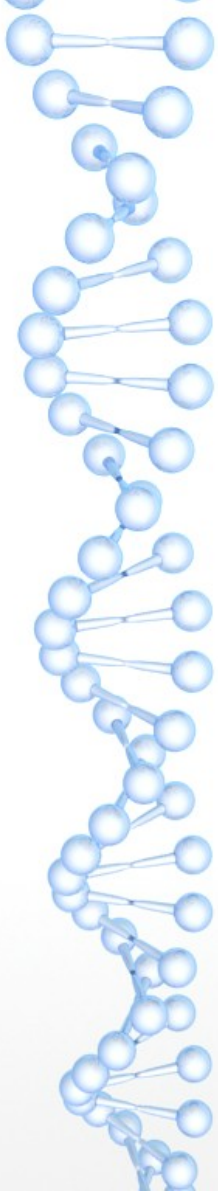


Функциональное и логическое программирование

Лекция 5. Введение в F#

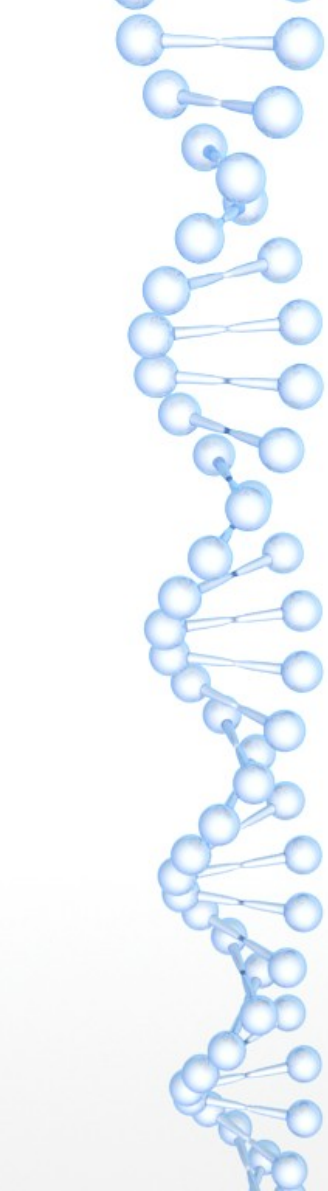
Первая программа

```
(*
Mega Hello World:
Принимает два параметра командной строки и выводит их
вместе со значением текущего времени в окно консоли.
*)

open System

[<EntryPoint>]
let main (args : string[]) =

    if args.Length <> 2 then
        failwith "Error: Expected arguments <greeting> and <thing>"

    let greeting, thing = args.[0], args.[1]
    let timeOfDay = DateTime.Now.ToString("hh:mm tt")

    printfn "%s, %s at %s" greeting thing timeOfDay

    // Код завершения программы
    0
```

Вторая программа

```
/// Compute the greatest common divisor of
/// two numbers.
let rec gcd x y =
    if y = 0 then x
    else gcd y (x % y)

let x = gcd 1024 12
```

```
val gcd : int -> int -> int

Compute the greatest common divisor of
two numbers.

Full name: ProgrammingFS.Ch1.gcd
```

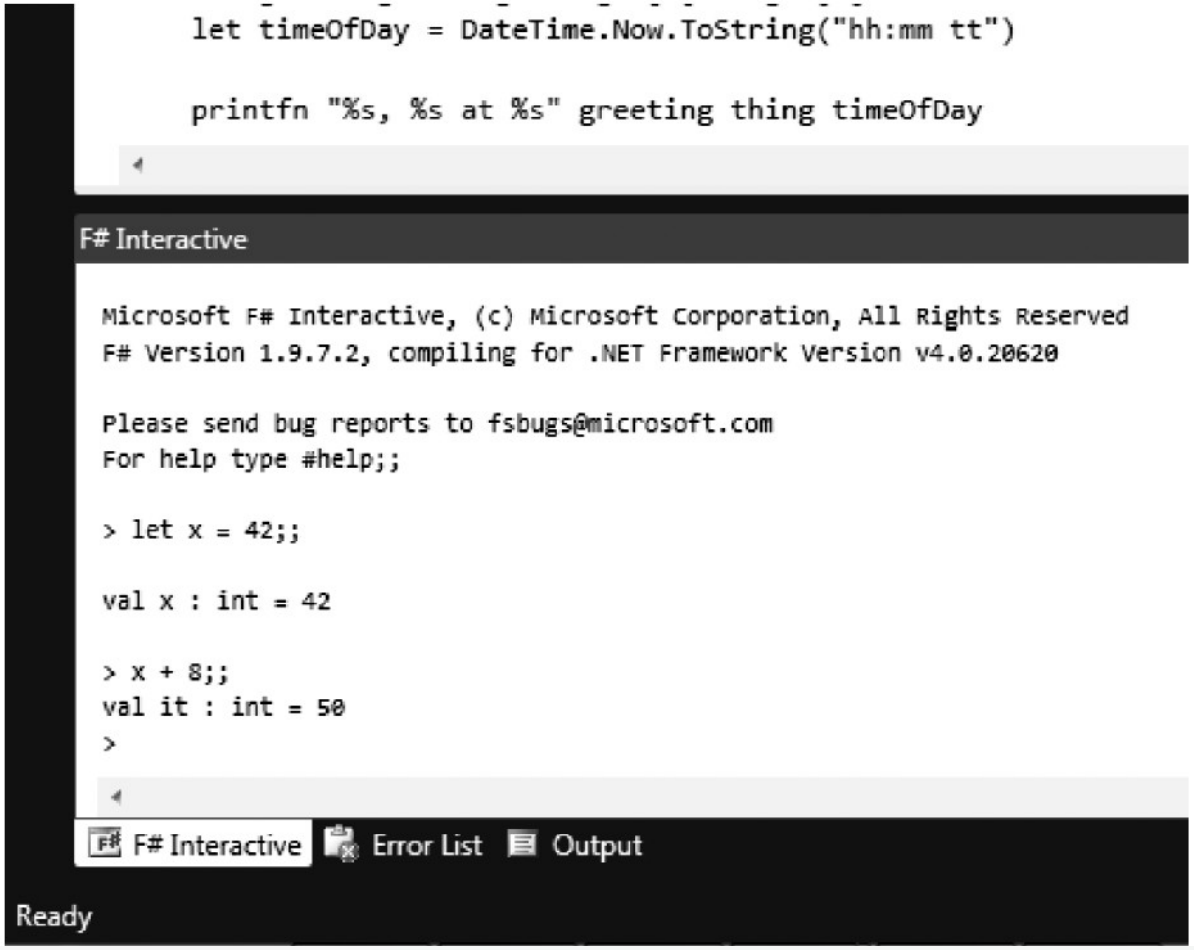


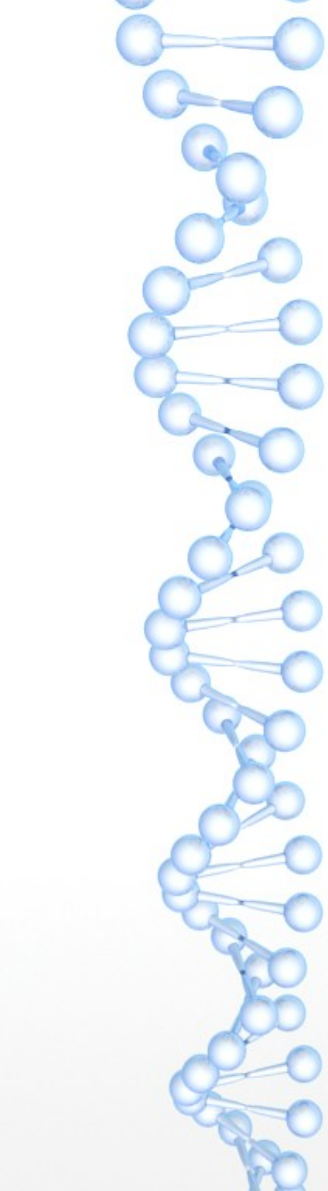
Interactive

В Visual Studio с типичными настройками окно F# Interactive можно открыть, нажав комбинацию клавиш Control+Alt+F. Как только окно FSI будет открыто, в него можно вводить код F#, пока вы не введете последовательность ;; и символ перевода строки. Введенный код будет скомпилирован и выполнен, как показано на рис. 1.4.

После выполнения каждого фрагмента в окне FSI все имена, созданные в нем, будут выводиться в виде `val <имя>`. Если для выражения не было указано имя, ему автоматически будет присвоено имя `it`. Вслед за именем идентификатора будет выводиться символ двоеточия, *тип* результата и фактическое значение. Например, на рис. 1.4 видно, что было создано значение типа `int` с именем `x` и со значением, равным `42`.

Interactive

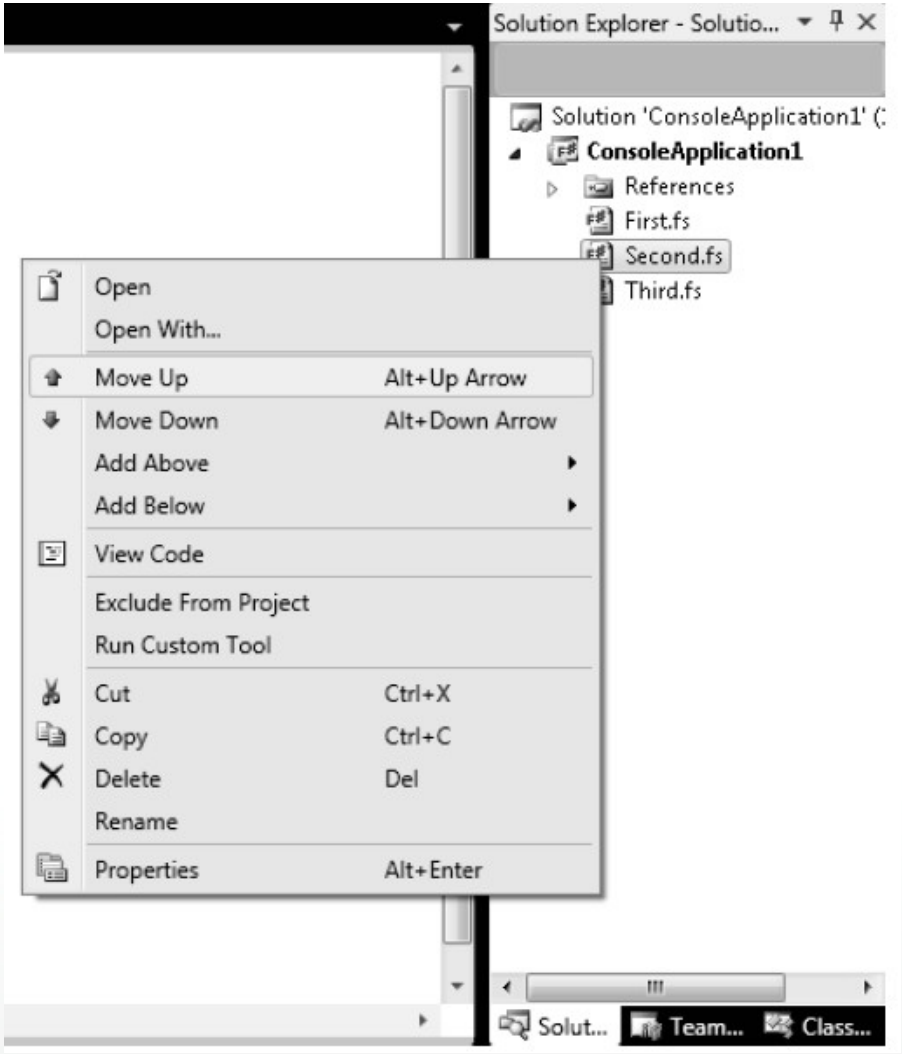


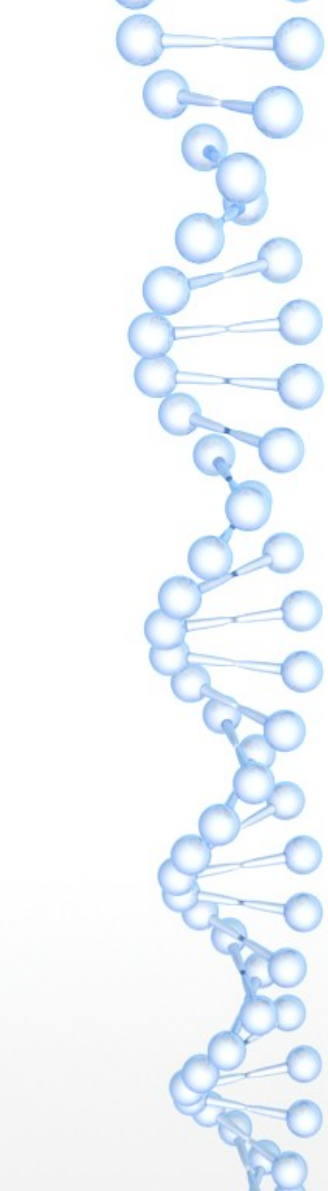


Interactive

```
> 2 + 2;;
val it : int = 4
> // Создаем два значения
let x = 1
let y = 2.3;;

val x : int = 1
val y : float = 2.3

> float x + y;;
val it : float = 3.3
> let cube x = x * x * x;;

val cube : int -> int

> cube 4;;
val it : int = 64
```

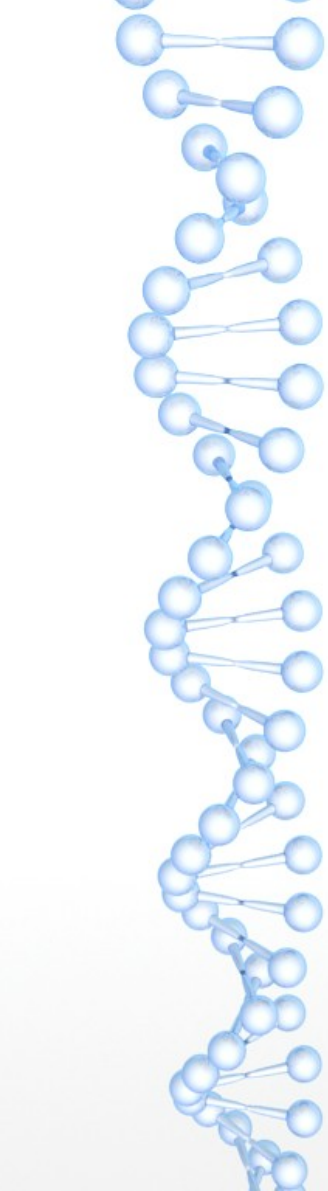
Solution


Solution Explorer - Solutio...
Solution 'ConsoleApplication1' (
ConsoleApplication1
References
First.fs
Second.fs
Third.fs
Open
Open With...
Move Up Alt+Up Arrow
Move Down Alt+Down Arrow
Add Above
Add Below
View Code
Exclude From Project
Run Custom Tool
Cut Ctrl+X
Copy Ctrl+C
Delete Del
Rename
Properties Alt+Enter
Solut...
Team...
Class...



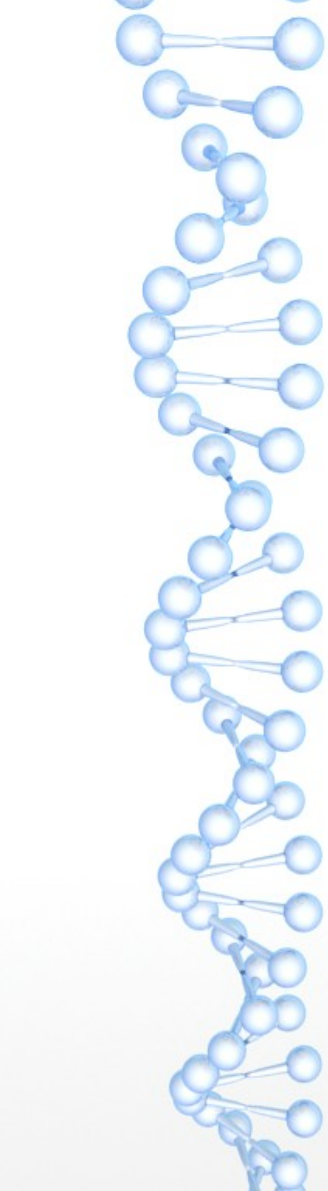
Элементарные числовые типы

```
> let answerToEverything = 42UL;;

val answerToEverything : uint64 = 42UL

> let pi = 3.1415926M;;

val pi : decimal = 3.1415926M

> let avogadro = 6.022e23;;

val avogadro : float = 6.022e+23
```



Элементарные числовые типы

Тип	Суффикс	Тип .NET	Диапазон
byte	uy	System.Byte	от 0 до 255
sbyte	y	System.SByte	от –128 до 127
int16	s	System.Int16	от –32768 до 32767
uint16	us	System.UInt16	от 0 до 65535
int, int32		System.Int32	от -2^{31} до $2^{31}-1$
uint32	u	System.UInt32	от 0 до $2^{32}-1$
int64	L	System.Int64	от -2^{63} до $2^{63}-1$
uint64	UL	System.UInt64	от 0 до $2^{64}-1$
float		System.Double	Вещественное двойной точности согласно стандарту IEEE64. Представляет значения, состоящие примерно из 15 значащих цифр.
float32	f	System.Single	Вещественное одинарной точности согласно стандарту IEEE32. Представляет значения, состоящие примерно из 7 значащих цифр.
decimal	M	System.Decimal	Вещественный тип фиксированной точности хранит точно 28 цифр после запятой.



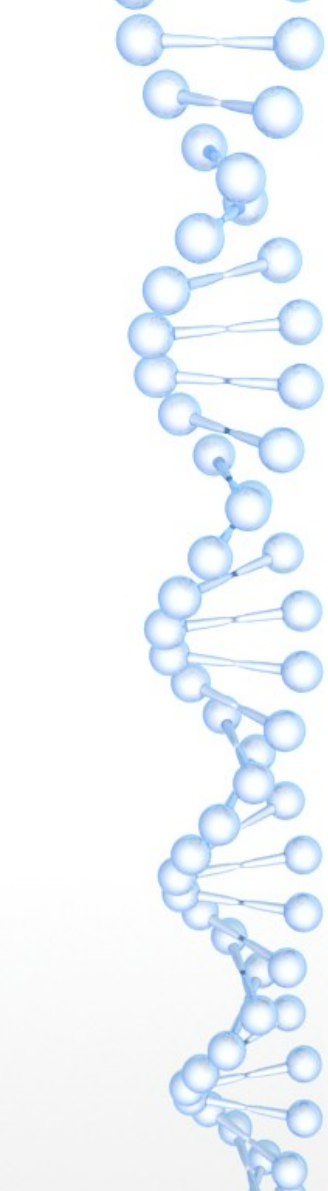
Элементарные числовые типы

```
> let hex = 0xFCAF;;

val hex : int = 64687

> let oct = 0o7771L;;

val oct : int64 = 4089L

> let bin = 0b00101010y;;

val bin : sbyte = 42y

> (hex, oct, bin);;

val it : int * int64 * sbyte = (64687, 4089L, 42y)
```

Элементарные числовые типы

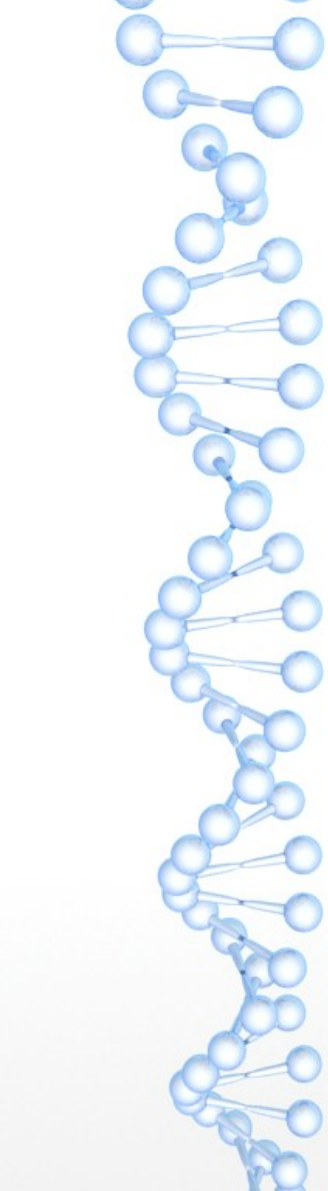
Оператор	Описание	Пример	Результат
+	Сложение	1 + 2	3
-	Вычитание	1 - 2	-1
*	Умножение	2 * 3	6
/	Деление	8L / 3L	2L
**	Возведение в степень[a]	2.0 ** 8.0	256.0
%	Деление по модулю (остаток от деления)	7 % 3	1

[a] Оператор ** возведения в степень может применяться только к значениям типов `float` и `float32`. Чтобы возвести в степень целочисленное значение, его необходимо либо преобразовать в вещественное число, либо использовать функцию `pown`.



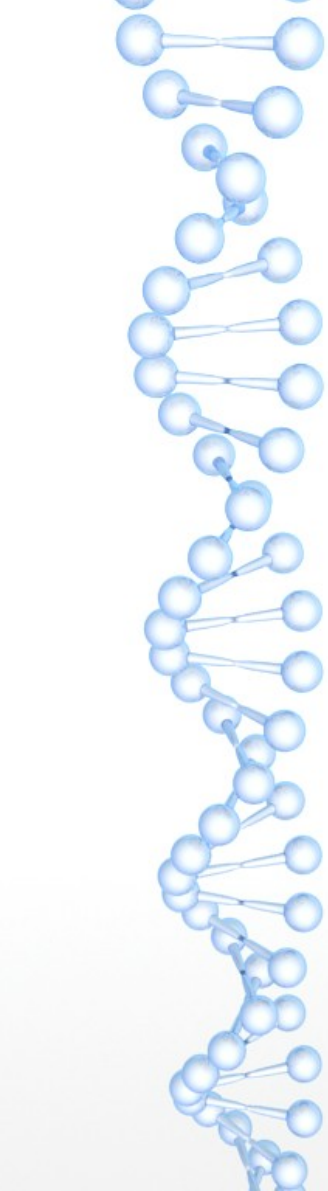
Элементарные числовые типы

Функция	Описание	Пример	Результат
abs	Абсолютное значение числа	abs -1.0	1.0
ceil	Округление вверх до ближайшего целого	ceil 9.1	10
exp	Возведение e в степень	exp 1.0	2.718
floor	Округление вниз до ближайшего целого	floor 9.9	9.0
sign	Знак числа	sign -5	-1
log	Натуральный логарифм	log 2.71828	1.0
log10	Логарифм по основанию 10	log10 1000.0	3
sqrt	Корень квадратный	sqrt 4.0	2.0
cos	Косинус	cos 0.0	1.0
sin	Синус	sin 0.0	0.0
tan	Тангенс	tan 1.0	1.557
pown	Возведение целого числа в степень	pown 2L 10	1024L



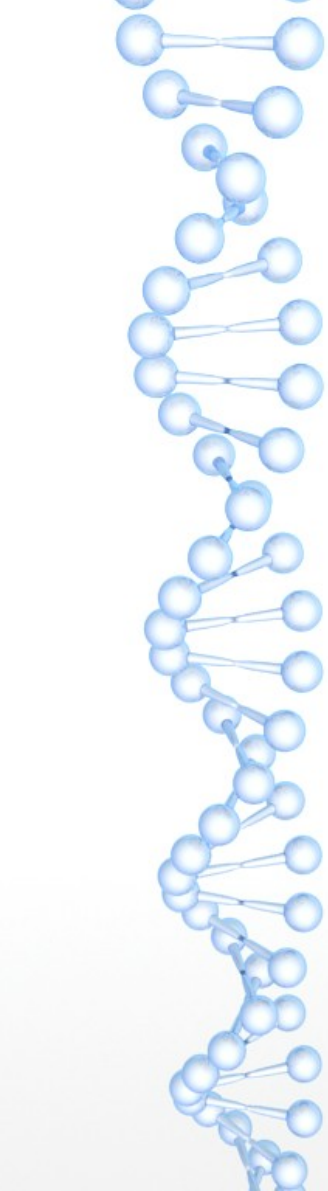
Элементарные числовые типы

Функция	Описание	Пример	Результат
sbyte	Преобразует значение в тип sbyte	sbyte -5	-5y
byte	Преобразует значение в тип byte	byte "42"	42uy
int16	Преобразует значение в тип int16	int16 'a'	97s
uint16	Преобразует значение в тип uint16	uint16 5	5us
int32, int	Преобразует значение в тип int	int 2.5	2
uint32	Преобразует значение в тип uint32	uint32 0xFF	255
int64	Преобразует значение в тип int64	int64 -8	-8L
uint64	Преобразует значение в тип uint64	uint64 "0xFF"	255UL
float	Преобразует значение в тип float	float 3.1415M	3.1415
float32	Преобразует значение в тип float32	float32 8y	8.0f
decimal	Преобразует значение в тип decimal	decimal 1.23	1.23M



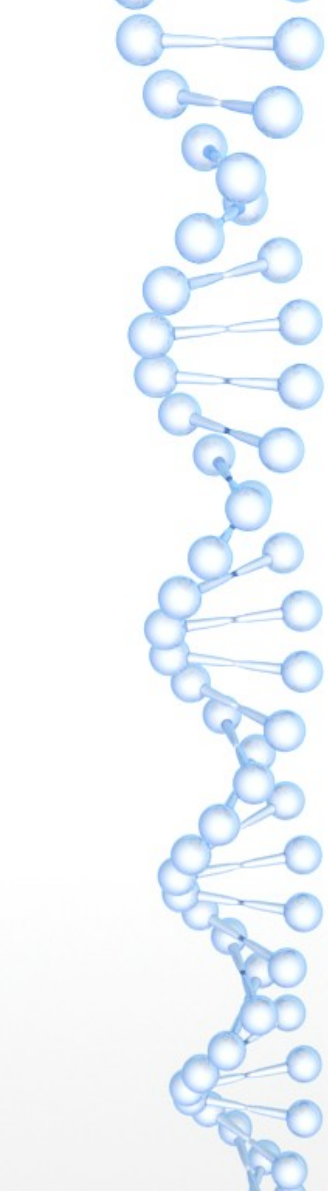
Элементарные числовые типы

```
> open System.Numerics

// Единицы измерения объема данных
let megabyte = 1024I * 1024I
let gigabyte = megabyte * 1024I
let terabyte = gigabyte * 1024I
let petabyte = terabyte * 1024I
let exabyte = petabyte * 1024I
let zettabyte = exabyte * 1024I;;

val megabyte : BigInteger = 1048576
val gigabyte : BigInteger = 1073741824
val terabyte : BigInteger = 1099511627776
val petabyte : BigInteger = 1125899906842624
val exabyte : BigInteger = 1152921504606846976
val zettabyte : BigInteger = 1180591620717411303424
```



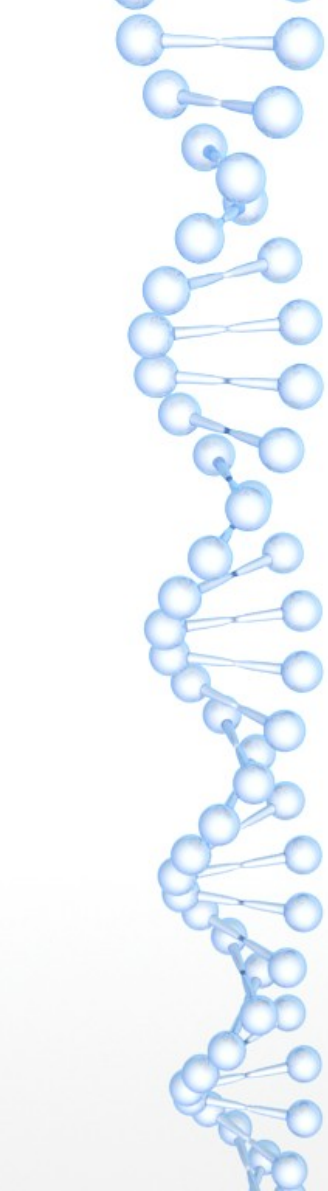
Элементарные числовые типы

Оператор	Описание	Пример	Результат
&&&	Битовое «И»	0b1111 &&& 0b0011	0b0011
|||	Битовое «ИЛИ»	0xFF00 ||| 0x00FF	0xFFFF
^^^	Битовое «Исключающее ИЛИ»	0b0011 ^^^ 0b0101	0b0110
<<<	Сдвиг влево	0b0001 <<< 3	0b1000
>>>	Сдвиг вправо	0b1000 >>> 3	0b0001



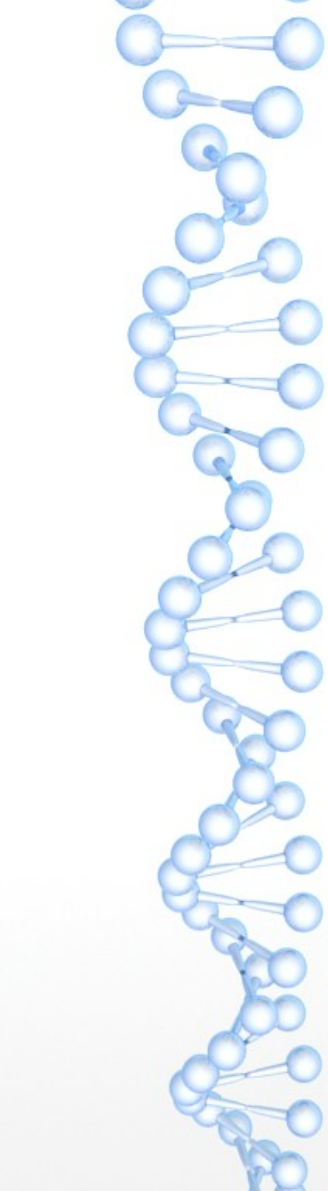
Символы

```
> let vowels = [‘a’; ‘e’; ‘i’; ‘o’; ‘u’];;

val vowels : char list = [‘a’; ‘e’; ‘i’; ‘o’; ‘u’]

> printfn “Hex u0061 = ‘%c’” ‘\u0061’;;
Hex u0061 = ‘a’
val it : unit = ()
```



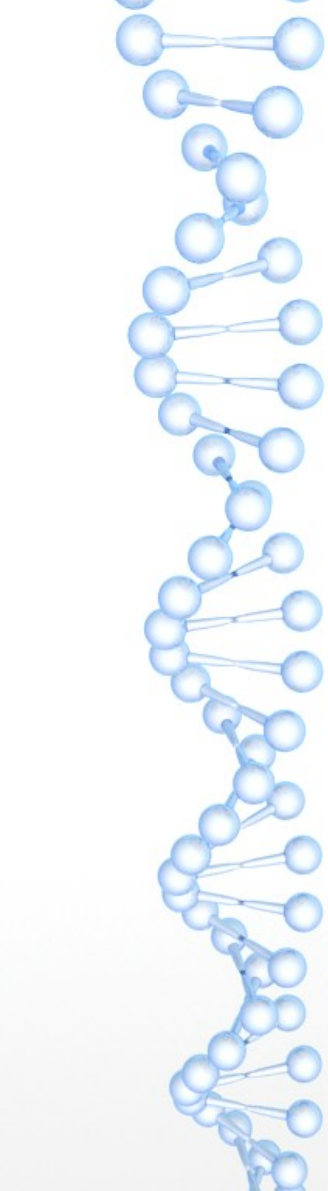
Символы

Символ	Значение
\'	Апостроф
\"	Кавычка
\\	Обратный слеш
\b	Забой (Backspace)
\n	Перевод строки
\r	Возврат каретки
\t	Горизонтальная табуляция



Символы

```
> // Преобразовать символ 'C' в целое число
int 'C';;
val it : int = 67
> // Преобразовать символ 'C' в байт
'C'B;;
val it : byte = 67uy
```

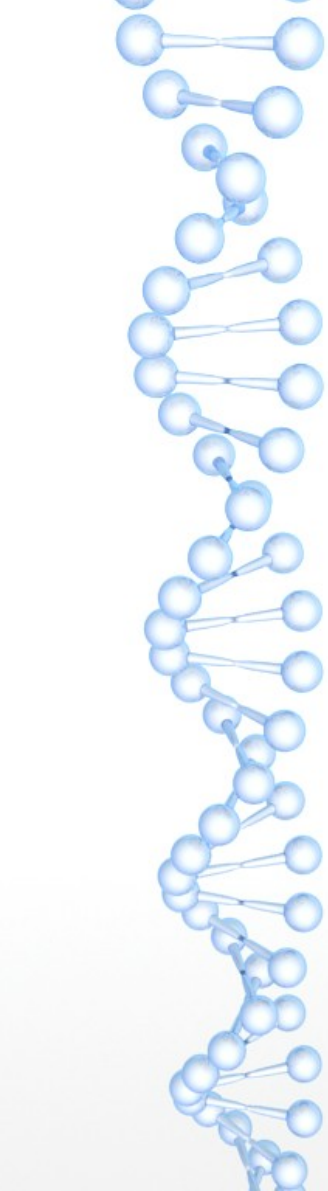


Строки

```
> let password = "abracadabra";;

val password : string = "abracadabra"

> let multiline = "This string
takes up
multiple lines";;

 val multiline : string = "This string
 takes up
 multiple lines"

 > multiline.[0];;
 val it : char = 'T'
 > multiline.[1];;
 val it : char = 'h'
 > multiline.[2];;
 val it : char = 'i'
 > multiline.[3];;
 val it : char = 's'
```

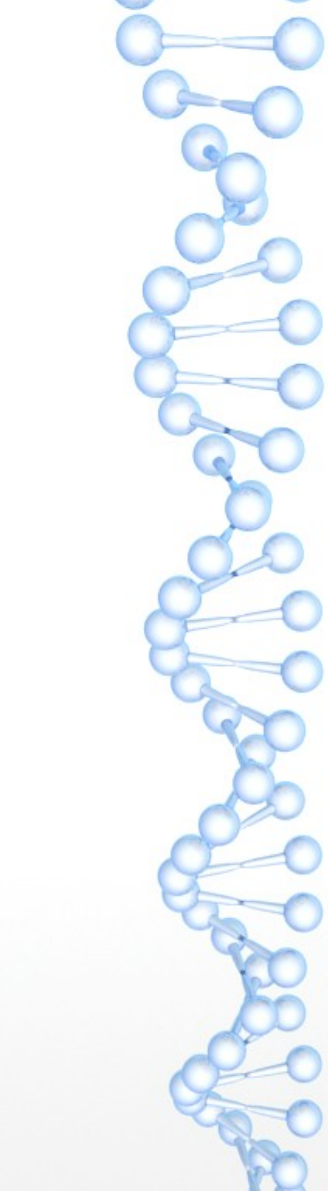
Строки

```
> let longString = "abc-\
                   def-\
                  ghi";;

> longString;;
val longString : string = "abc-def-ghi"

> let normalString = "Normal.\n.\n.\t.\t.String";;

val normalString : string = "Normal.
.
.       .       .String

> let verbatimString = @"Verbatim.\n.\n.\t.\t.String";;

val verbatimString : string = "Verbatim.\n.\n.\t.\t.String"

 > let hello = "Hello"B;;

 val hello : byte []= [|72uy; 101uy; 108uy; 108uy; 111uy|]
```



Булевы значения и функции

Оператор	Описание	Пример	Результат
`&&`	**Логическое «И»**	`true && false`	`false`
`||`	**Логическое «ИЛИ»**	`true || false`	`true`
`not`	**Логическое «НЕ»**	`not false`	`true`



Сравнение

Оператор	Описание	Пример	Результат
<	Меньше	1 < 2	true
<=	Меньше или равно	4.0 <= 4.0	true
>	Больше	1.4e3 > 1.0e2	true
>=	Больше или равно	0I >= 2I	false
=	Равно	“abc” = “abc”	true
<>	Не равно	‘a’ <> ‘b’	true
compare	Сравнивает два значения	compare 31 31	0

Функции

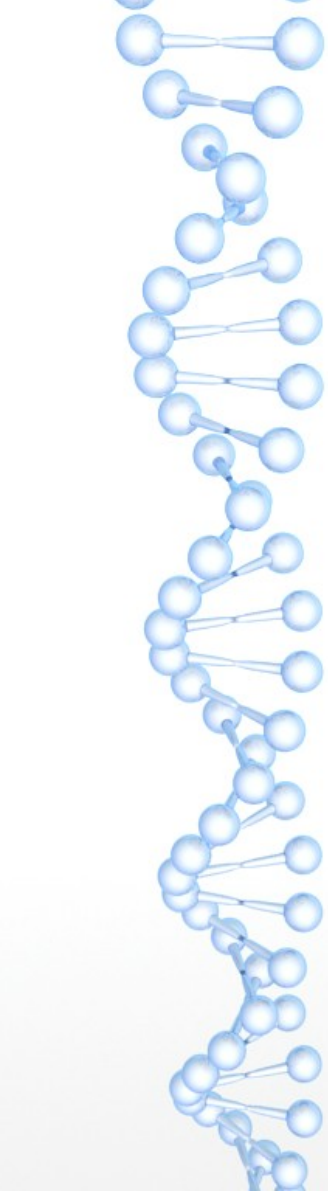
```
> let square x = x * x;;

val square : int -> int

> square 4;;
val it : int = 16
```

```
> let addOne x = x + 1;;

val addOne : int -> int
```

```
> let add x y = x + y;;

val add : int -> int -> int
```

```
> add 1 2;;

val it : 3
```

```
> let add x y = x + y;;

val add : x:int -> y:int -> int

> add 3
;;
val it : (int -> int) = <fun:it@31>
>
```

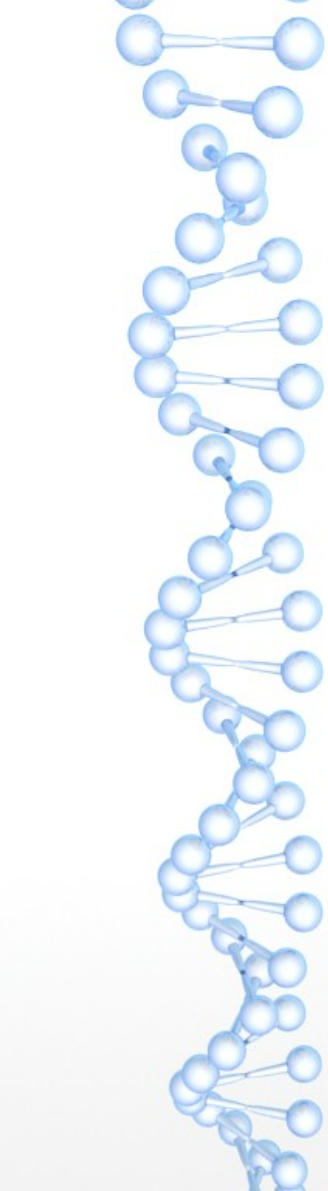


Функции

Оператор + может применяться к различным типам, таким как `byte`, `int` и `decimal`, поэтому при отсутствии дополнительной информации о типах компилятор по умолчанию просто предполагает, что аргументы имеют тип `int`.

Ниже приводится сеанс в окне FSI, где объявляется функция, которая возвращает произведение двух значений. Так же как и в случае использования оператора +, компилятор делает вывод, что ей будут передаваться целочисленные значения, потому что никакой дополнительной информации об ее использовании ему не предоставляется:

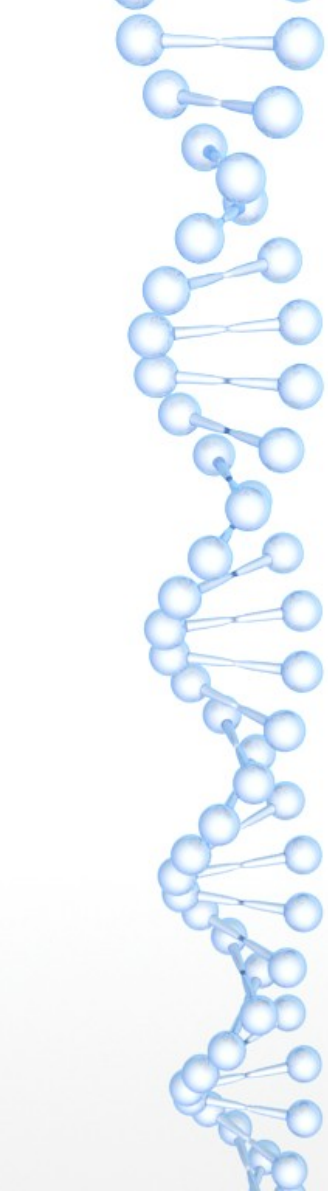
```
> // Нет никакой дополнительной информации для вывода типов
  // на основе использования этой функции
let mult x y = x * y;;

val mult : int -> int -> int
```



Функции

Если в интерактивном сеансе помимо определения функции `mult` добавить ее вызов с двумя вещественными значениями, сигнатура функции будет выведена как `float -> float -> float`:

```
> // Механизм выведения типов в действии
let mult x y = x * y
let result = mult 4.0 5.5;;

val mult : float -> float -> float
val result : float = 22.0
```



Функции

При этом имеется возможность явно добавить *аннотацию типов (type annotation)*, или подсказку для компилятора F# о типах аргументов. Чтобы добавить аннотацию типов, достаточно просто заменить параметр функции следующим образом:

```
ident -> (ident : type)
```

где *type* — это тип параметра. Чтобы указать, что первый аргумент нашей функции `add` имеет тип `float`, достаточно просто переопределить функцию, как показано ниже:

```
> let add (x : float) y = x + y;;

val add : float -> float -> float
```

Обратите внимание, как из-за добавления аннотации типа для `x` тип функции изменился на `float -> float -> float`. Дело в том, что единственная перегруженная версия оператора +, принимающего вещественное число в первом операнде, имеет тип `float -> float -> float`, поэтому компилятор F# выводит тип аргумента `y` как `float`.

Функции

```
> let ident x = x;;

val ident : 'a -> 'a

> ident "a string";;
val it : string = "a string"
> ident 1234L;;
val it : int64 = 1234L
```

Поскольку в данной ситуации механизм вывода типов не смог определить конкретный тип аргумента `x` функции `ident`, тип аргумента стал обобщенным. Если параметр становится *обобщенным (generic)*, это означает, что он может быть значением любого типа, таким как целое число, строка или вещественное число.

Имеется возможность явно объявлять обобщенные параметры, указывая в качестве имени типа любой допустимый идентификатор, перед которым стоит апостроф. Обычно для этих целей используются алфавитные символы, начиная с «a». В следующем фрагменте приводится определение функции `ident` с использованием аннотации типа, которая явно объявляет параметр `x` как обобщенный:

```
> let ident2 (x : 'a) = x;;

val ident2 : 'a -> 'a
```

Написание обобщенного кода имеет большое значение для его повторного использования. Мы продолжим обсуждение вопросов, связанных с механизмом вывода типов и обобщенными функциями, далее в этой книге, поэтому пока не следует стремиться вникнуть во все подробности. Просто имейте в виду, что всякий раз, когда вы увидите тип `'a`, это может быть тип `int`, `float`, `string`, пользовательский тип и так далее.

Управление потоком вывода

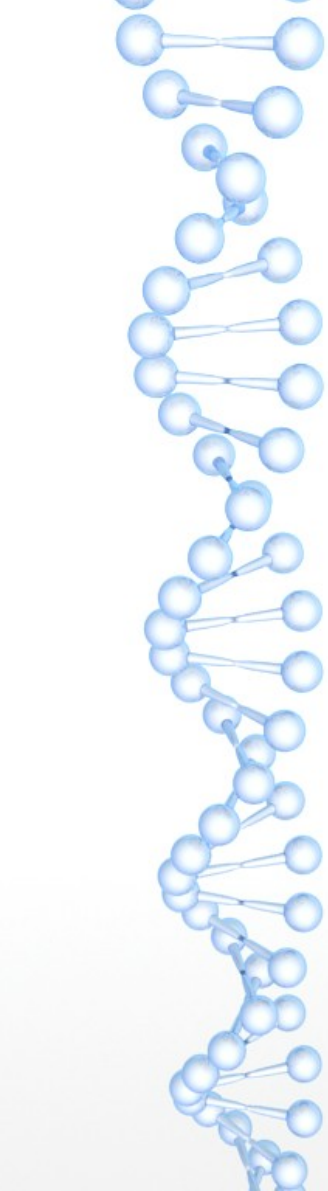
```
> // Условные инструкции
let printGreeting shouldGreet greeting =
    if shouldGreet then
        printfn "%s" greeting;;

val printGreeting : bool -> string -> unit

> printGreeting true "Hello!";;
Hello!
val it : unit = ()
> printGreeting false "Hello again!";;
val it : unit = ()
```

```
> // Условные выражения
let isEven x =
    let result =
        if x % 2 = 0 then
            "Yes it is"
        else
            "No it is not"
    result;;

val isEven : int -> string

> isEven 5;;
val it : string = "No it is not"
```

Более сложные ветвления могут быть реализованы с помощью *условных выражений (if-expression)*.

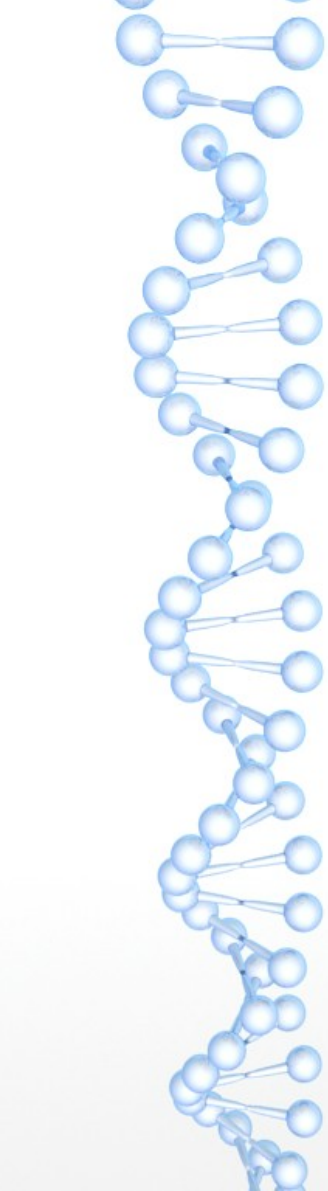
Условные выражения действуют именно так, как от них и ожидается: если условное выражение возвращает `true`, то выполняется первый блок кода, в противном случае выполняется второй блок. Однако есть одно обстоятельство, которое существенно отличает F# от других языков программирования: условные выражения в языке F# возвращают значение.

В следующем примере результат, возвращаемый условным выражением, связывается с именем `result`. То есть если условие `x % 2 = 0` выполняется, то значением `result` будет `"Yes it is"`, в противном случае `"No it is not"`:



Управление потоком вывода

```
let isWeekend day =
    if day = “Sunday” then
        true
    else
        if day = “Saturday” then
            true
        else
            false
```



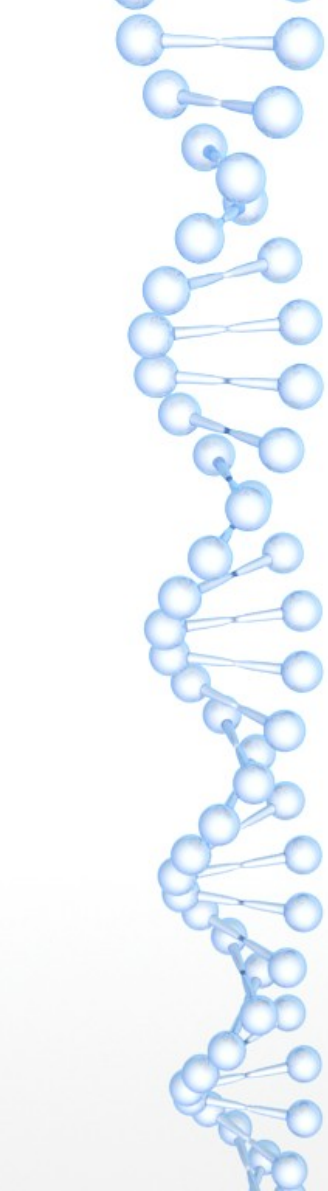
Управление потоком вывода

```
let isWeekday day =
    if day = “Monday”       then true
    elif day = “Tuesday”    then true
    elif day = “Wednesday” then true
    elif day = “Thursday”   then true
    elif day = “Friday”     then true
    else false
```

Управление потоком вывода

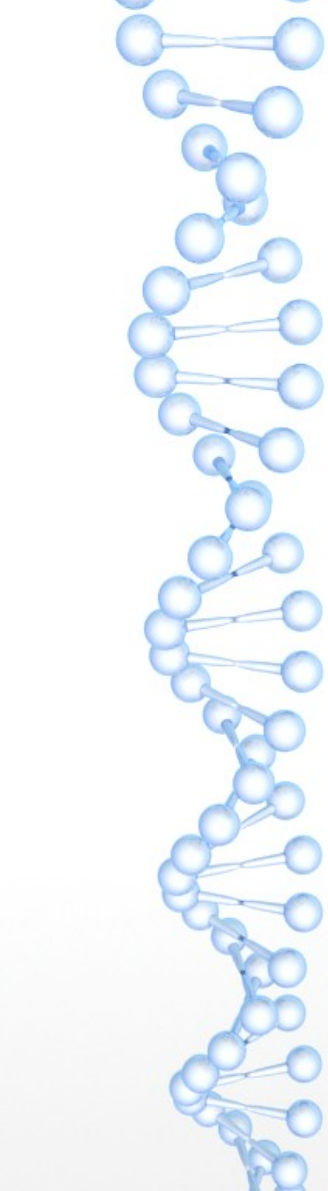
```
let x =
    if 1 > 2 then
        42
    else
        “a string”;;

      else          “a string”;;
  ------------------^^^^^^^^^^^

stdin(118,19): error FS0001: This expression was expected to have type
        int
but here has type
        string.
stopped due to error
```

(В данном выражении требовалось наличие типа int, но получен тип string, остановлено из-за ошибки.)



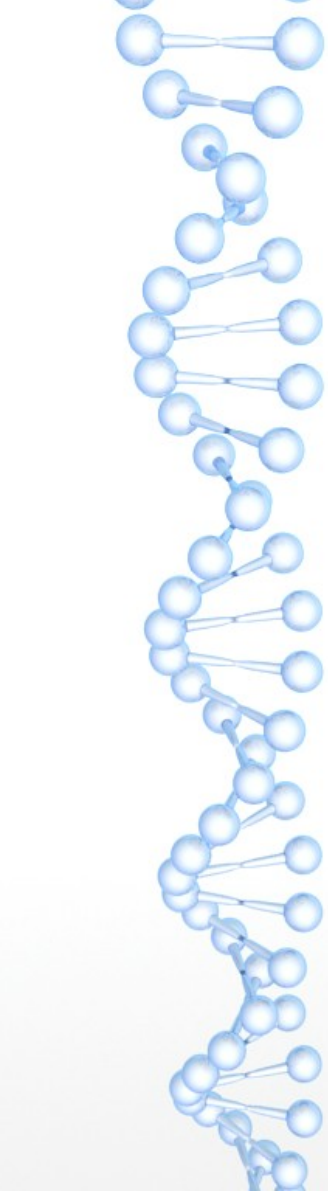
Управление потоком вывода

```
> // Тело инструкции if возвращает unit
let describeNumber x =
    if x % 2 = 0 then
        printfn “x is a multiple of 2”
    if x % 3 = 0 then
        printfn “x is a multiple of 3”
    if x % 5 = 0 then
        printfn “x is a multiple of 5”
     ();;

val describeNumber : int -> unit

> describeNumber 18;;
x is a multiple of 2
```



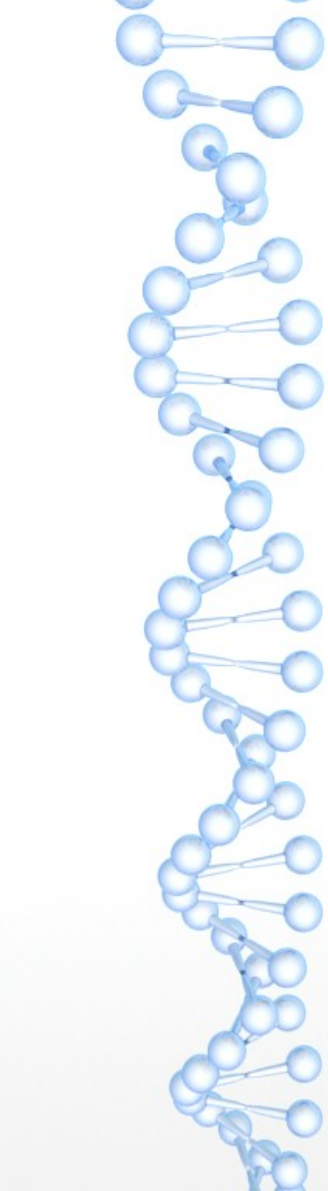
Основные типы

Сигнатура	Название	Описание	Пример
`unit`	Пустой тип	Пустое значение	`()`
`int, float`	Определенный тип	Определенный тип	`42, 3.14`
`'a, 'b`	Обобщенный тип	Обобщенный (свободный) тип	
`'a -> 'b`	Функция	Функция, возвращающая значение	`fun x -> x + 1`
`'a * 'b`	Кортеж	Упорядоченная коллекция значений	`(1, 2), ("eggs", "ham")`
`'a list`	Список	Список значений	`[ 1; 2; 3], [1 .. 3]`
`'a option`	Необязательный тип	Необязательное значение	`Some(3), None`



Основные типы

```
> let x = ();;

val x : unit

> ();;
val it : unit = ()
```



Основные типы

```
> let square x = x * x;;

val square : int -> int

> ignore (square 4);;
val it : unit = ()
```

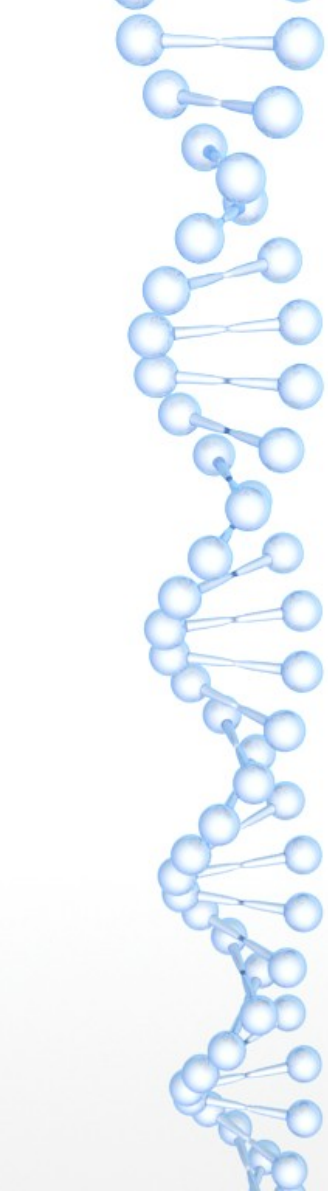
Основные типы

Кортеж (tuple) – это упорядоченная коллекция данных и наиболее простой способ группировки различных элементов в одну структуру. Например, кортежи могут использоваться для отслеживания промежуточных результатов вычислений.

В языке F# кортежи основаны на типе `System.Tuple<_>`, однако на практике вам никогда не придется использовать класс `Tuple<_>` напрямую.

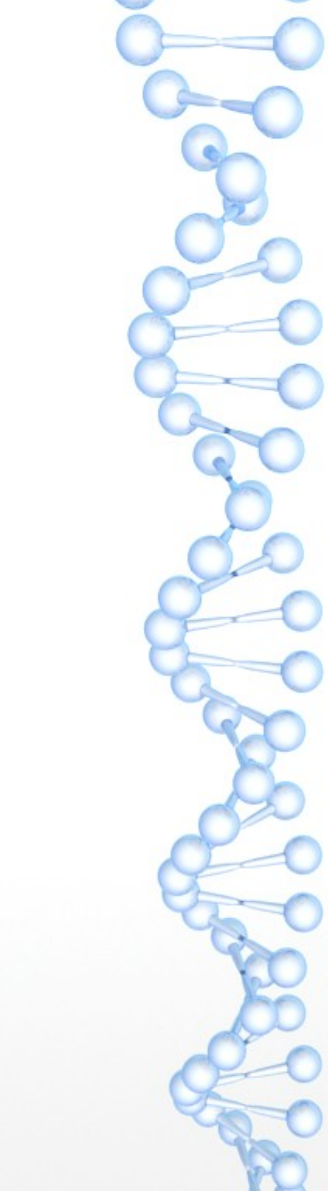
Чтобы создать экземпляр кортежа, достаточно указать список значений, разделенных запятыми и (необязательно) заключить его в круглые скобки. Тип кортежа обозначается с помощью списка типов его элементов, разделенных символами `*`. В следующем примере `dinner` – это экземпляр кортежа, а `string * string` – это тип данного кортежа:

```
> let dinner = ("green eggs", "ham");;

val dinner : string * string = ("green eggs", "ham")
```



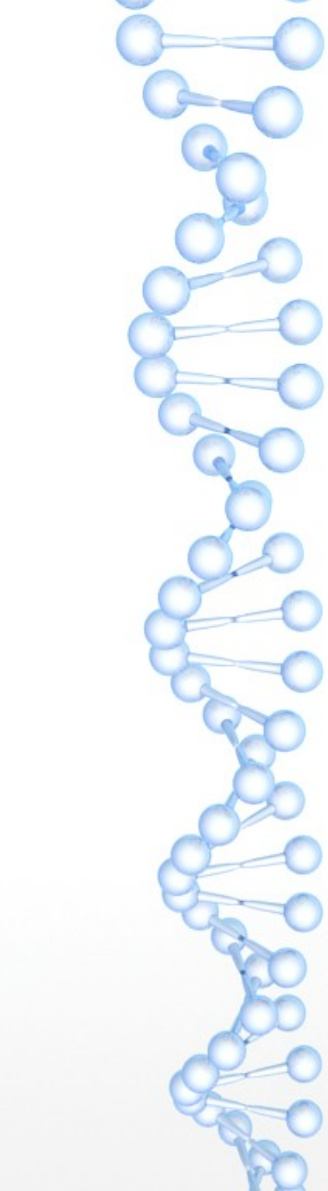
Основные типы

```
> let zeros = (0, 0L, 0I, 0.0);;

val zeros : int * int64 * bigint * float = (0, 0L, 0I, 0.0)

> let nested = (1, (2.0, 3M), (4L, "5", '6'));;

val nested : int * (float * decimal) * (int64 * string * char) = ...
```



Основные типы

```
> let nameTuple = (“John”, “Smith”);;

val nameTuple : string * string = (“John”, “Smith”)

> fst nameTuple;;
val it : string = “John”
> snd nameTuple;;
val it : string = “Smith”
```



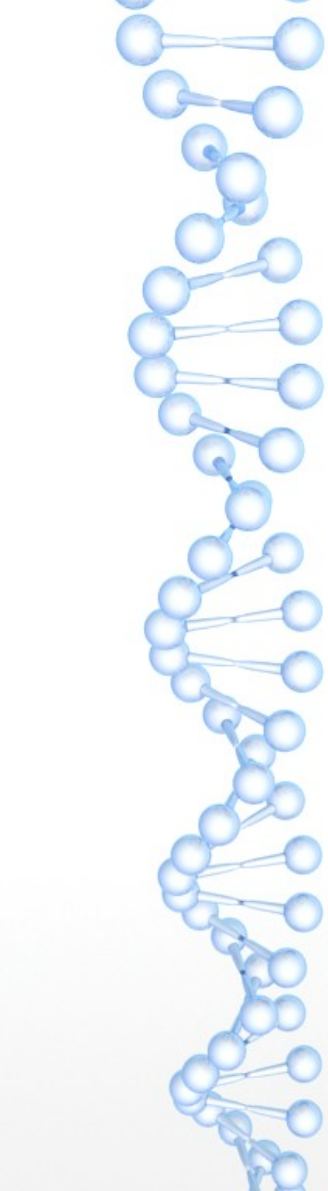
Основные типы

```
> let snacks = (“Soda”, “Cookies”, “Candy”);;

val snacks : string * string * string = (“Soda”, “Cookies”, “Candy”)

> let x, y, z = snacks;;

val z : string = “Soda”
val y : string = “Cookies”
val x : string = “Candy”

> y, z;;
val it : string * string = (“Cookies”, “Candy”)
```

Основные типы

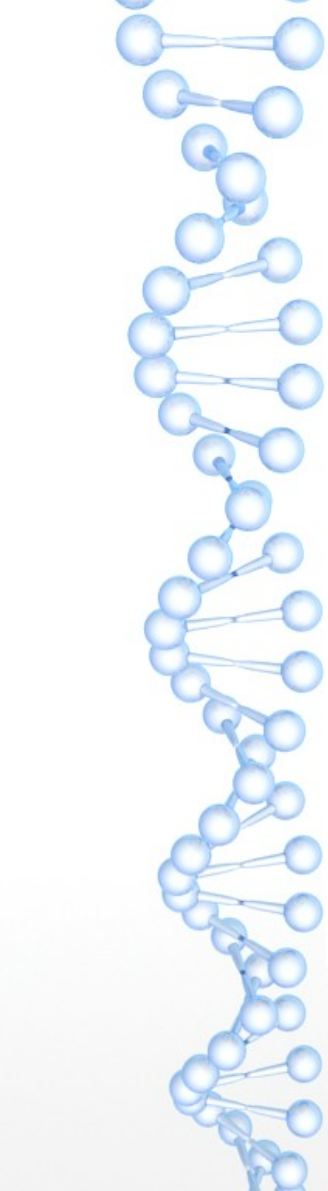
```
> let x, y = snacks;;

  let x, y = snacks;;
  -----------^^^^^^

stdin(8,12): error FS0001: Type mismatch. Expecting a
        string * string
but given a
        string * string * string.
The tuples have differing lengths of 2 and 3.
```

*(Несоответствие типов. Требуется string * string, но получен string * string * string. Кортежи имеют разную длину – 2 и 3.)*



Основные типы

```
> let add x y = x + y;;

val add : int -> int -> int

> let tupledAdd(x, y) = x + y;;

val tupledAdd : int * int -> int

> add 3 7;;
val it : int = 10

> tupledAdd(3, 7);;
val it : int = 10
```



Основные типы

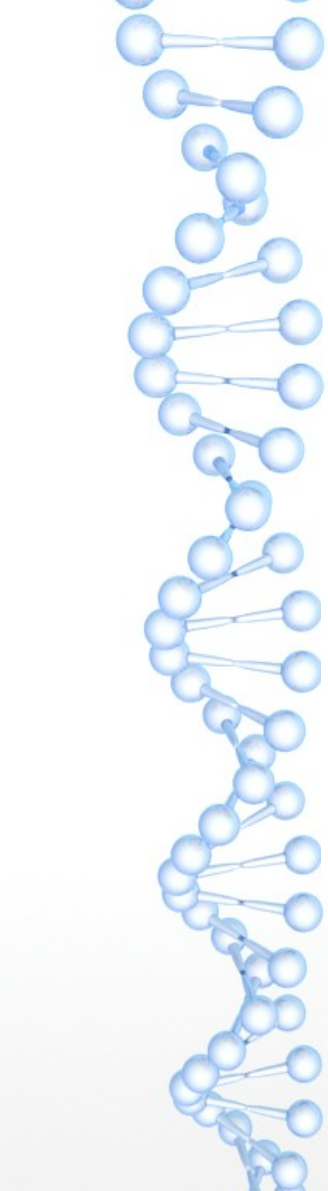
Кортежи группируют значения в единую сущность, а списки объединяют данные в виде цепочки. Такой подход позволяет обрабатывать сразу все элементы списка с помощью агрегатных операторов, которые обсуждаются чуть ниже.

Самый простой способ объявить список состоит в том, чтобы указать список значений, разделенных точками с запятой, заключенный в квадратные скобки. Позднее вы узнаете, как объявлять списки с использованием более мощного синтаксиса генераторов списков (list comprehension syntax). Пустой список, не имеющий элементов, объявляется с помощью пустых квадратных скобок `[]`:

Основные типы

```
> // Объявление списков
let vowels = [‘a’; ‘e’; ‘i’; ‘o’; ‘u’]
let emptyList = [];;

val vowels : char list = [‘a’; ‘e’; ‘i’; ‘o’; ‘u’]
val emptyList : ‘a list = []
```

В этом примере пустой список имеет тип `‘a list`, потому что он может иметь любой тип и механизм вывода типов не способен определить конкретный тип.



Основные типы

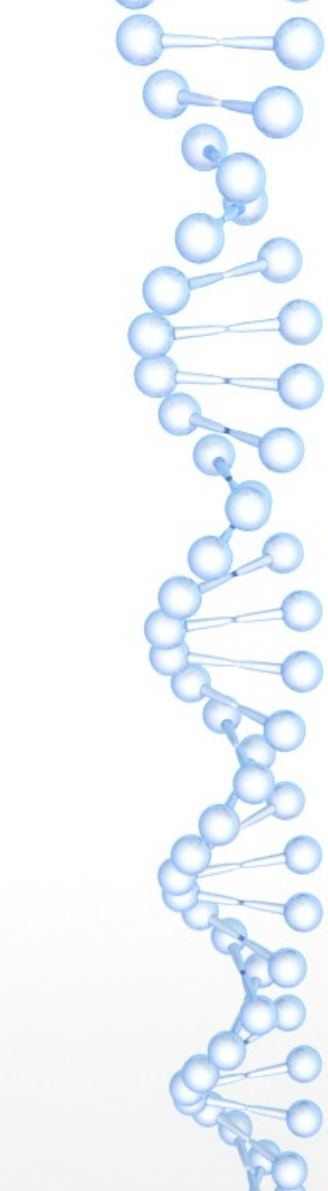
Первая элементарная операция над списками – операция добавления, которая выполняется с помощью оператора `::`. Этот оператор добавляет элемент в начало списка. В следующем примере значение `'y'` добавляется в начало списка `vowels`:

```
> // Использование оператора добавления в начало
let sometimes = 'y' :: vowels;;

val sometimes : char list = ['y'; 'a'; 'e'; 'i'; 'o'; 'u']
```

Основные типы

Вторая элементарная операция над списками – операция объединения, которая выполняется с помощью оператора @. Этот оператор объединяет два списка. В следующем примере выполняется объединение двух списков, odds и evens, в результате чего создается новый список:

```
> // Использование оператора объединения
let odds = [1; 3; 5; 7; 9]
let evens = [2; 4; 6; 8; 10]

val odds : int list = [1; 3; 5; 7; 9]
val evens : int list = [2; 4; 6; 8; 10]

> odds @ evens;;
val it : int list = [1; 3; 5; 7; 9; 2; 4; 6; 8; 10]
```

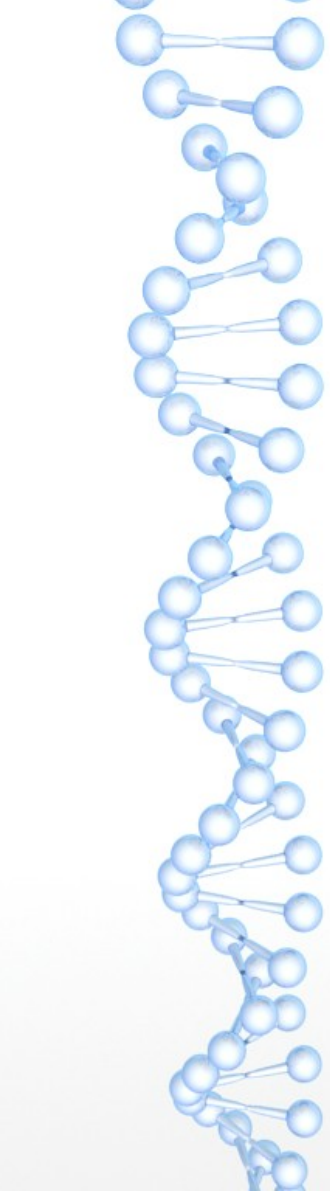


Основные типы

- Списки Черча
- Head::Tail

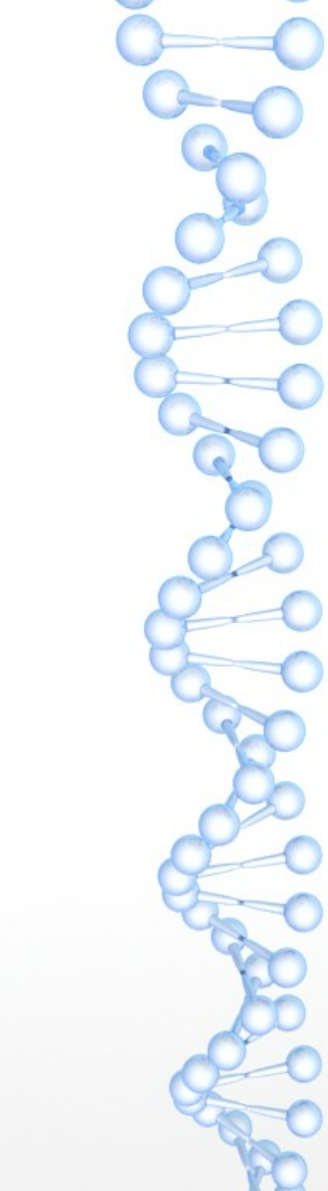
Смотреть пример https://github.com/Arseniy-Zhuck/Prolog-lections

Проект Lists



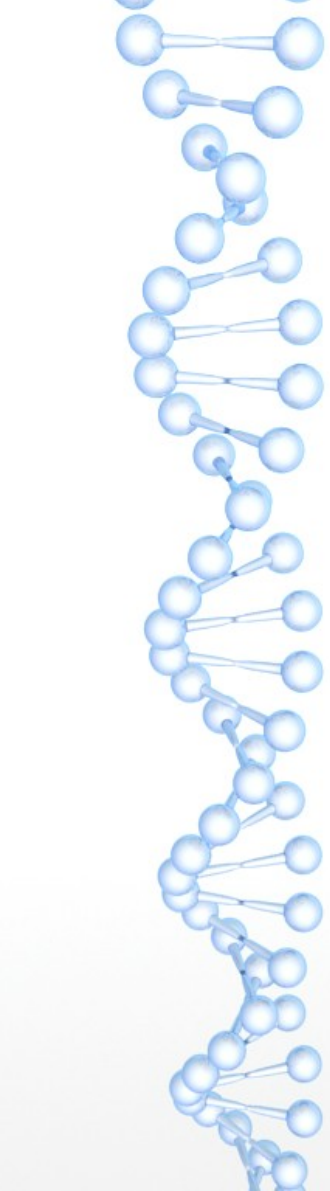
Основные типы

```
> let x = [1 .. 10];;

val x : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

> // Диапазоны списков
let tens = [0 .. 10 .. 50]
let countDown = [5L .. -1L .. 0L];;

val tens : int list = [0; 10; 20; 30; 40; 50]
val countDown : int list = [5L; 4L; 3L; 2L; 1L; 0L]
```



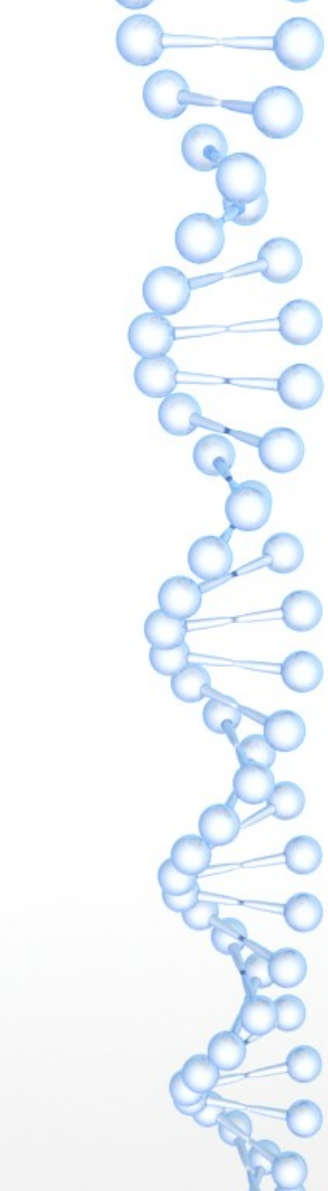
Основные типы

```
> // Простой генератор списков
let numbersNear x =
    [
        yield x - 1
        yield x
        yield x + 1
    ];;

val numbersNear : int -> int list

> numbersNear 3;;
val it : int list = [2; 3; 4]
```



Основные типы

```
> // Более сложный генератор списков
let x =
    [ let negate x = -x
      for i in 1 .. 10 do
          if i % 2 = 0 then
              yield negate i
          else
            yield i ];;

val x : int list = [1; -2; 3; -4; 5; -6; 7; -8; 9; -10]
```

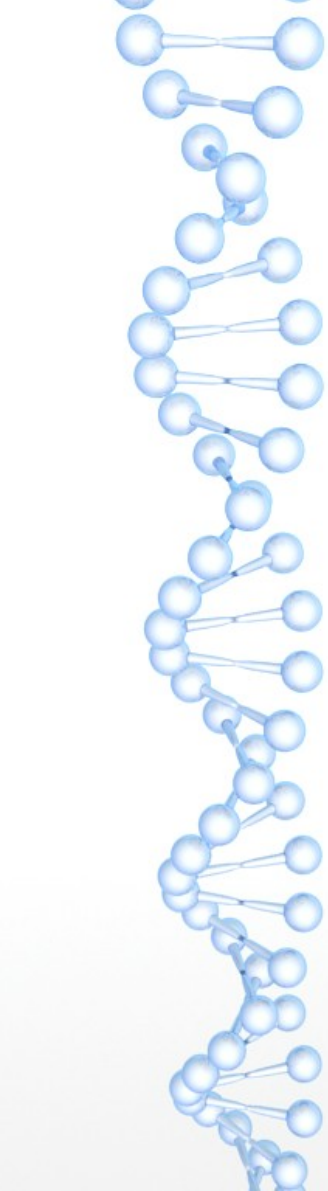


Основные типы

```
// Генерирует первые десять чисел, кратных заданному числу
let multiplesOf x = [ for i in 1 .. 10 do yield x * i ]

// Упрощенный генератор списков
let multiplesOf2 x = [ for i in 1 .. 10 -> x * i ]
```

Основные типы

```
> // Использование генератора списков для нахождения простых чисел
let primesUnder max =
    [
        for n in 1 .. max do
            let factorsOfN =
                [
                    for i in 1 .. n do
                        if n % i = 0 then
                            yield i
                ]

            // n - простое число, если для него имеется
            // всего два делителя, 1 и n
            if List.length factorsOfN = 2 then
                yield n
    ];;

val primesUnder : int -> int list

> primesUnder 50;;
val it : int list = [2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47]
```



Основные типы

Функция и тип	Описание
`List.length` `'a list -> int`	Возвращает длину списка.
`List.head` `'a list -> 'a`	Возвращает первый элемент списка.
`List.tail` `'a list -> 'a list`	Возвращает часть списка без первого элемента.
`List.exists` `('a -> bool) -> 'a list -> bool`	Возвращает признак того, удовлетворяет ли какой-либо элемент списка функции поиска или нет.
`List.rev` `'a list -> 'a list`	Изменяет порядок следования элементов в списке на противоположный.
`List.tryfind` `('a -> bool) -> 'a list -> 'a option`	Возвращает `Some(x)`, где `x` – первый элемент списка, для которого указанная функция вернула значение `true`. В противном случае возвращает `None`. (Значения `Some` и `None` мы рассмотрим чуть ниже.)
`List.zip` `'a list -> 'b list -> ('a * 'b) list`	Принимает два списка одинаковой длины. Возвращает объединенный список кортежей.
`List.filter` `('a -> bool) -> 'a list -> 'a list`	Возвращает список, содержащий только те элементы, для которых указанная функция вернула значение `true`.

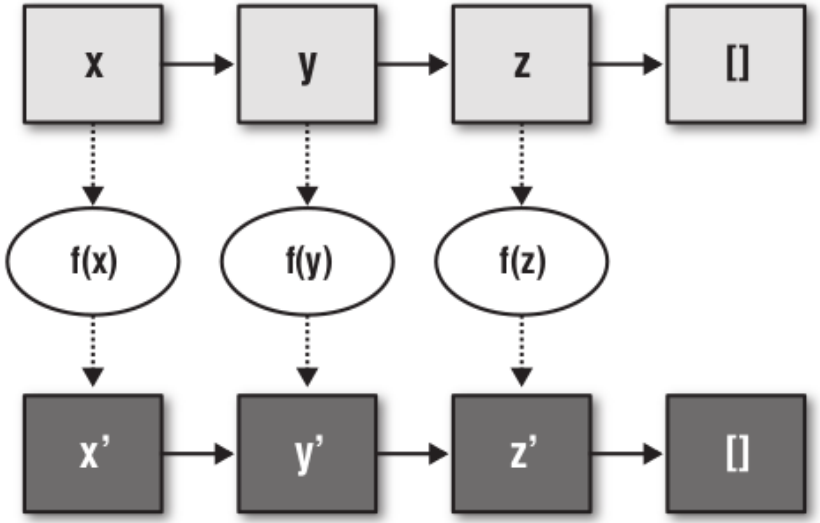
`List.partition`
`('a -> bool) -> 'a list ->`
`('a list * 'a list)`
	Принимает функцию-предикат и список, возвращает два новых списка. Первый список содержит элементы, для которых указанная функция вернула значение `true`, а второй – элементы, для которых эта функция вернула значение `false`.

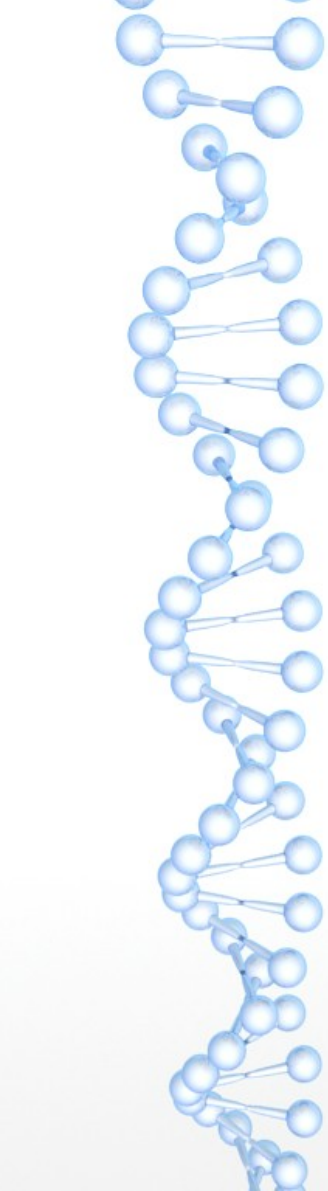
Агрегатные методы

Функция `List.map` – это операция проекции, которая создает новый список на основе заданной функции. Каждый элемент нового списка является результатом выполнения функции. Функция `List.map` имеет тип:

```
('a -> 'b) -> 'a list -> 'b list
```

Вы можете наглядно представить себе функцию отображения `f` на списке `[x; y; z]`, как показано на рис. 2.1.



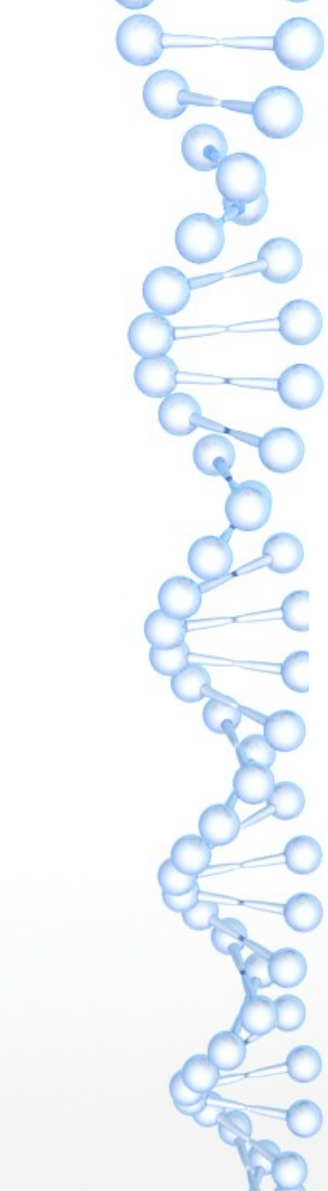


Агрегатные методы

```
> let squares x = x * x;;

val squares : int -> int

> List.map squares [1 .. 10];;
val it : int list = [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
```

Агрегатные методы

```
> // Подсчет количества гласных букв в строке
let countVowels (str : string) =
    let charList = List.ofSeq str

    let accFunc (As, Es, Is, Os, Us) letter =
        if letter = 'a' then (As + 1, Es, Is, Os, Us)
        elif letter = 'e' then (As, Es + 1, Is, Os, Us)
        elif letter = 'i' then (As, Es, Is + 1, Os, Us)
        elif letter = 'o' then (As, Es, Is, Os + 1, Us)
        elif letter = 'u' then (As, Es, Is, Os, Us + 1)
        else (As, Es, Is, Os, Us)

    List.fold accFunc (0, 0, 0, 0, 0) charList;;

val countVowels : string -> int * int * int * int * int

> countVowels "The quick brown fox jumps over the lazy dog";;
val it : int * int * int * int * int = (1, 3, 1, 4, 2)
```



Агрегатные методы

Функции `List.reduce` и `List.fold` обрабатывают списки в направлении слева направо. При этом существуют альтернативные им функции `List.reduceBack` и `List.foldBack`, обрабатывающие списки справа налево. В некоторых ситуациях обработка списков в обратном порядке может оказывать существенное влияние на производительность. (Более подробно проблема производительности при обработке списков рассматривается в главе 7.)

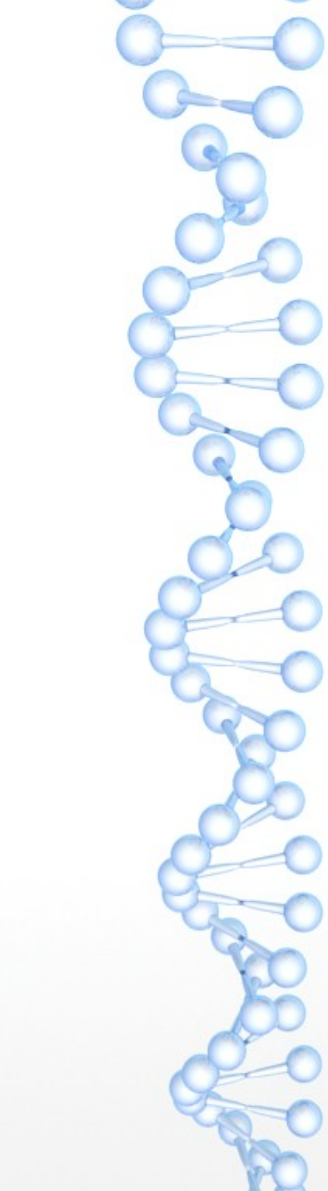
Агрегатные методы

List.iter

Последний агрегатный оператор, `List.iter`, проходит по всем элементам списка и вызывает функцию, переданную вами в виде параметра. Она имеет тип:

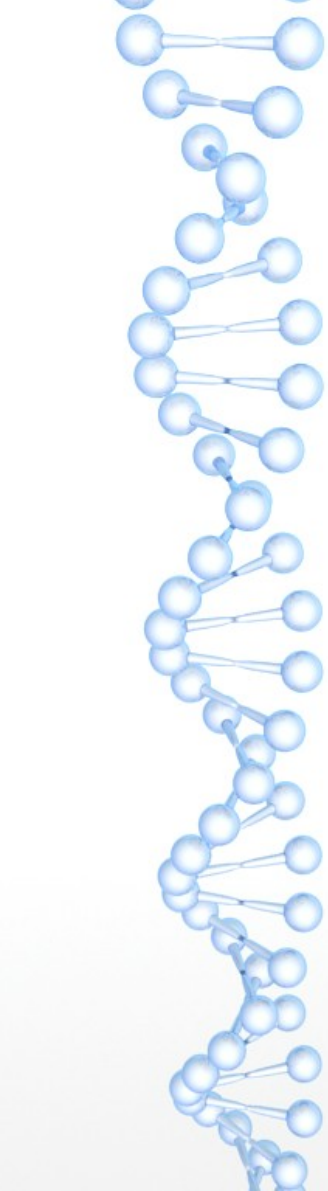
```
('a -> unit) -> 'a list -> unit
```

Функция `List.iter` возвращает значение `unit`, поэтому она прежде всего используется для получения побочного эффекта вызова указанной функции. Под термином «побочный эффект» подразумевается, что вызов функции приводит к некоторому побочному эффекту помимо возвращаемого значения. Например, побочным эффектом вызова функции `printfn` является вывод текста на консоль помимо возвращаемого значение типа `unit`.



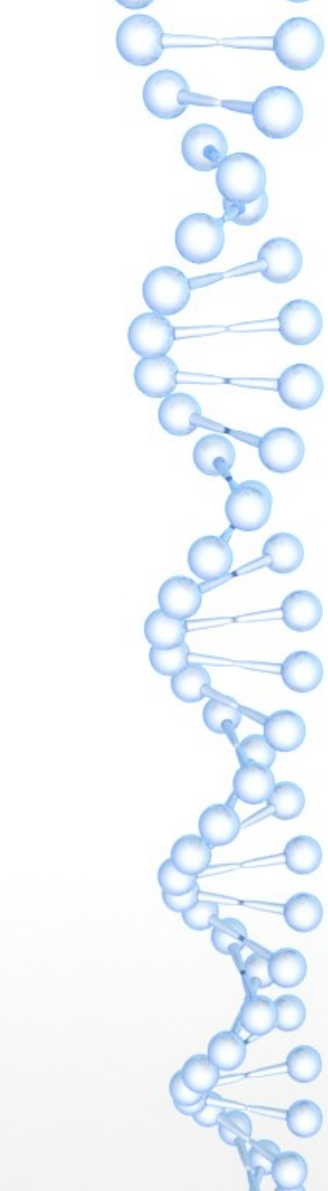
Агрегатные методы

```
> // Использование функции List.iter
let printNumber x = printfn "Printing %d" x
List.iter printNumber [1 .. 5];;

Printing 1
Printing 2
Printing 3
Printing 4
Printing 5

val printNumber : int -> unit
```



Тип option

```
> // Использование типа option для возвращаемого значения
open System

let isInteger str =
    let successful, result = Int32.TryParse(str)
    if successful
    then Some(result)
    else None;;

val isInteger : string -> int option

> isInteger "This is not an int";;
val it : int option = None
> isInteger "400";;
val it : int option = Some 400
```

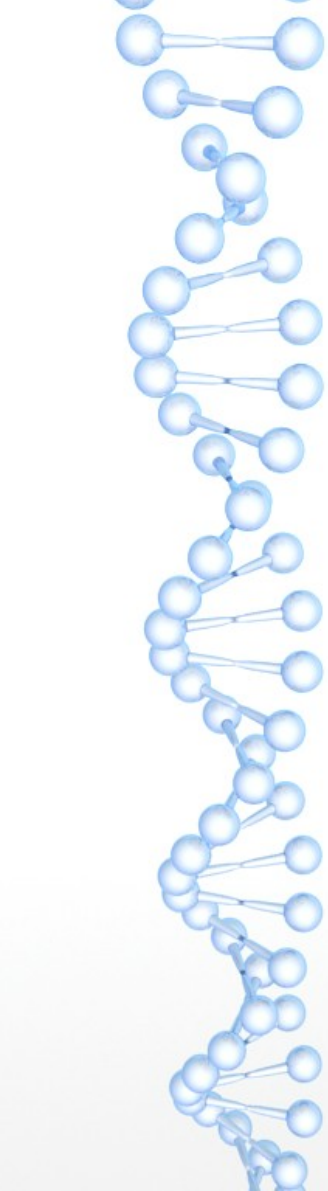


Тип option

```
> // Использование функции Option.get
let isLessThanZero x = (x < 0)

let containsNegativeNumbers intList =
    let filteredList = List.filter isLessThanZero intList
    if List.length filteredList > 0
    then Some(filteredList)
    else None;;

val containsNegativeNumbers : int list -> int list option

> let negativeNumbers = containsNegativeNumbers [6; 20; -8; 45; -5];;

val negativeNumbers : int list option = Some [-8; -5]

> Option.get negativeNumbers;;
val it : int list = [-8; -5]
```

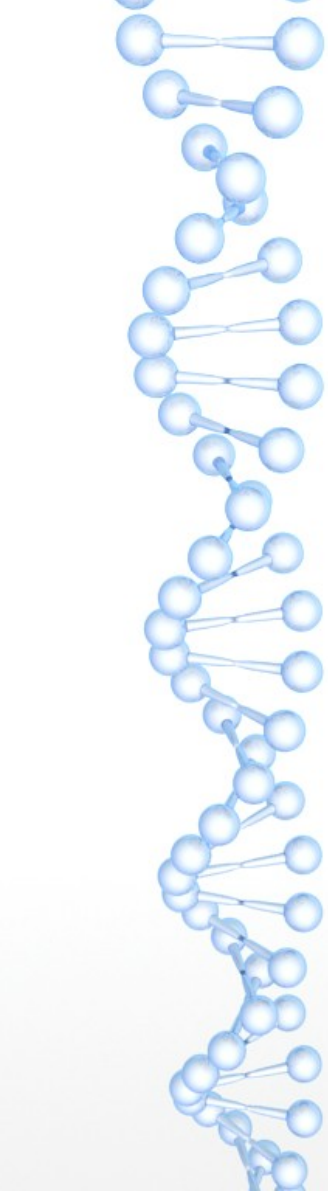
Тип option

Функция и ее тип	Описание
`Option.isSome` `'a option -> bool`	Для значения `Some` возвращает `true`, в противном случае возвращает `false`.
`Option.isNone` `'a option -> bool`	Для значения `Some` возвращает `false`, в противном случае возвращает `true`.



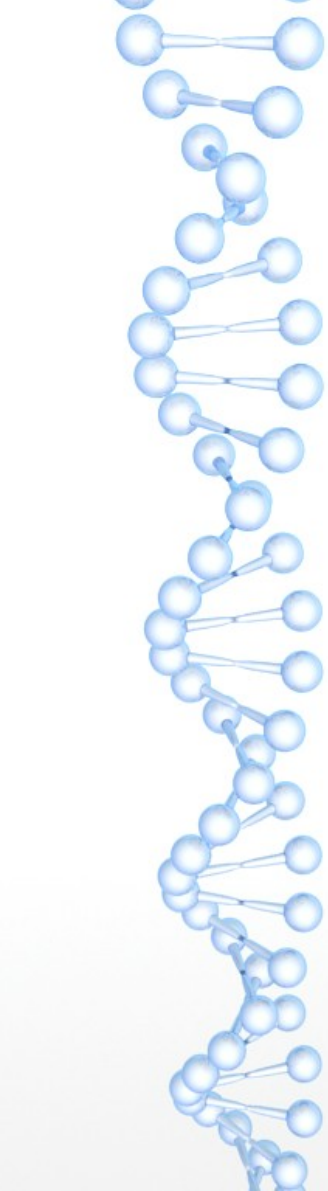
Форматирование строки

Спецификатор	Описание	Пример	Результат
%d, %i	Выводит целое значение	printf "%d" 5	5
%x, %o	Выводит любое целое значение в шестнадцатеричном и восьмеричном форматах соответственно	printfn "%x" 255	ff
%s	Выводит любую строку	printf "%s" "ABC"	ABC
%f	Выводит любое вещественное число	printf "%f" 1.1M	1.100000
%c	Выводит любой символ	printf "%c" '\097'	a
%b	Выводит любое булево значение	printf "%b" false	false
%0	Выводит любой объект	printfn "%0" (1,2)	(1,2)
%A	Выводит значение любого типа	printf "%A" (1, [])	(1, [])



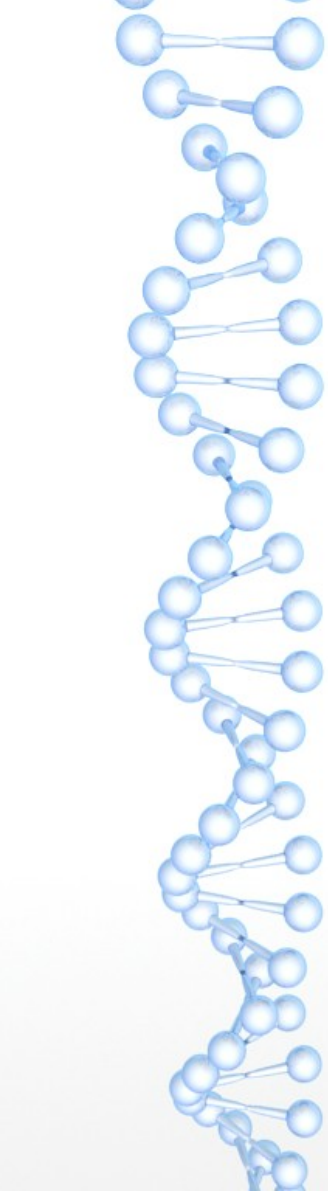
Модули

```
module Utilities

module ConversionUtils =

    // Utilities.ConversionUtils.intToString
    let intToString (x : int) = x.ToString()

    module ConvertBase =
        // Utilities.ConversionUtils.ConvertBase.convertToHex
        let convertToHex x = sprintf "%x" x

        // Utilities.ConversionUtils.ConvertBase.convertToOct
        let convertToOct x = sprintf "%o" x

module DataTypes =
    // Utilities.DataTypes.Point
    type Point = Point of float * float * float
```



Пространства имен

```
namespace PlayingCards

// PlayingCards.Suit
type Suit =
  | Spade
  | Club
  | Diamond
  | Heart

// PlayingCards.PlayingCard
type PlayingCard =
  | Ace       of Suit
  | King      of Suit
  | Queen     of Suit
  | Jack      of Suit
  | ValueCard of int * Suit
```



Пространства имен

```
// Program.fs
let numbers = [1 .. 10]
let square x = x * x

let squaredNumbers = List.map square numbers

printfn "SquaredNumbers = %A" squaredNumbers

open System

printfn "(press any key to continue)"
Console.ReadKey(true)
```

Запуск программ

```
// Program.fs
open System

[<EntryPoint>]
let main (args : string[]) =
    let numbers = [1 .. 10]
    let square x = x * x

    let squaredNumbers = List.map square numbers

    printfn "SquaredNumbers = %A" squaredNumbers

    printfn "(press any key to continue)"
    Console.ReadKey(True) |> ignore

    // Вернуть 0
    0
```