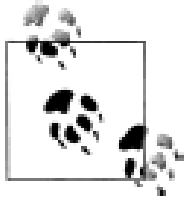
Хвостовая рекурсия

Отметим, что характерным признаком хвостовой рекурсии является отсутствие накопленных вычислений, так как вычисления производятся в операндах вызываемой функции. Поэтому нет необходимости накапливать данные в стеке, и хвостовая рекурсия может быть преобразована в цикл, где роль переменной цикла выполняет аккумулятор.

Большинство современных компиляторов с функциональных языков программирования гарантируют генерацию оптимизированного кода в случае хвостовой рекурсии.

Символьные операторы

Представьте, насколько сложно было бы программировать, если бы всякий раз вместо выражения `1 + 2` приходилось бы писать `add 1 2`. К счастью, F# не только поддерживает встроенные символьные операторы для выполнения таких операций, как сложение и вычитание, но и дает нам возможность определять собственные операторы, позволяя писать более понятный и выразительный код.



Символьные функции не следует рассматривать как разновидность перегрузки операторов, скорее это функции, имена которых составлены из специальных символов.

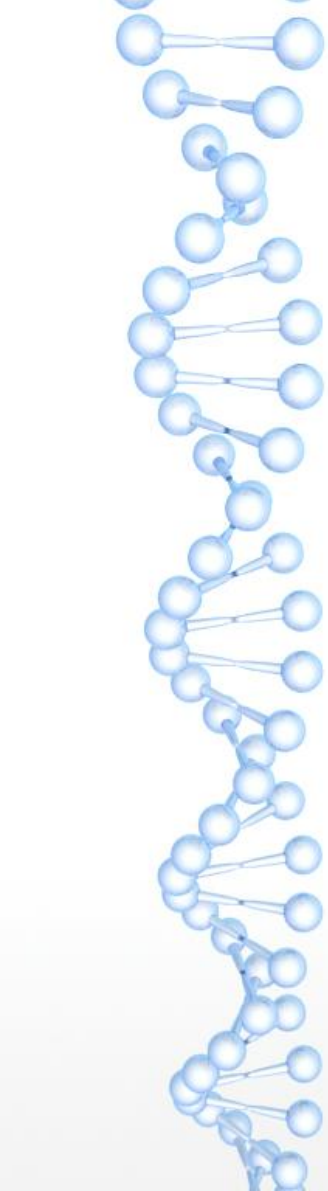
Символьный оператор может представлять собой любую комбинацию из следующих символов: `!%&*+-./<=>?@^|~` (включая `:`, при условии, что он не будет первым в комбинации). Следующий фрагмент определяет новую функцию `!`, которая вычисляет факториал числа:

Символьные операторы

```
> // Факториал
let rec (!) x =
    if x <= 1 then 1
    else x * !(x - 1);;

val ( ! ) : int -> int

> !5;;
val it : int = 120
```

```
> // Определение (===) для сравнения строк на основе регулярного выражения
open System.Text.RegularExpressions;;

let (===) str (regex : string) =
    Regex.Match(str, regex).Success;;

val ( === ) : string -> string -> bool

> "The quick brown fox" === "The (.*) fox";;
val it : bool = true
```

По умолчанию при использовании символьных функций, принимающих более одного параметра, применяется *инфиксная нотация*. То есть первый параметр предшествует символу, что является привычной формой использования символьных функций. В следующем примере определяется функция ===, которая сравнивает строку на основе регулярного выражения:

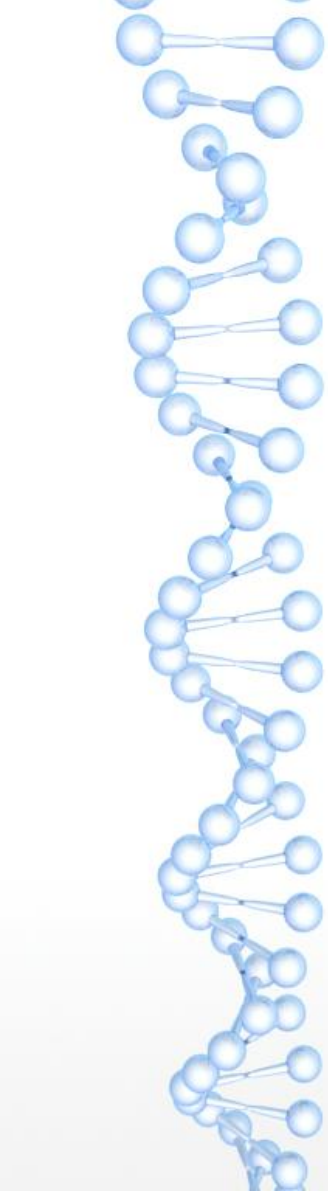


Символьные операторы

```
> let (~+++) x y z = x + y + z;;
val ( ~+++ ) : int -> int -> int -> int
> ~+++ 1 2 3;;
val it : int = 6
```

Помимо возможности создавать имена функций, которые точно соответствуют математической нотации, вы можете передавать символьные операторы функциям высшего порядка – для этого достаточно заключить оператор в круглые скобки. Например, если вам потребуется найти сумму или произведение элементов списка, вы можете просто записать:

```
> // Вычисление суммы элементов списка
// с использованием символьной функции (+)
List.fold (+) 0 [1 .. 10];;
val it : int = 55

> // Вычисление произведения элементов списка
// с использованием символьной функции (*)
List.fold (*) 1 [1 .. 10];;
val it : int = 3628800

> let minus = (-);;

val minus : (int -> int -> int)

> List.fold minus 10 [3; 3; 3];;
val it : int = 1
```

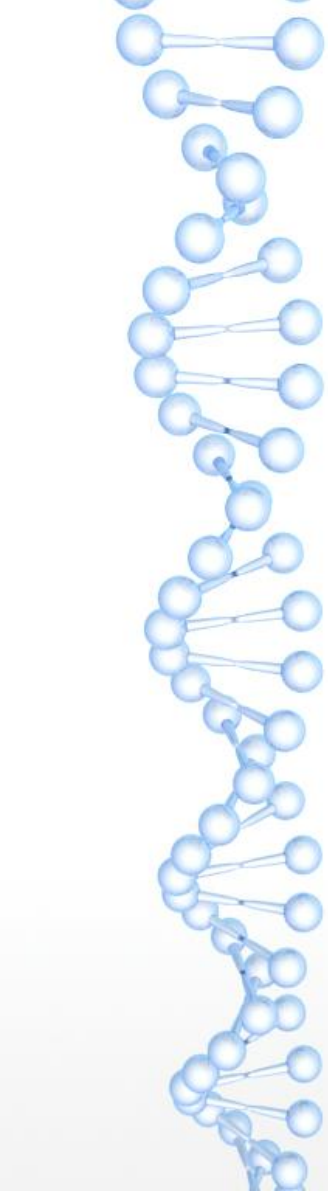


Композиция функций

Как только вы освоите работу с функциями, вы можете начинать объединять их в более крупные и мощные функции. Этот прием известен как *композиция функций* и является еще одним основополагающим принципом функционального программирования.

Но прежде чем перейти к объединению функций, давайте посмотрим, какую проблему это решает. Ниже приводится пример, как не надо делать: вся реализация помещена внутрь одной огромной функции. Этот код определяет размер указанной папки на диске (я добавил аннотации типов, чтобы помочь определить тип возвращаемого значения):

```
open System
open System.IO

let sizeOfFolder folder =
```



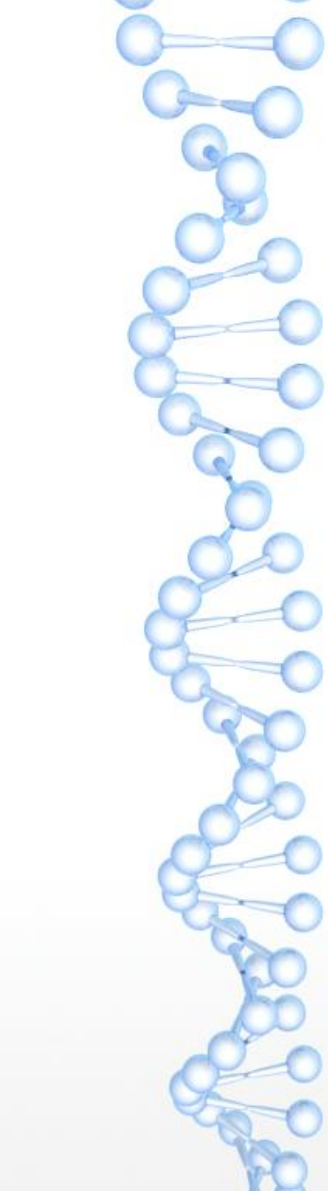
Композиция функций

```
// Получить список всех файлов в папке
let filesInFolder : string [] =
    Directory.GetFiles(
        folder, "*.*",
        SearchOption.AllDirectories)

// Отобразить имена файлов на соответствующие им объекты FileInfo
let fileInfos : FileInfo [] =
    Array.map
        (fun file -> new FileInfo(file))
        filesInFolder

// Отобразить объекты fileInfo на значения размеров файлов
let fileSizes : int64 [] =
    Array.map
        (fun (info : FileInfo) -> info.Length)
        fileInfos

// Суммарный размер файлов
let totalSize = Array.sum fileSizes

// Вернуть суммарный размер файлов
totalSize
```

Композиция функций

У этого кода есть три основные проблемы:

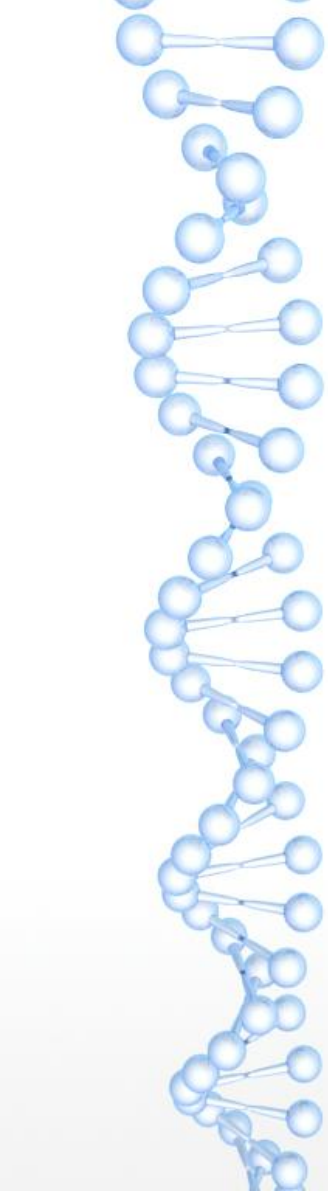
- Механизм вывода типов не способен правильно определить типы, поэтому пришлось добавить аннотации типов параметров в каждое лямбда-выражение. Это обусловлено тем, что механизм вывода типов просматривает код слева направо, сверху вниз и встречает лямбда-выражение, передаваемое функции `Array.map`, до того, как он сможет определить тип элементов массива. (То есть тип параметров лямбда-выражения неизвестен.)
- Результат каждого этапа вычислений передается в виде параметра следующему этапу, поэтому инструкции `let` в функции выглядят ненужным излишеством.
- Код попросту ужасен. Чтобы понять, что он делает, придется потратить времени больше, чем следует.



Композиция функций

Композиция функций позволяет разбить подобный код на несколько маленьких функций, а затем объединить их в окончательную реализацию.

В предыдущем примере результаты одного вычисления передаются следующему. На языке математики передача результата функции `f(x)` в функцию `g(x)` записывается так: `g(f(x))`. Мы могли бы избежать всех этих операторов связывания `let`, используя вложенное вычисление промежуточных результатов, но такой код крайне сложно читать, как можно убедиться по следующему фрагменту:



Композиция функций

```
let uglySizeOfFolder folder =
    Array.sum
        (Array.map
            (fun (info : FileInfo) -> info.Length)
            (Array.map
                (fun file -> new FileInfo(file))
                (Directory.GetFiles(
                    folder, "*.*",
                    SearchOption.AllDirectories))))
```

Прямой конвейерный оператор

К счастью, язык F# предоставляет лаконичное решение проблемы передачи промежуточных результатов от функции к функции с помощью прямого конвейерного оператора (pipe-forward opeator) |>. Он определяется следующим образом:

```
let (|>) x f = f x
```

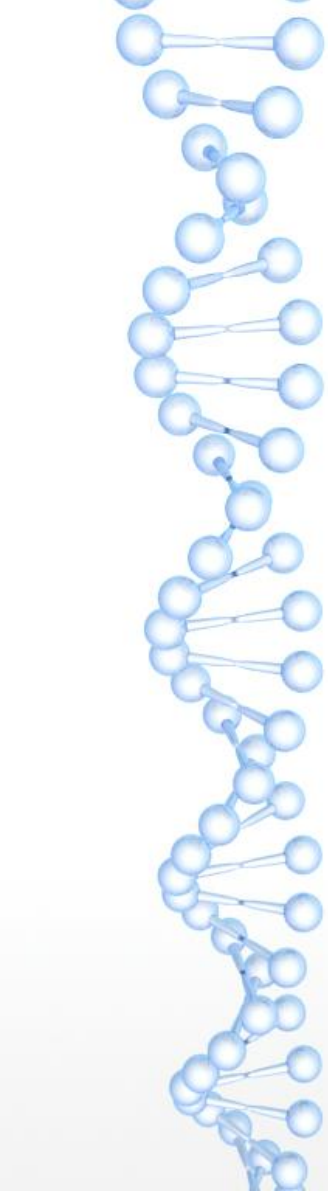
Прямой конвейерный оператор позволяет переупорядочить параметры функции так, что при вызове функции последний ее параметр указывается первым. Если учесть, что последним параметром функции List.iter передается список, по которому выполняются итерация, то, используя прямой конвейерный оператор, можно отправить этот список функции List.iter «по конвейеру», поместив его первым:

```
> [1 .. 3] |> List.iter (printfn "%d");;
1
2
3
val it : unit = ()
```

Прямой конвейерный оператор

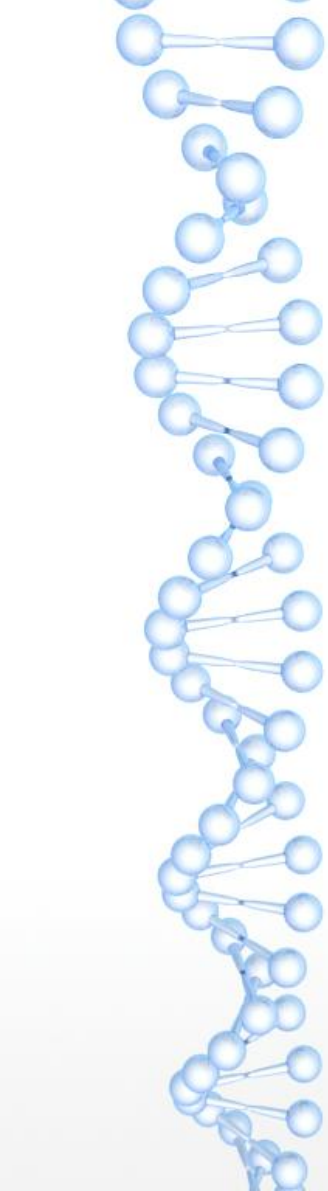
С технической точки зрения прямой конвейерный оператор и его брат, обратный конвейерный оператор, в действительности не являются механизмом композиции функций. Скорее они связаны с применением функций.

Преимущество прямого конвейерного оператора состоит в том, что с его помощью можно составлять целые цепочки вызовов функций. То есть передавать результат предыдущей функции на вход следующей функции. Теперь мы можем переписать функцию `sizeOfFolder`, как показано ниже. Обратите внимание, что функция `Directory.GetFiles` принимает свои параметры в виде кортежа, поэтому она не может быть каррирована:



Прямой конвейерный оператор

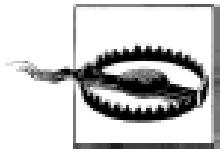
```
let sizeOfFolderPiped folder =

    let getFiles folder =
        Directory.GetFiles(folder, "*.*", SearchOption.AllDirectories)

    let totalSize =
        folder

          |> getFiles
          |> Array.map (fun file -> new FileInfo(file))
          |> Array.map (fun info -> info.Length)
          |> Array.sum

      totalSize
```



Прямой конвейерный оператор

Простота, достигнутая с помощью прямого конвейерного оператора, является наглядным примером того, насколько полезным может оказаться каррирование функций. Прямой конвейерный оператор принимает значение и функцию с единственным параметром. Однако использованная нами функция `Array.map` принимает два параметра (функцию, выполняющую отображение, и сам массив). Причина, по которой наш программный код оказался работоспособным, заключается в том, что мы каррировали один аргумент, получили в результате функцию с единственным входным параметром и благодаря этому смогли использовать ее с прямым конвейерным оператором.

Прямой конвейерный оператор



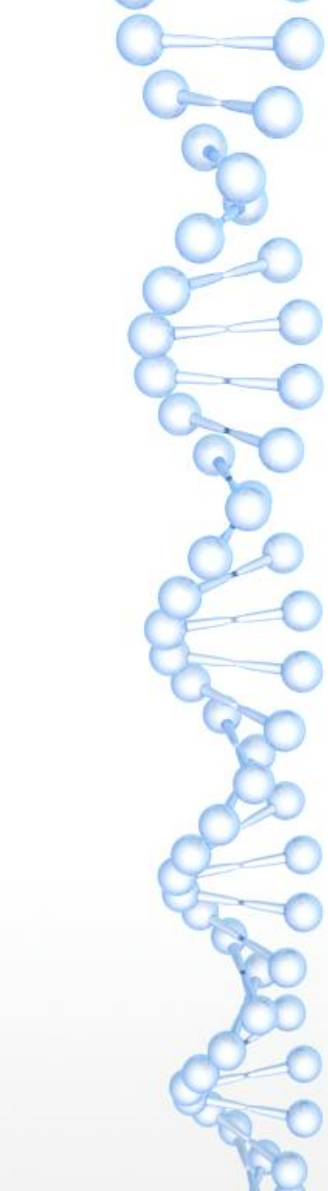
Прямой конвейерный оператор способен существенно упростить код за счет устранения ненужных объявлений промежуточных результатов, однако отладка последовательностей функций, объединенных конвейером, становится намного сложнее, потому что вы лишаетесь возможности проверить значения промежуточных результатов.

Дополнительным преимуществом прямого конвейерного оператора заключается в том, что он оказывает помощь механизму выведения типов. Вы не сможете получить доступ ни к одному свойству или методу значения, если компилятор не может определить его тип. Поэтому для явного указания типов в подобных ситуациях приходится использовать аннотации:

Прямой конвейерный оператор

```
> // ОШИБКА: Компилятору неизвестно, что значение s имеет свойство Length
List.iter
    (fun s -> printfn "s has length %d" s.Length)
    ["Pipe"; "Forward"];;

      (fun s -> printfn "s has length %d" s.Length)
  ----------------------------------------^^^^^^^^^

stdin(89,41): error FS0072: Lookup on object of indeterminate type based on
information prior to this program point. A type annotation may be needed prior
to this program point to constrain the type of the object. This may allow the
lookup to be resolved
stopped due to error
```

(Поиск объекта неопределенного типа, основанного на информации до данной точки программы. Возможно, перед данной точкой программы потребуется аннотация типа с целью ограничения типа объекта. Возможно, это позволит разрешить поиск. Остановлено из-за ошибки.)

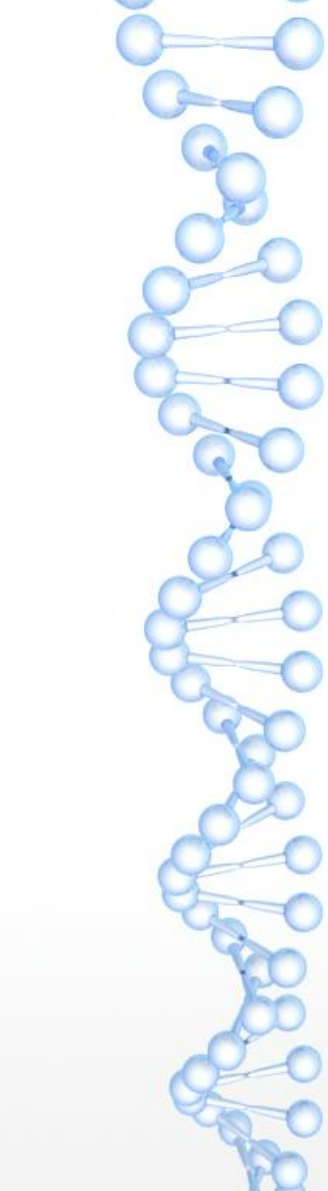


Прямой конвейерный оператор

Прямой конвейерный оператор помогает компилятору «увидеть» последний параметр функции ранее, благодаря чему механизм выведения типов может определить правильные типы функций без дополнительных аннотаций.

В следующем примере благодаря использованию прямого конвейерного оператора параметр лямбда-выражения, передаваемого функции `List.iter`, определяется как значение типа `string`, вследствие чего отпадает необходимость в аннотации типа:

```
> ["Pipe"; "Forward"] |> List.iter (fun s -> printfn "s has length %d"
s.Length);;
s has length 4
s has length 7
val it : unit = ()
```



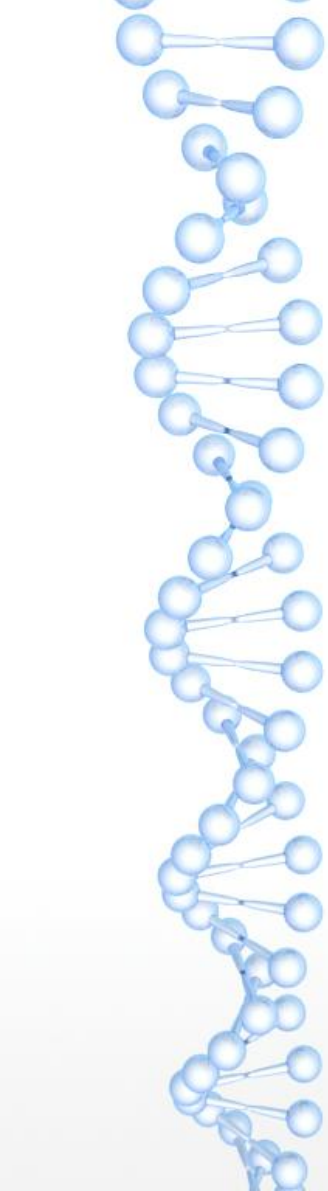
Прямой оператор композиции

Прямой оператор композиции (forward composition operator) >> объединяет две функции, при этом функция слева вызывается первой:

```
let (>>) f g x = g(f x)
```

При использовании прямого конвейерного оператора необходимо указать переменную, чтобы «запустить» конвейерную обработку. В нашем последнем примере функция принимала параметр folder, который передавался первой функции в конвейере:

```
let sizeOfFolderPiped2 folder =

    let getFiles folder =
        Directory.GetFiles(folder, "*.*", SearchOption.AllDirectories)

    folder
    |> getFiles
    |> Array.map (fun file -> new FileInfo(file))
    |> Array.map (fun info -> info.Length)
    |> Array.sum
```

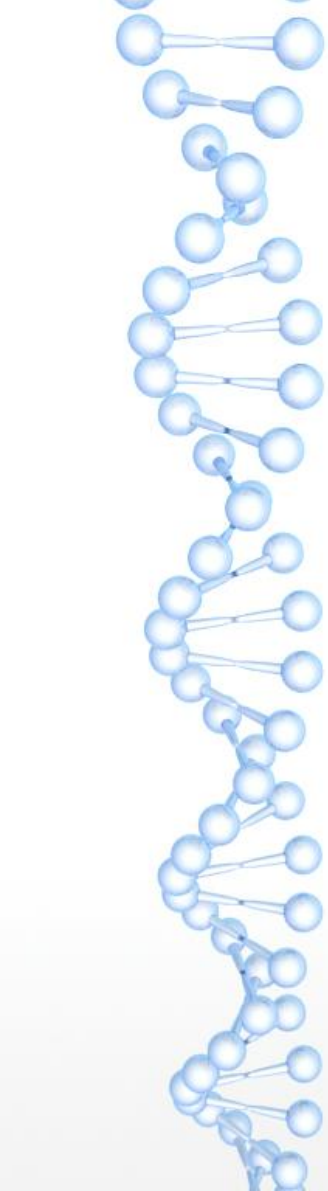


Прямой оператор композиции

```
> // Композиция функций

open System.IO

let sizeOfFolderComposed (*Нет параметров!*) =

    let getFiles folder =
        Directory.GetFiles(folder, "*.*", SearchOption.AllDirectories)

        // Результатом этого выражения является функция, принимающая
        // один параметр, который будет передан функции getFiles и далее
        // по конвейеру последующим функциям.
        getFiles
        >> Array.map (fun file -> new FileInfo(file))
        >> Array.map (fun info -> info.Length)
        >> Array.sum;;

    val sizeOfFolderComposed : (string -> int64)

    > sizeOfFolderComposed
        (Environment.GetFolderPath(Environment.SpecialFolder.MyPictures));;
    val it : int64 = 904680821L
```

Прямой оператор композиции

```
> // Композиция простых функций
let square x = x * x
let toString (x : int) = x.ToString()
let strLen (x : string) = x.Length
let lenOfSquare = square >> toString >> strLen;;

val square : int -> int
val toString : int -> string
val strLen : string -> int
val lenOfSquare : (int -> int)

> square 128;;
val it : int = 16384
> lenOfSquare 128;;
val it : int = 5
```



Сопоставление с образцом

В любых программах возникает необходимость фильтровать и сортировать данные. Для этой цели в функциональном программировании используется сопоставление с образцом (pattern matching). Сопоставление с образцом напоминает инструкцию `switch` в языках C# и C++, но обладает более широкими возможностями. По сути это набор правил, которые выполняются при совпадении входного значения с образцом. Затем выражение сопоставления с образцом возвращает результат правила, для которого было найдено соответствие. Вследствие этого все правила должны возвращать значения одного и того же типа.

Для использования сопоставления с образцом применяются ключевые слова `match` и `with` с набором правил, за каждым из которых следует стрелка `->`. Следующий фрагмент демонстрирует использование сопоставления с результатом выражения `isOdd x` для имитации поведения условного выражения. Первое правило соответствует значению `true`, и, если будет найдено совпадение с этим правилом, в консоль будет выведено сообщение `"x is odd"` (`x` – нечетное число):

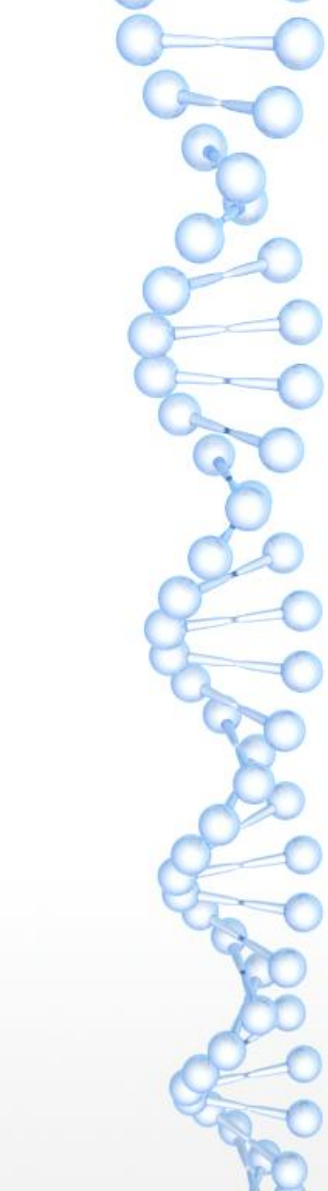
Сопоставление с образцом

```
> // Простое сопоставление с образцом
let isOdd x = (x % 2 = 1)

let describeNumber x =
    match isOdd x with
    | true  -> printfn "x is odd"
    | false -> printfn "x is even";;

val isOdd : int -> bool
val describeNumber : int -> unit

  > describeNumber 4;;
  x is even
  val it : unit = ()
```

Сопоставление с образцом

Пример 3.5. Создание таблицы истинности с помощью сопоставления с образцом

```
> // Таблица истинности для функции AND с помощью сопоставления с образцом
let testAnd x y =
    match x, y with
    | true,  true  -> true
    | true,  false -> false
    | false, true  -> false
    | false, false -> false;;

val testAnd : bool -> bool -> bool

> testAnd true true;;
val it : bool = true
```



Сопоставление с образцом

Обратите внимание, что механизм вывода типов работает с выражениями сопоставления с образцом. В примере 3.5 первое правило сопоставляет `x` и `y` с булевыми величинами, механизм выведения типов определяет, что `x` и `y` являются булевыми величинами.

Символ подчеркивания `_` в выражениях сопоставления играет роль группового символа (wildcard) и совпадает с любыми значениями. Благодаря этому мы можем упростить предыдущий пример, добавив правило с групповым символом, которое будет выполняться для любых комбинаций входных значений кроме `true true`:

```
let testAnd x y =
    match x, y with
    | true, true -> true
    | _, _       -> false
```

Правила сопоставления с образцом проверяются в порядке их объявления. Поэтому если вставить правило с групповыми символами первым, все последующие правила никогда не будут проверяться.

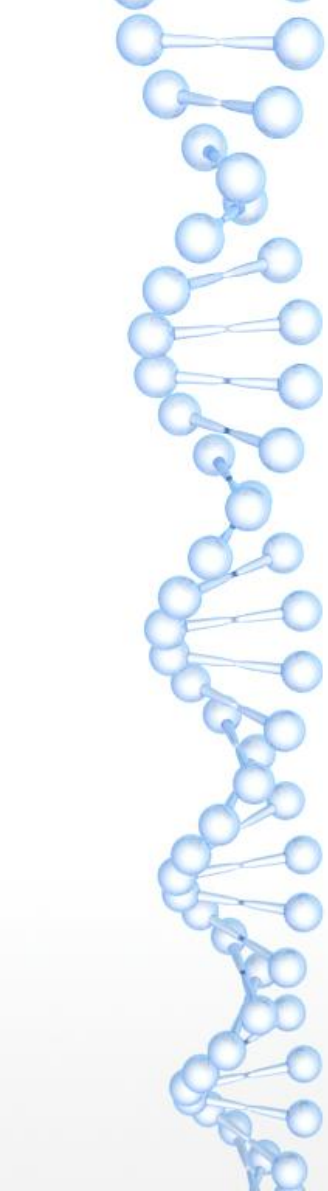
Сопоставление с образцом

```
        | false, false -> false
```

Если в процессе сопоставления с образцом не будет найдено совпадение, будет сгенерировано исключение `Microsoft.FSharp.Core.MatchFailureException`. Вы можете избежать этого, определив правила для всех возможных значений. К счастью, компилятор F# выводит предупреждение, когда оказывается в состоянии обнаружить, что набор правил неполон:

```
> // Неполный набор правил.
// ВНИМАНИЕ! Нет совпадений для каждого символа.
let letterIndex l =
    match l with
    | 'a' -> 1
    | 'b' -> 2;;

      match l with
  ----------^
stdin(48,11): warning FS0025: Incomplete pattern matches on this expression.
For example, the value '' '' may indicate a case not covered by the pattern(s).
```

(Незавершенный шаблон соответствует данному выражению. К примеру, значение "' '" может указывать на случай, не покрытый шаблоном(ами).)



Именованные образцы

До сих пор мы выполняли сопоставление с константами, но точно так же можно использовать именованные образцы при извлечении данных и привязке их к новым значениям. Взгляните на следующий пример. В нем выполняется сопоставление с определенными строками, а в случае несоответствия используется значение, связанное с именем `x`:

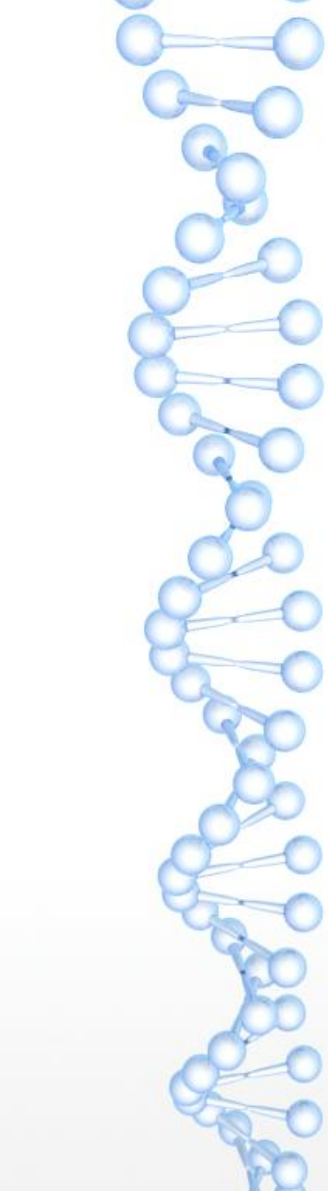
```
> // Именованные образцы
let greet name =
    match name with
    | "Robert"  -> printfn "Hello, Bob"
    | "William" -> printfn "Hello, Bill"
    | x         -> printfn "Hello, %s" x;;
```

Сопоставление с литералами

Атрибуты – это способ аннотирования программного кода в .NET. Дополнительная информация об атрибутах и метаданных .NET приводится в главе 12.

```
> // Определение литерального значения
[<Literal>]
let Bill = "Bill Gates";;

val Bill : string = "Bill Gates"
> // Сопоставление с литеральными значениями

let greet name =
    match name with
    | Bill -> "Hello Bill!"
    | x    -> sprintf "Hello, %s" x;;

val greet : string -> string
> greet "Bill G.";;
val it : string = "Hello, Bill G."
> greet "Bill Gates";;
val it : string = "Hello Bill!"
```

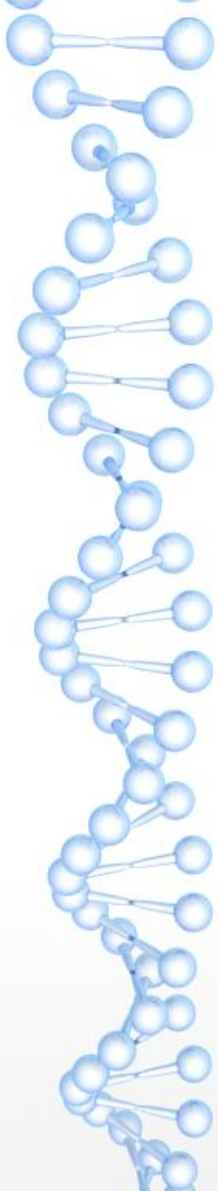
Атрибутом `[<Literal>]` могут быть помечены только целые числа, символы, булевы значения, строки и вещественные числа. Если вам потребуется использовать сопоставление со значениями более сложных типов, такими как словари или отображения, придется использовать ограничение `when`.



Ограничение when

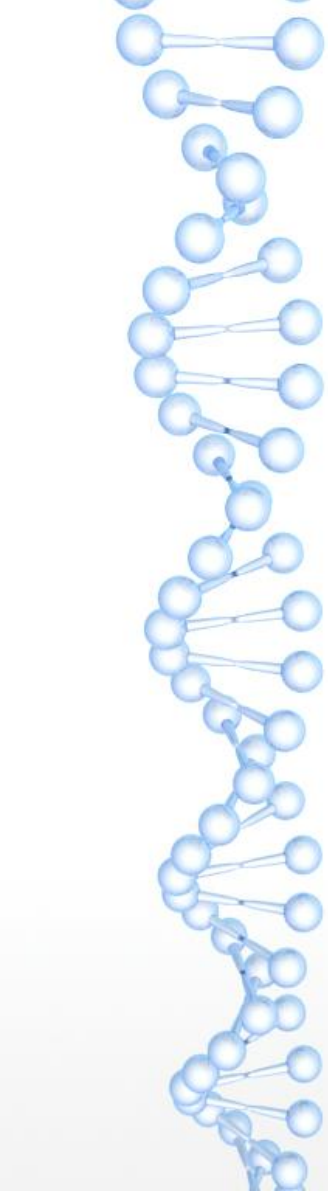
Сопоставление с образцом – достаточно мощная концепция, но иногда бывает необходимо добавить дополнительную логику, чтобы определить, соответствует значение данному правилу или нет. Именно для этого предназначено ограничение `when`. Если сопоставление с образцом найдено, вычисляется необязательное ограничение `when`, и правило выполняется тогда и только тогда, когда выражение `when` возвращает `true`.

В следующем примере реализована простая игра, в которой вам нужно угадать случайное число. Ограничение `when` используется, чтобы определить, является ли предложенное вами число больше, меньше или равно секретному числу:



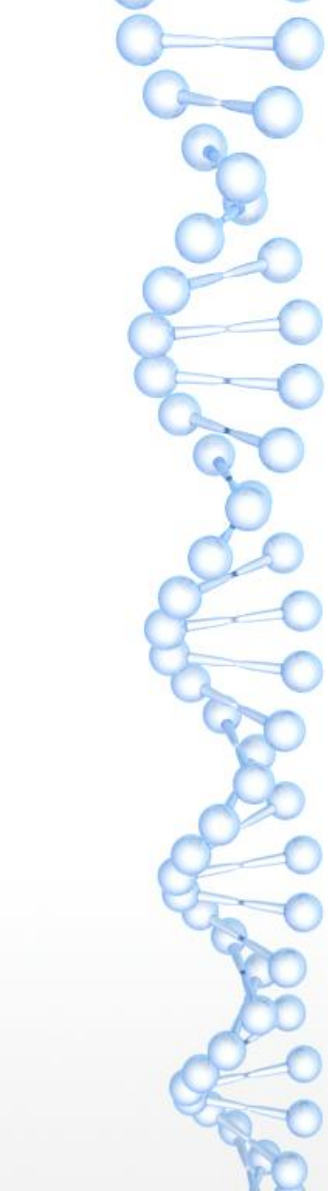
Ограничение when

```
let highLowGame () =

    let rng = new Random()
    let secretNumber = rng.Next() % 100

    let rec highLowGameStep () =

        printfn "Guess the secret number:"
        let guessStr = Console.ReadLine()
        let guess = Int32.Parse(guessStr)
        match guess with
        | _ when guess > secretNumber
            -> printfn "The secret number is lower."
               highLowGameStep()

        | _ when guess = secretNumber
            -> printfn "You've guessed correctly!"
               ()

        | _ when guess < secretNumber
            -> printfn "The secret number is higher."
               highLowGameStep()

    // Начало игры
    highLowGameStep();;

val highLowGame : unit -> unit

> highLowGame();;
Guess the secret number: 50
The secret number is lower.
Guess the secret number: 25
The secret number is higher.
Guess the secret number: 37
You've guessed correctly!
val it : unit = ()
```



Группировка образцов

```
let vowelTest c =
    match c with

    | 'a' | 'e' | 'i' | 'o' | 'u'
        -> true
    | _ -> false

let describeNumbers x y =
    match x, y with
    | 1, _
    | _, 1
        -> "One of the numbers is one."
    | (2, _) & (_, 2)
        -> "Both of the numbers are two"
    | _ -> "Other."
```



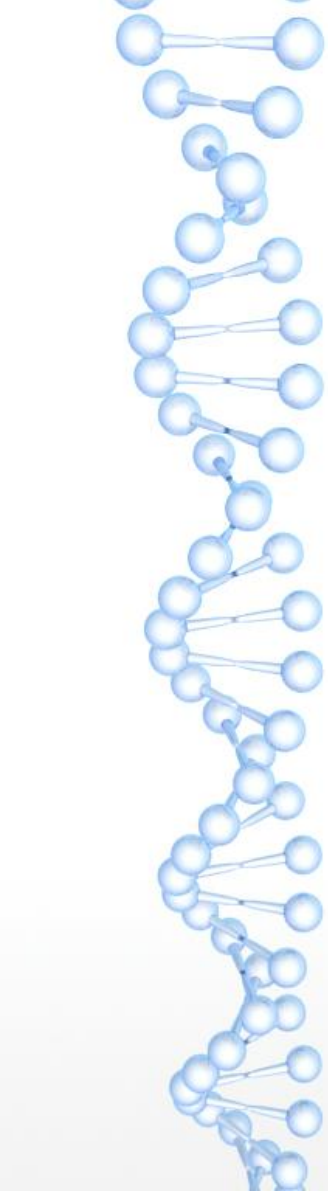
Сопоставление структур данных

Кортежи

Вы уже видели, как можно использовать сопоставление с кортежами. Если в сопоставлении элементы кортежа отделены друг от друга запятыми, то все элементы кортежа будут сопоставляться по отдельности. Однако если с именованным образцом используется кортеж, связываемое значение должно иметь тип кортежа.

В следующем примере первое правило связывает значение с именем `tuple` с кортежем и захватывает оба значения `x` и `y`. Другие правила выполняют сопоставление элементов кортежа по отдельности:

```
let testXor x y =
    match x, y with
    | tuple when fst tuple <> snd tuple
        -> true
    | true, true -> false
    | false, false -> false
```



Сопоставление структур данных

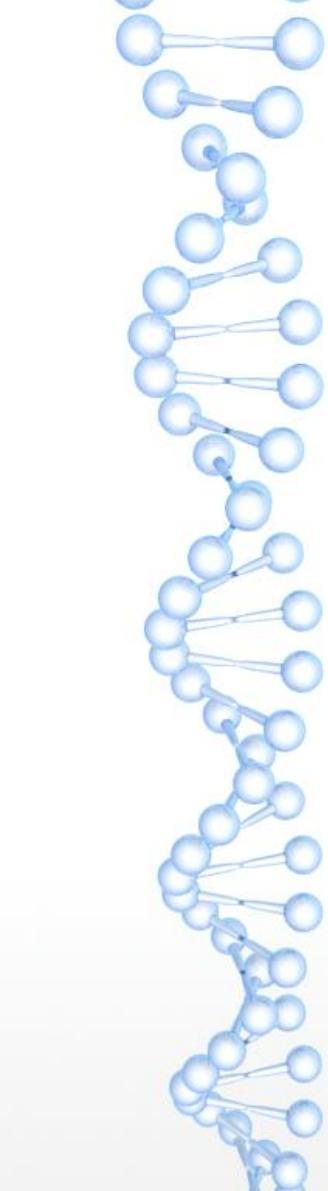
Списки

В примере 3.6 демонстрируется использование сопоставления с образцом совместно со списками. Функция `listLength` сначала находит сопоставление со списками фиксированной длины, и если совпадение не найдено, она рекурсивно вызывает саму себя со списком без первого элемента.

Пример 3.6. Определение длины списка

```
let rec listLength l =
   match l with
   | []         -> 0
   | [ ]        -> 1

   | [_; _]     -> 2
   | [_; _; _]  -> 3
   | hd :: tail -> 1 + listLength tail
```

Первые четыре правила находят сопоставления со списками определенной длины, используя групповые символы, чтобы показать, что значения элементов списка не важны. Однако в последней строке сопоставления с образцом используется оператор добавления элемента `::`, чтобы найти соответствие первого элемента списка, `hd`, оставшейся его части, `tail`. `tail` может быть любым списком: как пустым `[]`, так и содержащим миллион элементов. (Впрочем, учитывая предыдущие правила в функции, можно смело заявить, что список `tail` будет содержать не менее трех элементов.)



Сопоставление структур данных

Необязательные значения

Сопоставление с образцом предоставляет более функциональный подход к использованию типов `option`:

```
let describeOption o =
    match o with
    | Some(42) -> "The answer was 42, but what was the question?"[1]
    | Some(x)  -> sprintf "The answer was %d" x
    | None     -> "No answer found."
```

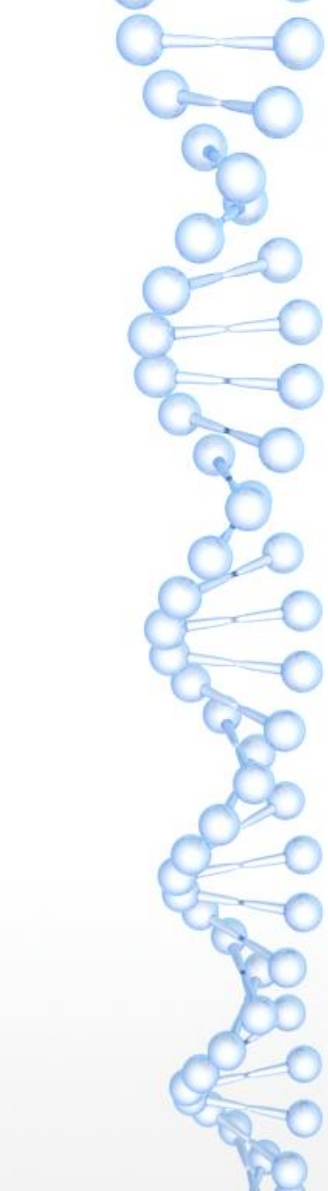
Альтернативный синтаксис лямбда-выражений

Наконец, сопоставления с образцом можно использовать как упрощенный синтаксис лямбда-выражений. В процессе разработки программ на языке F# обычной практикой является передача параметра непосредственно в выражение сопоставления с образцом. Например:

```
let rec listLength theList =
    match theList with
    | [] -> 0
    | [_] -> 1
    | [_; _] -> 2
    | [_; _; _] -> 3
    | hd :: tail -> 1 + listLength tail
```

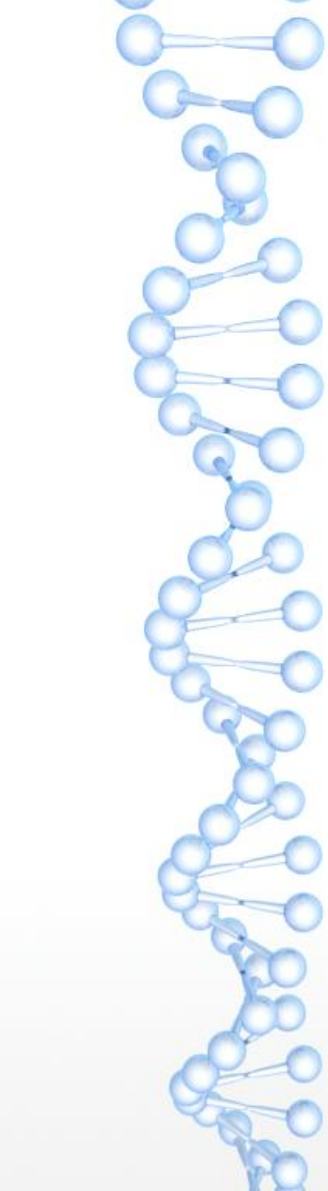
Альтернативный синтаксис лямбда-выражений

```
> // Ключевое слово 'function'
let rec funListLength =
    function
    | [] -> 0
    | [_] -> 1
    | [_; _] -> 2
    | [_; _; _] -> 3
    | hd :: tail -> 1 + funListLength tail;;

val funListLength : 'a list -> int

> funListLength [1 .. 5];;
val it : int = 5
```



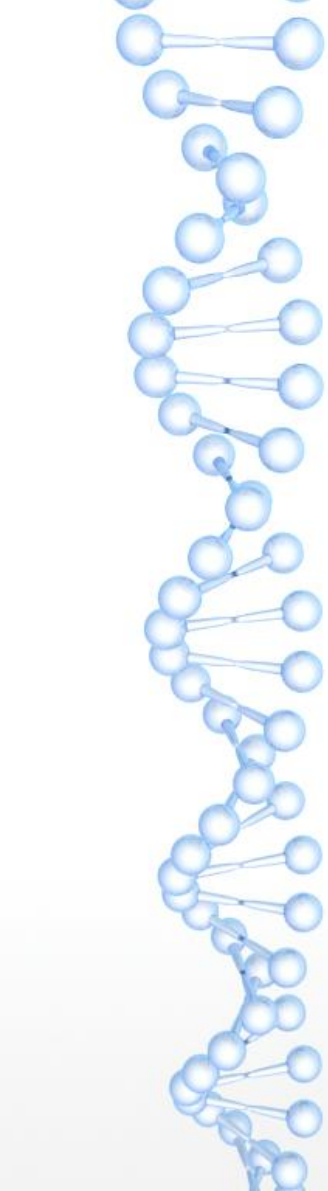
Размеченные объединения

```
> // Размеченное объединение, представляющее масть карты
type Suit =
    | Heart
    | Diamond
    | Spade
    | Club;;

type Suit =
    | Heart
    | Diamond
    | Spade
    | Club

> let suits = [ Heart; Diamond; Spade; Club ];;

val suits : Suit list = [Heart; Diamond; Spade; Club]
```



Размеченные объединения

```
// Размеченное объединение для представления игральных карт
type PlayingCard =
    | Ace   of Suit
    | King  of Suit
    | Queen of Suit
    | Jack  of Suit
    | ValueCard of int * Suit

// Использовать генератор списка для создания колоды карт
let deckOfCards =
    [
        for suit in [ Spade; Club; Heart; Diamond ] do
            yield Ace(suit)
            yield King(suit)
            yield Queen(suit)
            yield Jack(suit)
            for value in 2 .. 10 do
                yield ValueCard(value, suit)
    ]
```

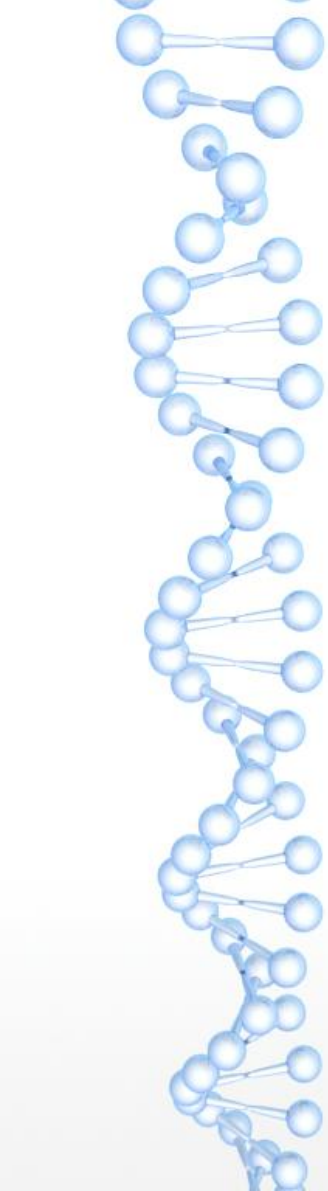


Размеченные объединения

Размеченные объединения могут образовывать рекурсивные структуры. Если вам потребуется объявить множество взаимно рекурсивных размеченных объединений, то, как и в случае с функциями, вам необходимо будет связать их с помощью ключевого слова `and`.

Ниже приводится пример простого формата для описания языка программирования:

```
// Инструкции
type Statement =
    | Print    of string
    | Sequence of Statement * Statement
    | IfStmt   of Expression * Statement * Statement

// Выражения
and Expression =
    | Integer     of int
    | LessThan    of Expression * Expression
    | GreaterThan of Expression * Expression
```



Размеченные объединения

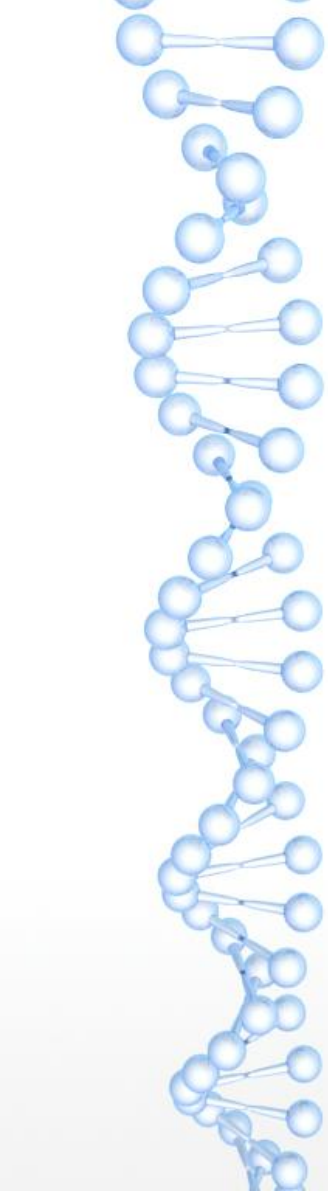
```
(*
    if (3 > 1)
        print "3 is greater than 1"
    else
        print "3 is not"
        print "greater than 1"
*)
let program =
    IfStmt(
        GreaterThan(
            Integer(3),
            Integer(1)),
        Print("3 is greater than 1"),
        Sequence(
            Print("3 is not"),
            Print("greater than 1")
        )
    )
```

Использование размеченных объединений для создания древовидных структур

```
type BinaryTree =
    | Node of int * BinaryTree * BinaryTree
    | Empty

let rec printInOrder tree =
    match tree with
    | Node (data, left, right)
        -> printInOrder left
           printfn "Node %d" data
           printInOrder right
    | Empty
        -> ()

(*
        2
       / \
      1   4
         / \
        3   5

*)
let binTree =
    Node(2,
        Node(1, Empty, Empty),
        Node(4,
            Node(3, Empty, Empty),
            Node(5, Empty, Empty)
        )
    )
```

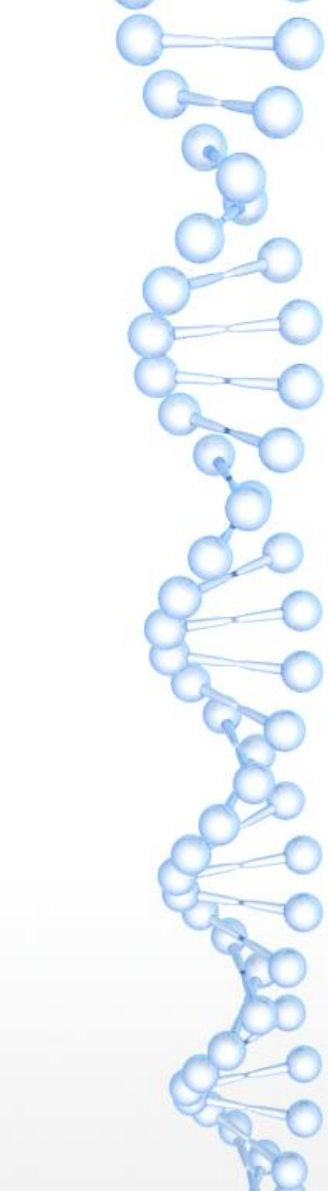
Если выполнить этот пример в окне FSI, то получим следующее:

```
> printInOrder binTree;;
Node 1
Node 2
Node 3
Node 4
Node 5
val it : unit = ()
```



Использование размеченных объединений для создания древовидных структур

```
type Employee =
    | Manager of string * Employee list
    | Worker of string

let rec printOrganization worker =
    match worker with
    | Worker(name) -> printfn "Employee %s" name

    // Руководитель, в списке подчиненных у которого всего один элемент
    | Manager(managerName, [ Worker(employeeName) ])
        -> printfn "Manager %s with Worker %s" managerName employeeName

    // Руководитель, в списке подчиненных у которого два элемента
    | Manager(managerName, [ Worker(employee1); Worker(employee2) ])
        -> printfn
                "Manager %s with two workers %s and %s"
                managerName employee1 employee2

    // Руководитель со списком подчиненных
    | Manager(managerName, workers)
        -> printfn "Manager %s with workers..." managerName
           workers |> List.iter printOrganization
```



Использование размеченных объединений для создания древовидных структур

Если предыдущий пример запустить в окне FSI, он выведет следующее:

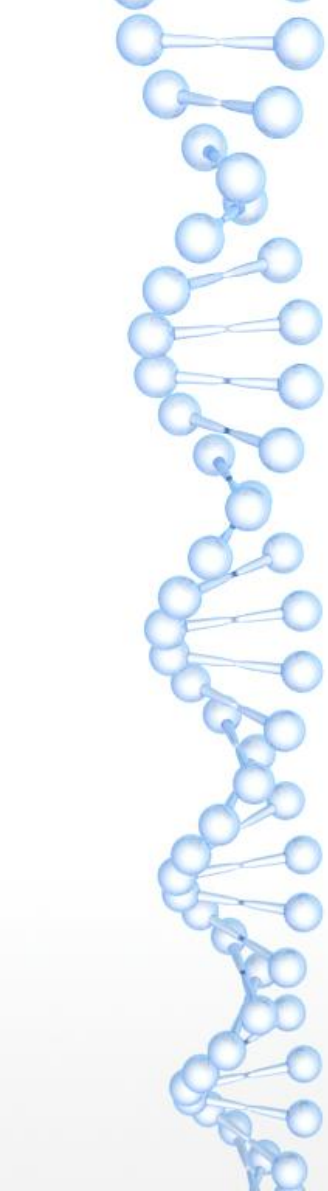
```
> let company = Manager("Tom", [ Worker("Pam"); Worker("Stuart") ] );;

val company : Employee = Manager ("Tom",[Worker "Pam"; Worker "Stuart"])

> printOrganization company;;
Manager Tom with two workers Pam and Stuart
val it : unit = ()
```

Записи

Размеченные объединения с успехом могут использоваться для определения иерархических данных, но когда вы пытаетесь получить данные из размеченного объединения, возникает та же проблема, что и с кортежами, а именно: значения не имеют какого-то особого смысла – скорее, это просто данные, сваленные в общую кучу в некотором фиксированном порядке. Например, взгляните на следующее размеченное объединение, предназначенное для описания человека. Первая строковое поле в нем описывает имя, а второе – фамилию, или наоборот? Такая неопределенность может приводить к путанице и появлению ошибок.

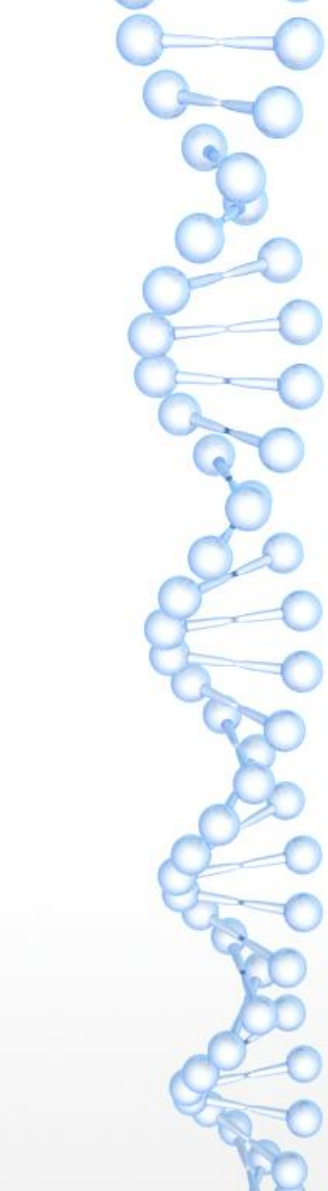
```
type Person =
    | Person of string * string * int

let steve = Person("Steve", "Holt", 17)
let gob = Person("Bluth", "George Oscar", 36)
```

Если вам необходимо сгруппировать данные в структуру с помощью достаточно простого синтаксиса, вы можете использовать тип *запись (record)*. Записи позволяют организовать значения в типы, а также присвоить имена отдельным *полям*.



Записи

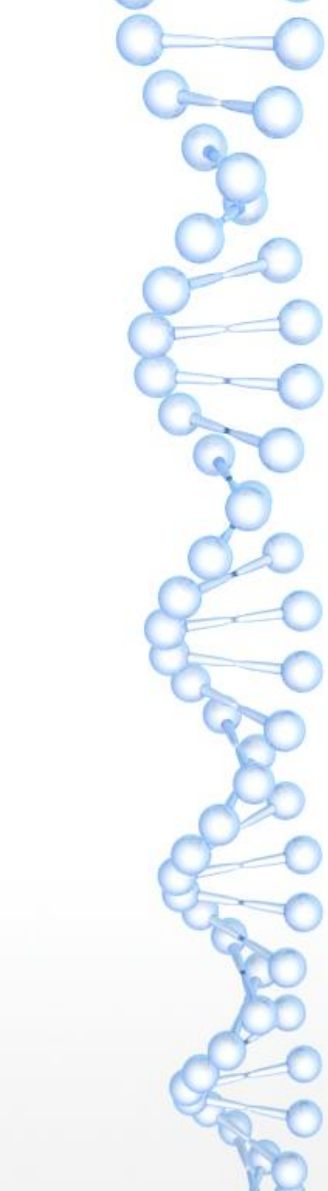
```
> // Определение типа записи
type PersonRec = { First : string; Last : string; Age : int};;

type PersonRec =
  {First: string;
   Last: string;
   Age: int;}

> // Конструирование экземпляра записи
let steve = { First = "Steve"; Last = "Holt"; Age = 17 };;

val steve : PersonRec = {First = "Steve";
                         Last = "Holt";
                         Age = 17;}

> // Использование '.field'
// для доступа к полям записи
printfn "%s is %d years old" steve.First steve.Age;;
Steve is 17 years old
val it : unit = ()
```



Записи

Опытному разработчику приложений на платформе .NET записи могут показаться упрощенной версией стандартных классов .NET. В языке F# легко можно добавлять свойства и методы, тогда зачем вообще могут понадобиться записи? Записи предлагают ряд явных преимуществ перед традиционными объектно-ориентированными структурами данных:

- Механизм вывода типов способен автоматически определять типы записей, без дополнительных аннотаций типов.
- Поля записи по умолчанию являются неизменяемыми, тогда как классы не обладают такой особенностью.
- Записи не могут наследоваться, гарантируя свою неизменность на будущее.
- Записи могут использоваться в выражениях сопоставления с образцом, тогда как классы – нет, без применения активных шаблонов или ограничения when.



Записи

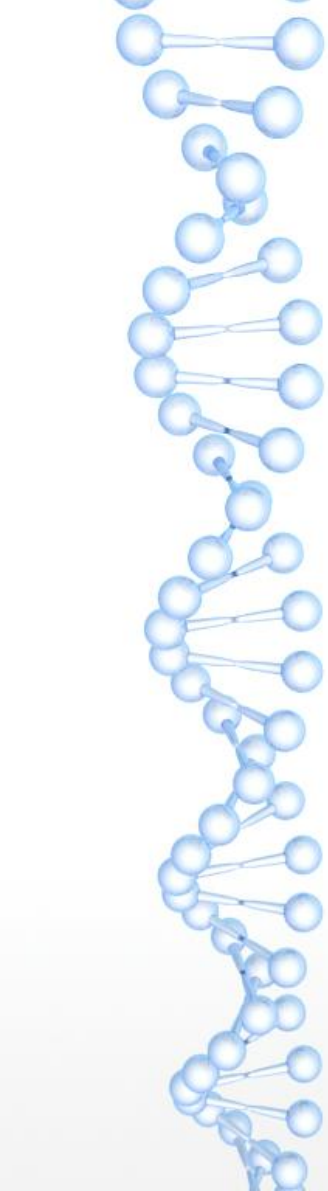
Клонирование записей

Записи легко могут клонироваться с помощью ключевого слова `with`:

```
type Car =
    {
        Make : string
        Model : string
        Year : int
    }

let thisYear's = { Make = "FSharp"; Model = "Luxury Sedan"; Year = 2010 }
let nextYear's = { thisYear's with Year = 2011 }
```

Что эквивалентно следующему:

```
let nextYear's =
    {
        Make  = thisYear's.Make
        Model = thisYear's.Model
        Year  = 2011
    }
```



Записи

В следующем примере выполняется фильтрация исходного списка записей типа `Car`, чтобы в новом списке остались только записи, в которых поле `Model` имеет значение `"Coup"`:

```
let allCoups =
    allNewCars
    |> List.filter
       (function
         | { Model = "Coup" } -> true
         | _                  -> false)
```

Пример 3.9. Определение типов записей

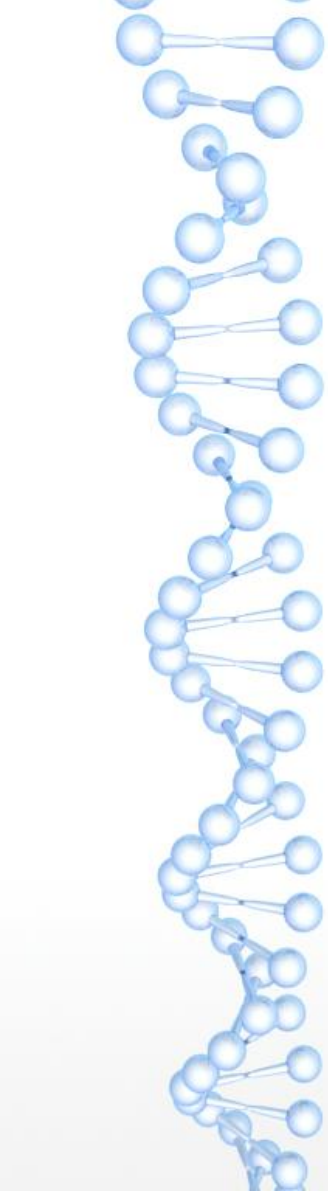
```
> type Point = { X : float; Y : float };;

type Point =
  {X: float;
   Y: float;}

> // Расстояние между двумя точками
let distance pt1 pt2 =
    let square x = x * x
    sqrt <| square (pt1.X - pt2.X) + square (pt1.Y - pt2.Y);;

val distance : Point -> Point -> float

> distance {X = 0.0; Y = 0.0} {X = 10.0; Y = 10.0};;
val it : float = 14.14213562
```

Записи

Методы и свойства

Как и для размеченных объединений, вы можете добавлять к записям свойства и методы:

```
> // Добавление свойства к записи типа Vector =
type Vector =
    { X : float; Y : float; Z : float }
    member this.Length =
        sqrt <| this.X ** 2.0 + this.Y ** 2.0 + this.Z ** 2.0;;

type Vector =
  {X: float;
   Y: float;
   Z: float;}
  with
    member Length : float
  end

> let v = { X = 10.0; Y = 20.0; Z = 30.0 };;

val v : Vector = {X = 10.0;
                  Y = 20.0;
                  Z = 30.0;}

> v.Length;;
val it : float = 37.41657387
```



Отложенные вычисления

Пример 3.10. Использование отложенных вычислений

```
> // Определяются два отложенных значения
let x = Lazy<int>.Create(fun () -> printfn "Evaluating x..."; 10)
let y = lazy (printfn "Evaluating y..."; x.Value + x.Value);;

val x : Lazy<int> = <unevaluated>
val y : Lazy<int> = <unevaluated>

> // Прямое обращение к значению y приводит к его вычислению
y.Value;;
Evaluating y...
Evaluating x...
val it : int = 20

> // При повторном обращении к значению y используется сохраненное ранее
// значение (побочные эффекты отсутствуют)
y.Value;;
val it : int = 20
```

Отложенные вычисления

```
> let seqOfNumbers = seq { 1 .. 5 };;

val seqOfNumbers : seq<int>

> seqOfNumbers |> Seq.iter (printfn "%d");;
1
2
3
4
5
val it : unit = ()
```

Различия между типами `seq` и `list` заключается в том, что в каждый конкретный момент времени в памяти существует только один элемент последовательности. Тип `seq` – это всего лишь псевдоним для интерфейса `System.Collections.Generic.IEnumerable <'a>` в платформе .NET.

Тогда зачем нужны эти ограничения? Зачем нам два типа, когда мы просто можем использовать тип `list`? Так как содержимое списка целиком хранится в памяти, это означает, что вы ограничены объемом имеющихся ресурсов. Вы легко можете определить бесконечную последовательность, но бесконечный список просто не поместится в память. Кроме того, при использовании списков значение каждого его элемента должно быть известно заранее, тогда как последовательности могут развертываться динамически (это так называемая «pull»-модель).

Отложенные вычисления

Пример 3.11. Последовательность всех целых чисел

```
> // Последовательность всех целых чисел
let allIntsSeq = seq { for i = 0 to System.Int32.MaxValue -> i };;

val allIntsSeq : seq<int>

> allIntsSeq;;
val it : seq<int> = seq [0; 1; 2; 3; ...]
> // Список всех целых чисел - ОШИБКА: не умещается в памяти!
let allIntsList = [ for i = 0 to System.Int32.MaxValue do yield i ];;
System.OutOfMemoryException: Exception of type 'System.OutOfMemoryException' was
thrown.
```

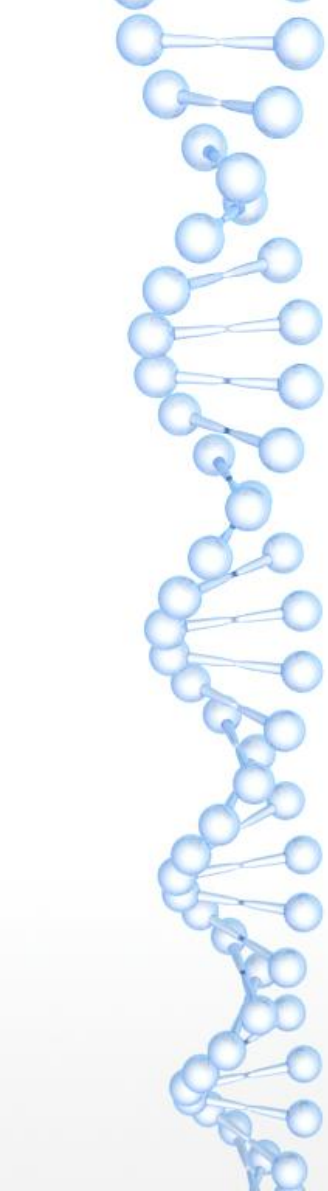
Отложенные вычисления

Выражения последовательности

Для определения последовательностей можно использовать синтаксис генераторов списков (такие конструкции называются *выражениями последовательности (sequence expressions)*). Определение последовательности начинается с ключевого слова `seq`, за которым следуют фигурные скобки:

```
> let alphabet = seq { for c in 'A' .. 'Z' -> c };;

val alphabet : seq<char>

> Seq.take 4 alphabet;;
val it : seq<char> = seq ['A'; 'B'; 'C'; 'D']
```

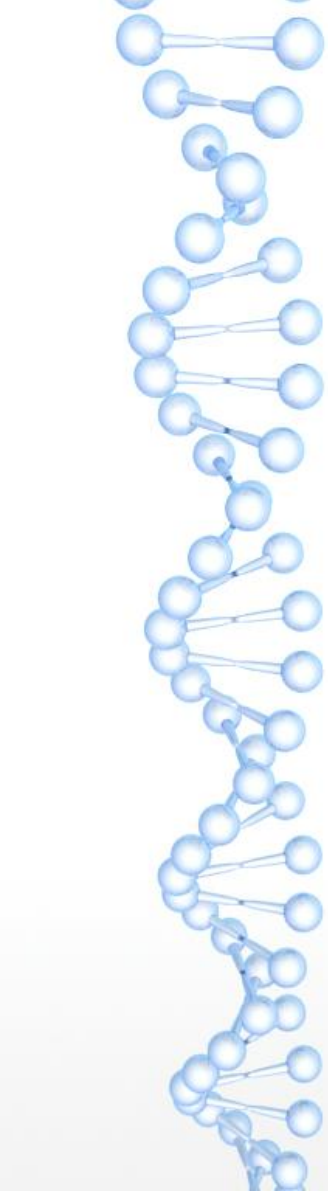
Последовательности вычисляются отложенно, поэтому каждый раз при получении очередного элемента выполняется код выражения последовательности. Попробуем выполнить предыдущий пример еще раз, но теперь добавим побочный эффект, который будет выполняться при возврате каждого элемента. Теперь выражение будет не только возвращать алфавитные символы, но и выводить их в консоль:



Отложенные вычисления

```
> // Последовательность с побочным эффектом
let noisyAlphabet =
    seq {
        for c in 'A' .. 'Z' do
            printfn "Yielding %c..." c
            yield c
    };;

val noisyAlphabet : seq<char>

> let fifthLetter = Seq.nth 4 noisyAlphabet;;
Yielding A...
Yielding B...
Yielding C...
Yielding D...
Yielding E...

val fifthLetter : char = 'E'
```

Интересно отметить, что выражения последовательности могут быть рекурсивными. С помощью ключевого слова `yield!` (произносится как *йилд банг*) можно вернуть подпоследовательность, которая объединяется с основной последовательностью.



Отложенные вычисления

Seq.take

Возвращает первые *n* элементов последовательности:

```
> // Последовательность случайных чисел
open System
let randomSequence =
    seq {
        let rng = new Random()
        while true do
            yield rng.Next()
    };;

val randomSequence : seq<int>

> randomSequence |> Seq.take 3;;
val it : seq<int> = seq [2101281294; 1297716638; 1114462900]
```

Seq.unfold

Генерирует последовательность, используя указанную функцию. Имеет тип `('a -> ('b * 'a) option) -> 'a -> seq<'b>`.

Отложенные вычисления

Пример 3.13. Использование функции Seq.unfold

```
> // Генерирует очередной элемент последовательности чисел Фибоначчи на основе
// двух предыдущих элементов. Предназначена для использования функцией
// Seq.unfold.
let nextFibUnder100 (a, b) =
    if a + b > 100 then
        None
    else
        let nextValue = a + b
        Some(nextValue, (nextValue, a));;

val nextFibUnder100 : int * int -> (int * (int * int)) option

> let fibsUnder100 = Seq.unfold nextFibUnder100 (0, 1);;

val fibsUnder100 : seq<int>

> Seq.tolist fibsUnder100;;
val it : int list = [1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89]
```

Отложенные вычисления

Таблица 3.1. Часто используемые функции из модуля Seq

Функция и ее тип	Описание
`Seq.length` `seq<'a> -> int`	Возвращает длину последовательности.
`Seq.exists` `('a -> bool) -> seq<'a> -> bool`	Возвращает признак наличия в последовательности элемента, который удовлетворяет функции поиска.
`Seq.tryFind` `('a -> bool) -> seq<'a> -> 'a option`	Возвращает `Some(x)` для первого элемента `x`, для которого указанная функция вернет `true`. В противном случае вернет `None`.
`Seq.filter` `('a -> bool) -> seq<'a> -> seq<'a>`	Отфильтровывает все элементы последовательности, для которых указанная функция возвращает `false`.
`Seq.concat` `(seq< #seq<'a> > -> seq<'a>`	Объединяет серию последовательностей в единую последовательность `seq`.

Агрегатные операторы

Кроме этого, модуль `Seq` содержит агрегатные операторы, аналогичные операторам в модуле `List`.