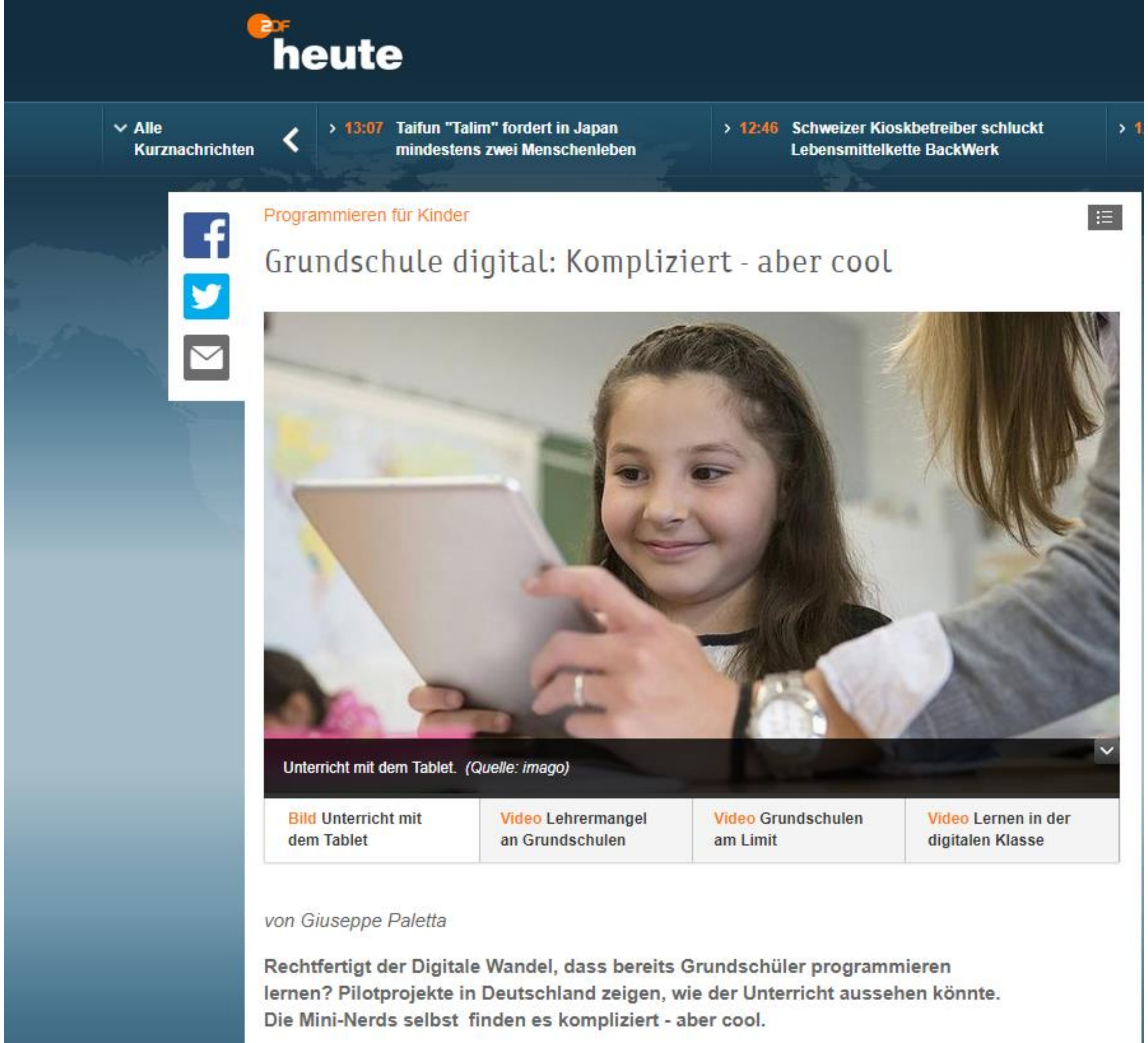
TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Scientific functional programming

01 – Programming using F#

Timo Mühlhaus
Computational systems biology



Programmieren für Kinder

Grundschule digital: Kompliziert - aber cool

Unterricht mit dem Tablet. *(Quelle: imago)*

Bild Unterricht mit dem Tablet	Video Lehrermangel an Grundschulen	Video Grundschulen am Limit	Video Lernen in der digitalen Klasse

von Giuseppe Paletta

Rechtfertigt der Digitale Wandel, dass bereits Grundschüler programmieren lernen? Pilotprojekte in Deutschland zeigen, wie der Unterricht aussehen könnte. Die Mini-Nerds selbst finden es kompliziert - aber cool.

Scientific computing

- Understanding the computational problem-solving process:

Problem

identify the problem to be solved

Model

use simplification and abstraction to arrive at a mathematical model for the problem

Method

Identify a computational method (algorithm) for solving the model

Implementation

Implement the method in the appropriate programming language

Scientific programming

Method

Identify a computational method (algorithm) for solving the model

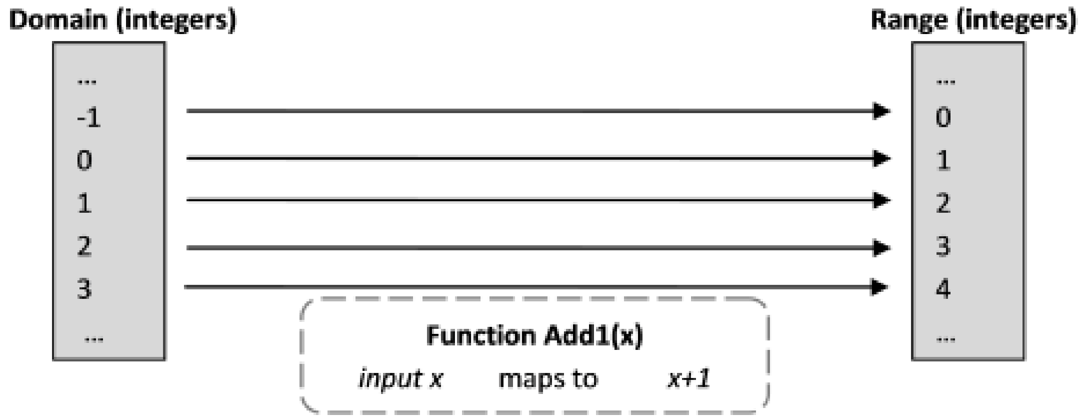
Implementation

Implement the method in the appropriate programming language

Paradigm: functional programming

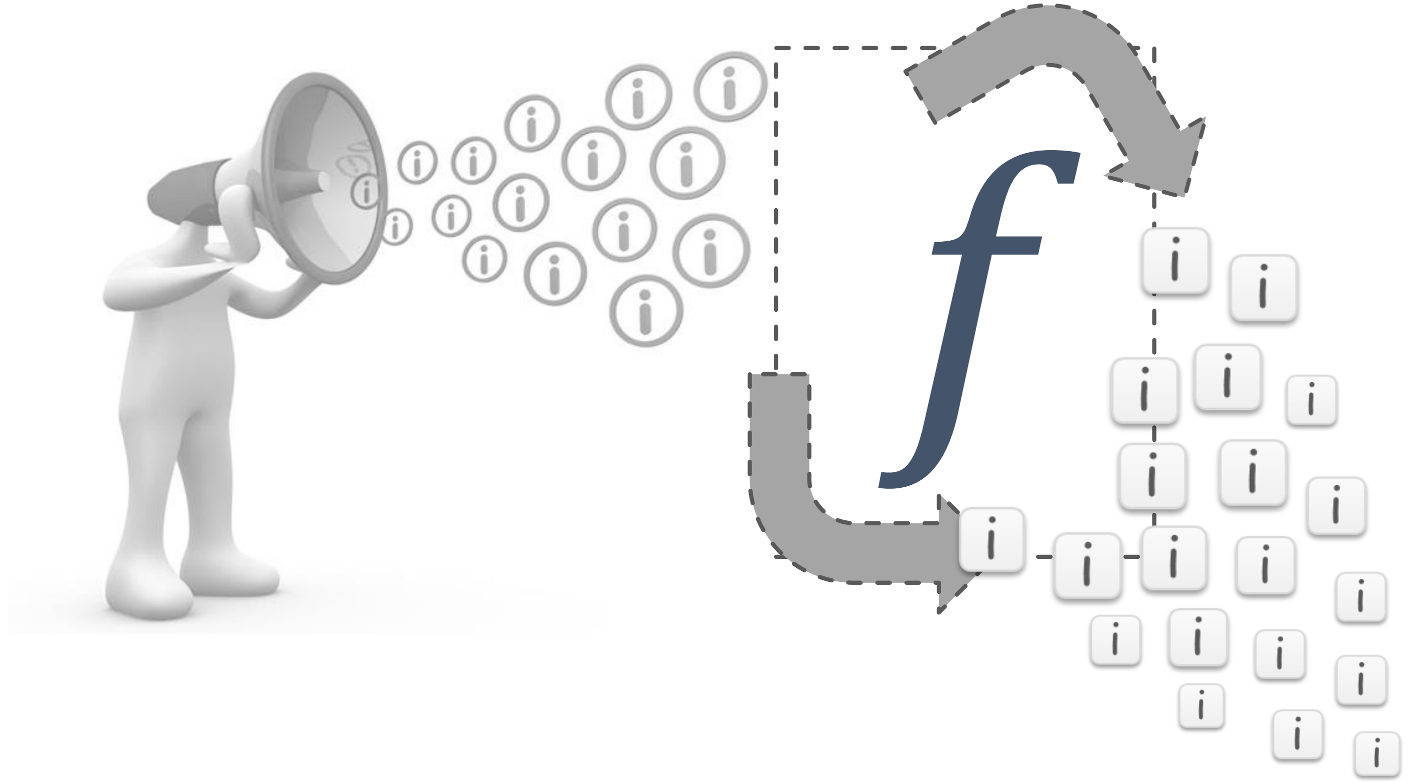
- Definition: Functional programming is programming with mathematical function, where function is a first class citizen of a program

A function as instruction



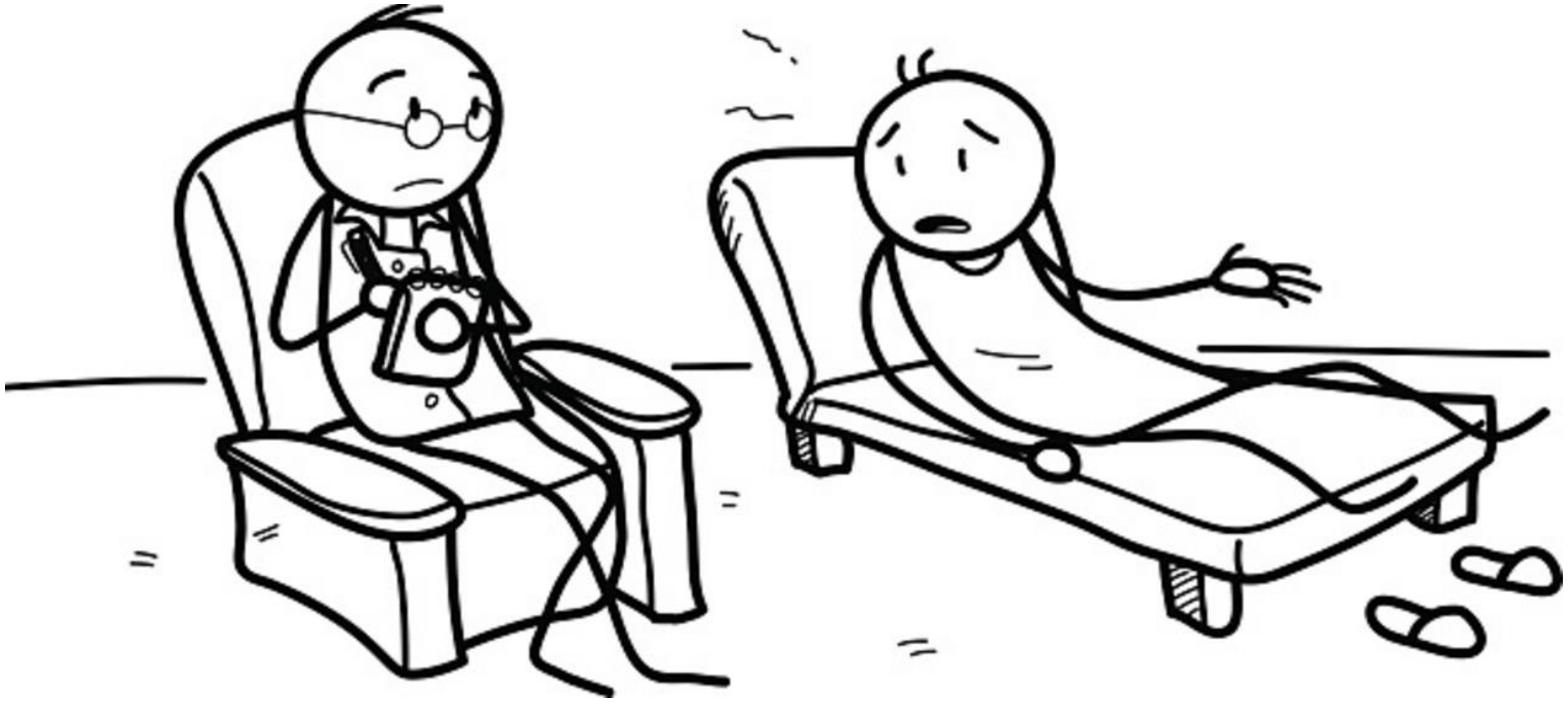
- Domain: The set of values that can be used as input to the function
- Range: The set of possible output values from the function
- **The function is said to map the domain to the range**

Paradigm: functional abstraction



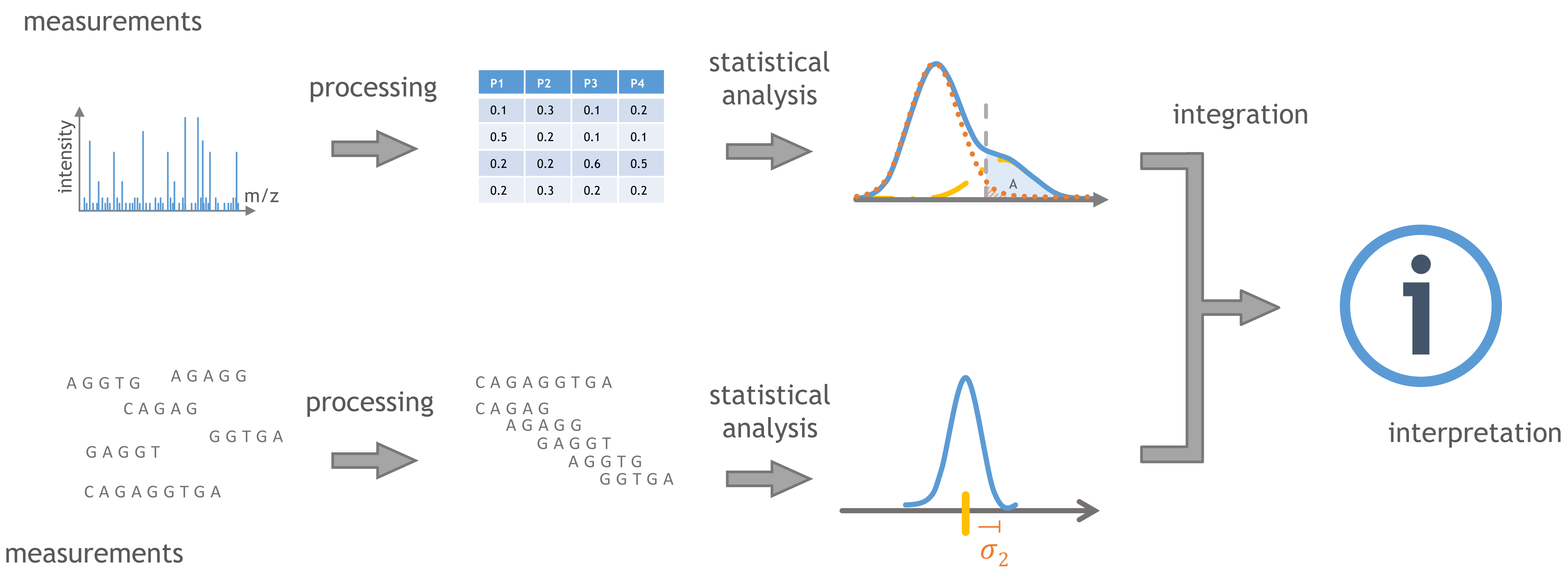
- Within the real world we are surrounded by information
- Functions declare the instructions of transforming information (declarative style)
- Information remains unchanged (immutability)

Why functional…



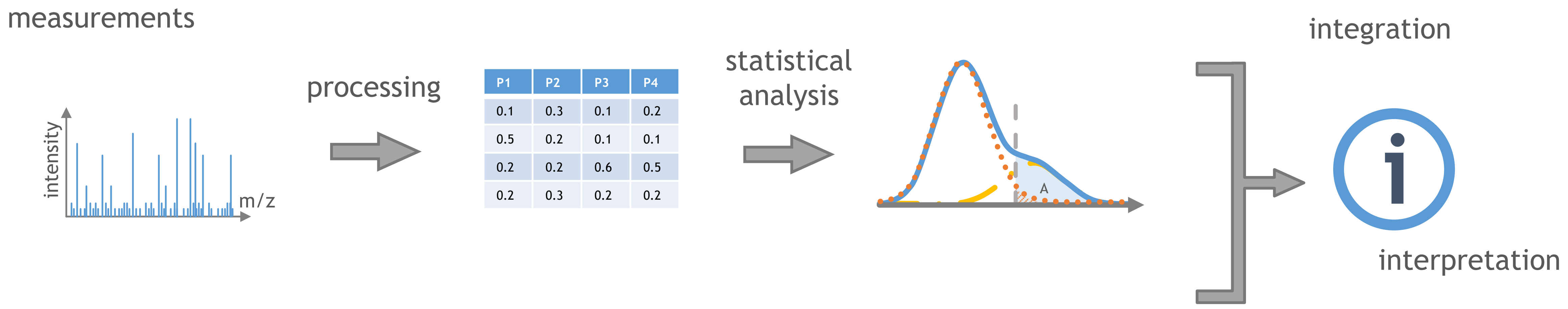
…when all my friends use python?

Take a look at a scientific programming workflow

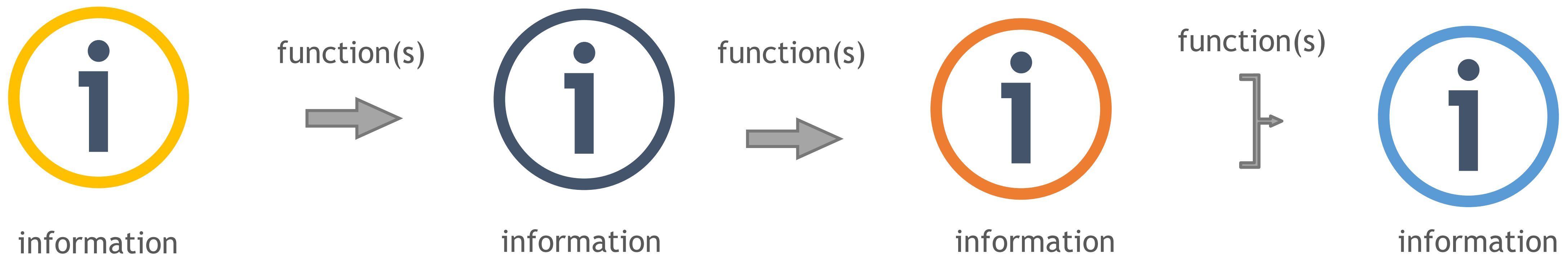


P1	P2	P3	P4
0.1	0.3	0.1	0.2
0.5	0.2	0.1	0.1
0.2	0.2	0.6	0.5
0.2	0.3	0.2	0.2

Take a look at a scientific programming workflow

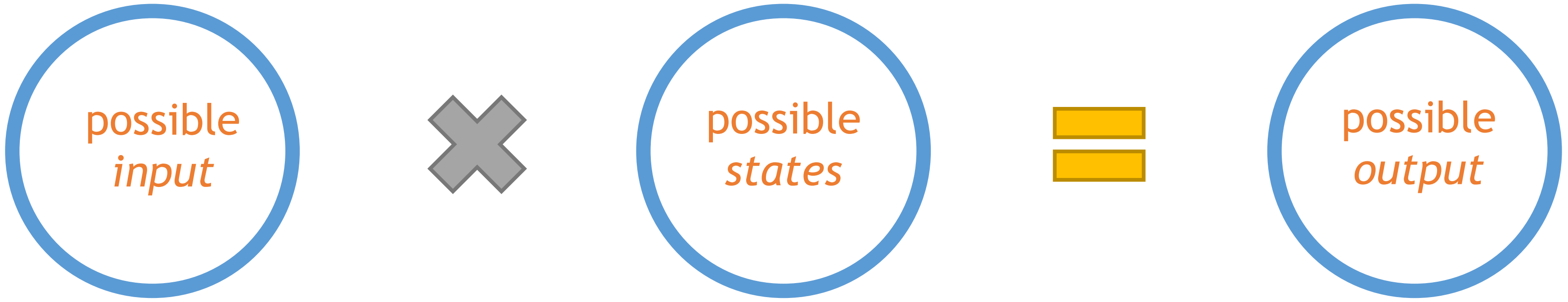


P1	P2	P3	P4
0.1	0.3	0.1	0.2
0.5	0.2	0.1	0.1
0.2	0.2	0.6	0.5
0.2	0.3	0.2	0.2



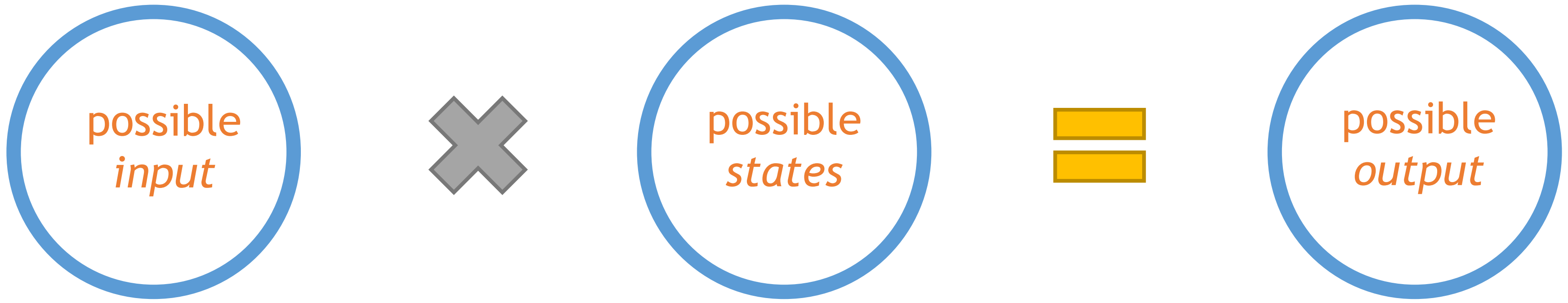
“Controlling complexity is the essence of computer programming.” -Brian Kernighan

- Complexity comes from *input* and *state*



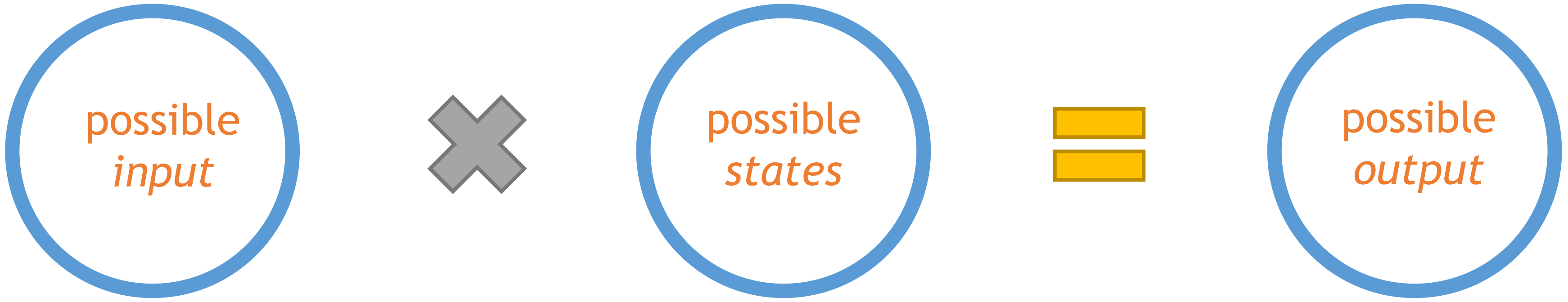
“Controlling complexity is the essence of computer programming.” -Brian Kernighan

- Functional programming imposes constraints that eliminate *states*
- *Immutability eliminates states as you can safely share the reference*

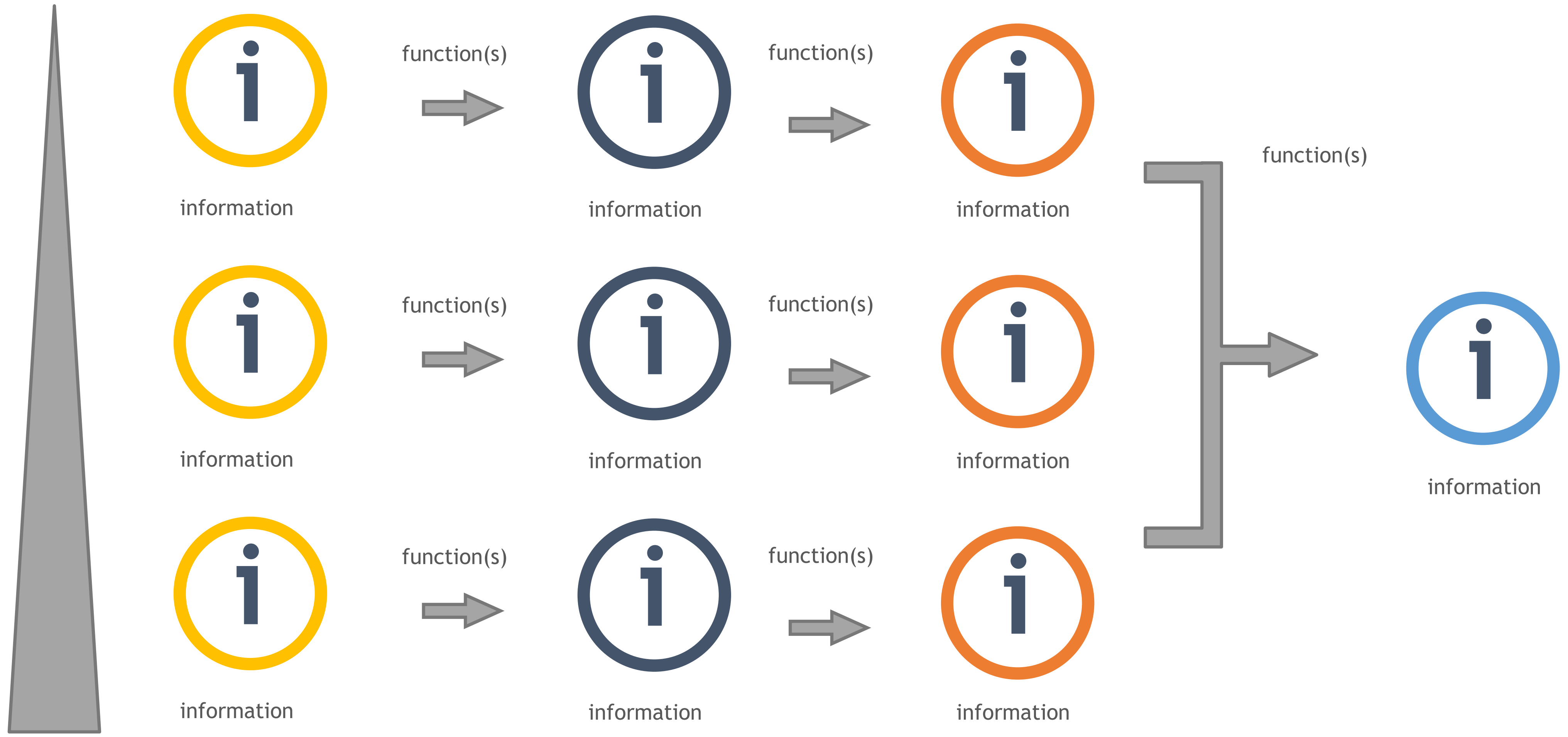


"Controlling complexity is the essence of computer programming." –Brian Kernighan

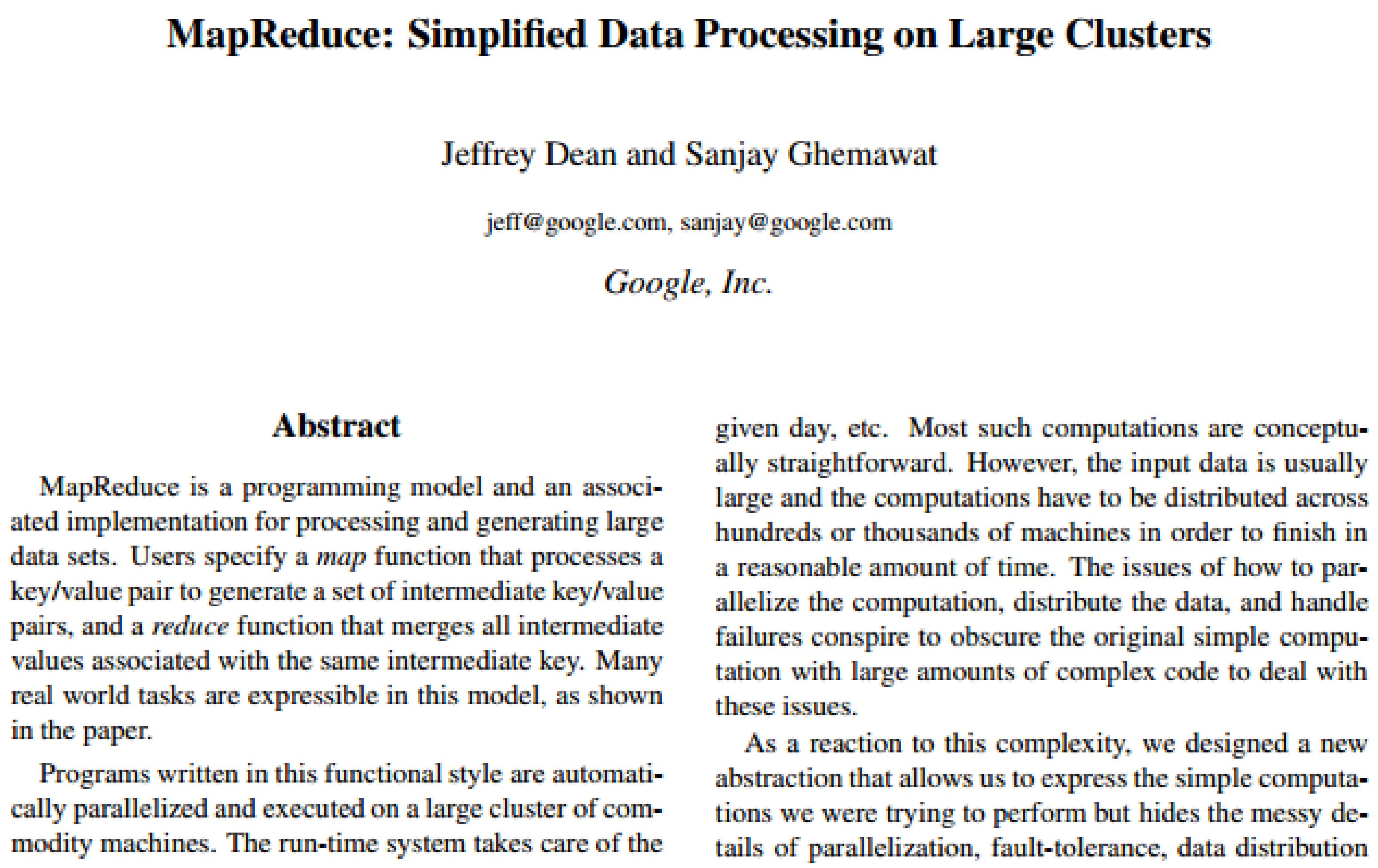
- Functional programming imposes constraints that eliminate *possible inputs*
- *A function takes arguments and produces a result (models the input)*



Scalability



Googles mapReduce



MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

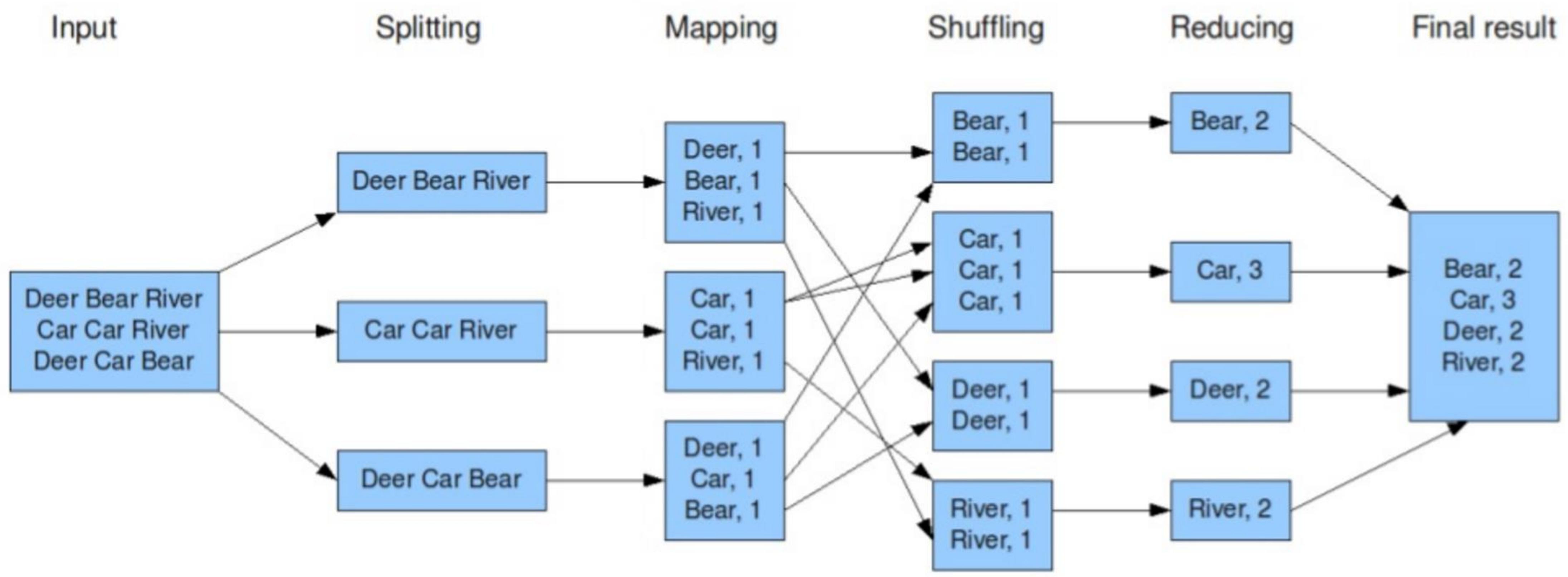
Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

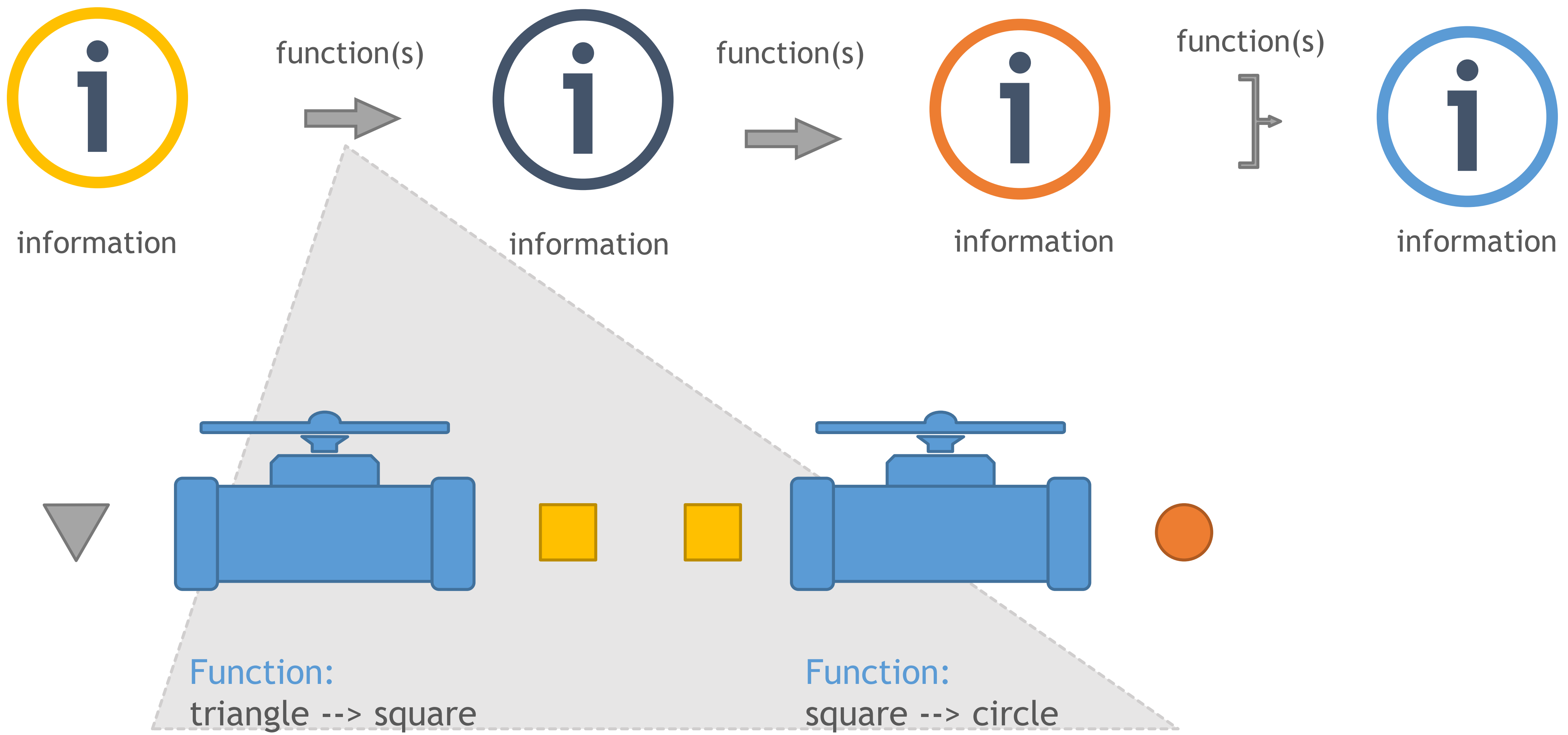
As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution

- 2004

Googles mapReduce



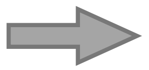
Modularity

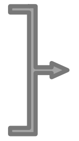

information
function(s)
information
function(s)
information
function(s)
information
Function:
triangle --> square
Function:
square --> circle

Modularity

- Modules should be simple parts that can combine in many ways to give interesting results
- A module encapsulates one functionality
- "Functional programming is good because it leads to modules that can be combined freely"
  (John Hughes, 1985 Why functional programming matters)

- Leads to map and fold concept





Imperative vs. declarative

- Imperative programming tells the machine how to do something (resulting in what you want to happen)
- Declarative programming tells the machine what you would like to happen (and the computer figures out how to do it)

Example: Doubling all numbers in an array

```
// imperative
let numbers = [|1;2;3;4;5|]
let doubled = [|0;0;0;0;0|]

for i = 0 to numbers.length-1 do
  let newNumber = numbers[i] * 2
  doubled.[i] <- newNumber
```

```
// declarative
let numbers = [|1;2;3;4;5|]

numbers
|> Array.map (fun x -> x * 2)
```

Imperative vs. declarative

Correctness

- Mathematical reasoning
  - Types define classifications of values
  - With types, the compiler can verify some aspects of correctness

$$\lambda x. x + 2 \leftarrow$$

Lambda calculus:
minimal notation to define functions

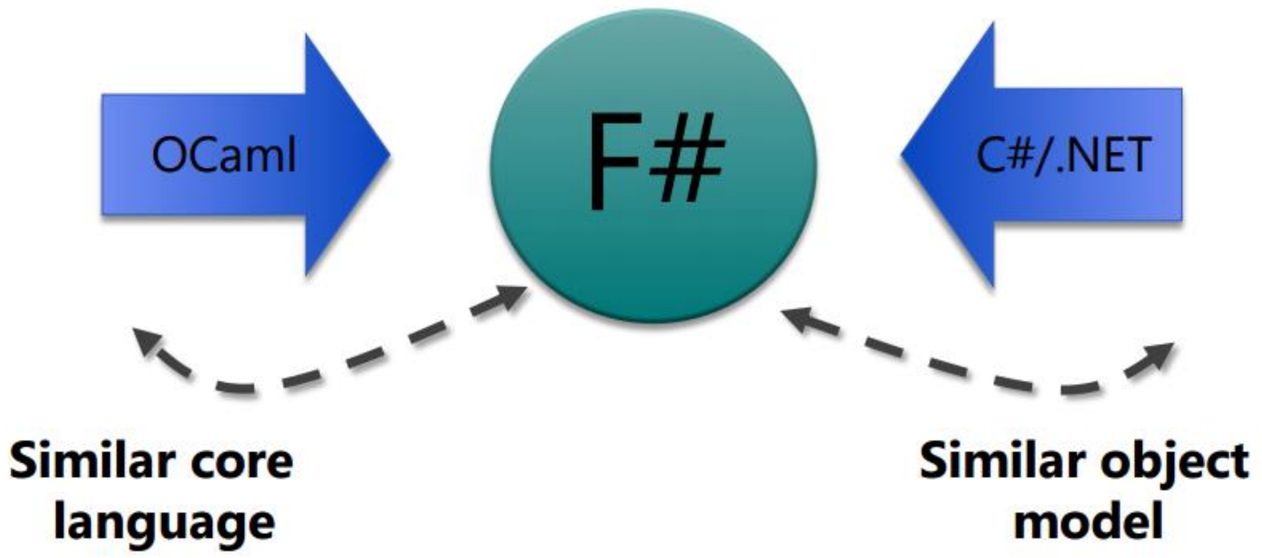
$$f(x) = x + 2$$

```
let f x = x + 2
```

F# (Fsharp)

Two flavours of functional programming

- Pure
  - No side-effects at all: values are immutable, no state Mathematics has got on just fine with pure functions for centuries!
- Impure
  - Some values are mutable, functions can have side-effects



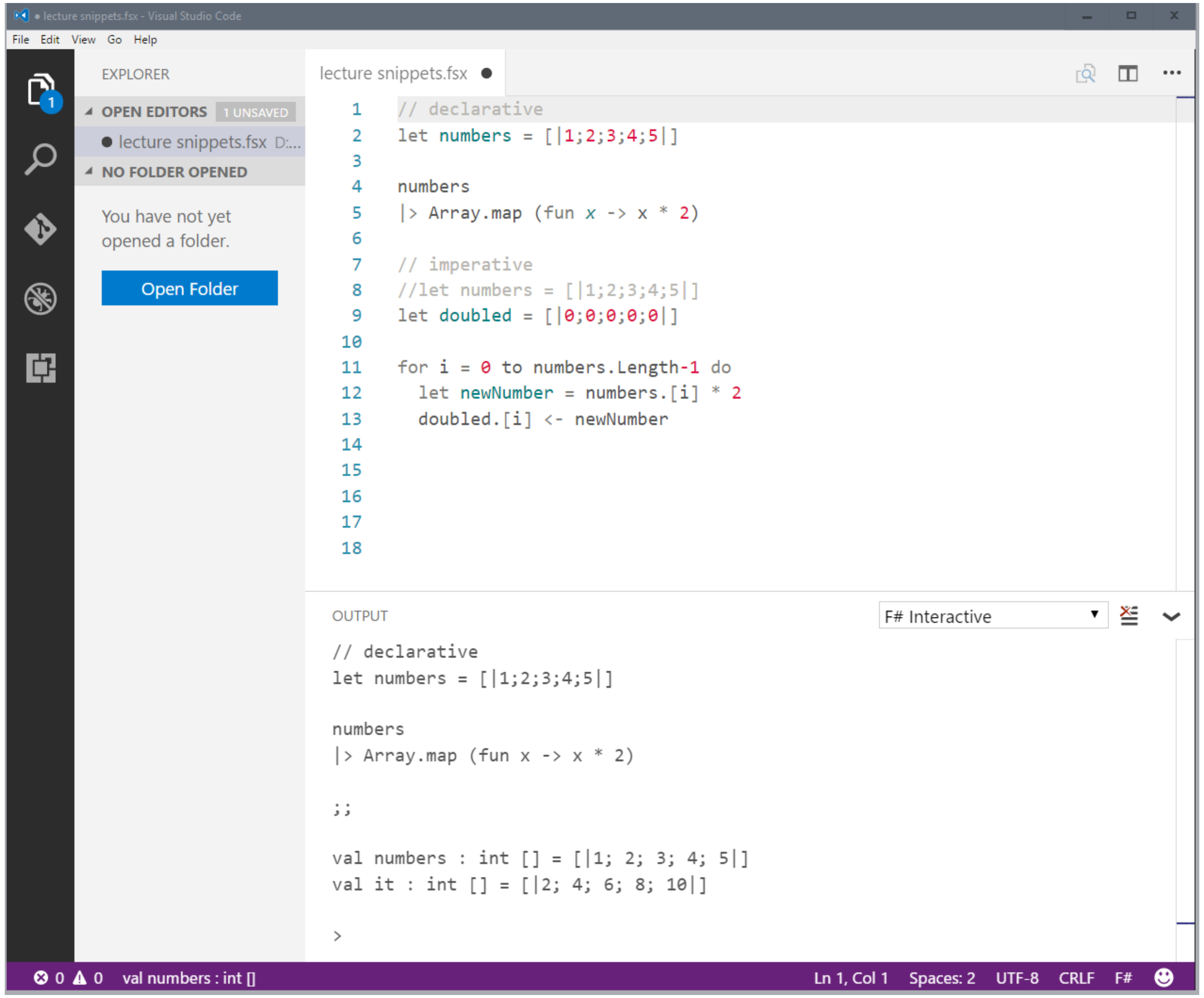
The functional mindset

- You can program imperative and object oriented in F# …
- … but you should always have a functional mindset.

Getting started

- If you're using Visual Studio, you've already got F# installed, so you're ready to go!
- There are two ways to use F# interactively:
  (1) typing in the F# interactive window directly, or
  (2) creating a F# script file (.FSX) and then evaluating code snippets.

Visual Studio Code


● lecture snippets.fsx - Visual Studio Code
File Edit View Go Help
EXPLORER
OPEN EDITORS 1 UNSAVED
● lecture snippets.fsx D:...
NO FOLDER OPENED
You have not yet opened a folder.
Open Folder
lecture snippets.fsx ●
OUTPUT
F# Interactive
⊗ 0 ⚠ 0 val numbers : int []
Ln 1, Col 1 Spaces: 2 UTF-8 CRLF F#

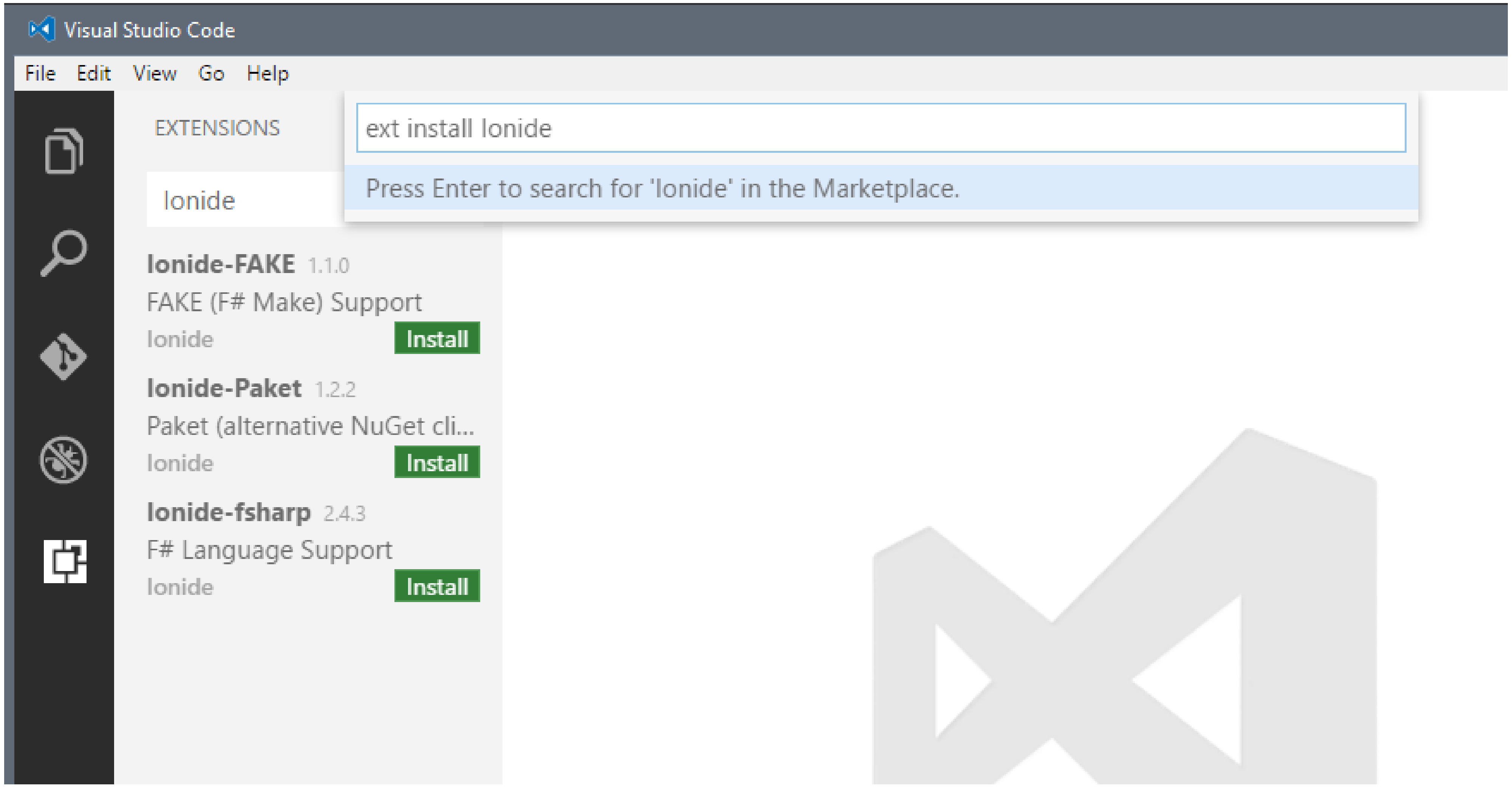
```
// declarative
let numbers = [|1;2;3;4;5|]

numbers
|> Array.map (fun x -> x * 2)

// imperative
//let numbers = [|1;2;3;4;5|]
let doubled = [|0;0;0;0;0|]

for i = 0 to numbers.Length-1 do
  let newNumber = numbers.[i] * 2
  doubled.[i] <- newNumber
```

```
// declarative
let numbers = [|1;2;3;4;5|]

numbers
|> Array.map (fun x -> x * 2)

;;

val numbers : int [] = [|1; 2; 3; 4; 5|]
val it : int [] = [|2; 4; 6; 8; 10|]

>
```

Installation procedure

- Visual Studio Code
  https://code.visualstudio.com/
- Visual F# Tools 4.0 RTM
  https://www.microsoft.com/en-us/download/details.aspx?id=48179
- Microsoft Build Tools 2015
  https://www.microsoft.com/en-us/download/details.aspx?id=48159&wa=wsignin1.0

Install Ionide F# support



```
ext install Ionide
```

Press Enter to search for 'Ionide' in the Marketplace.

Ionide

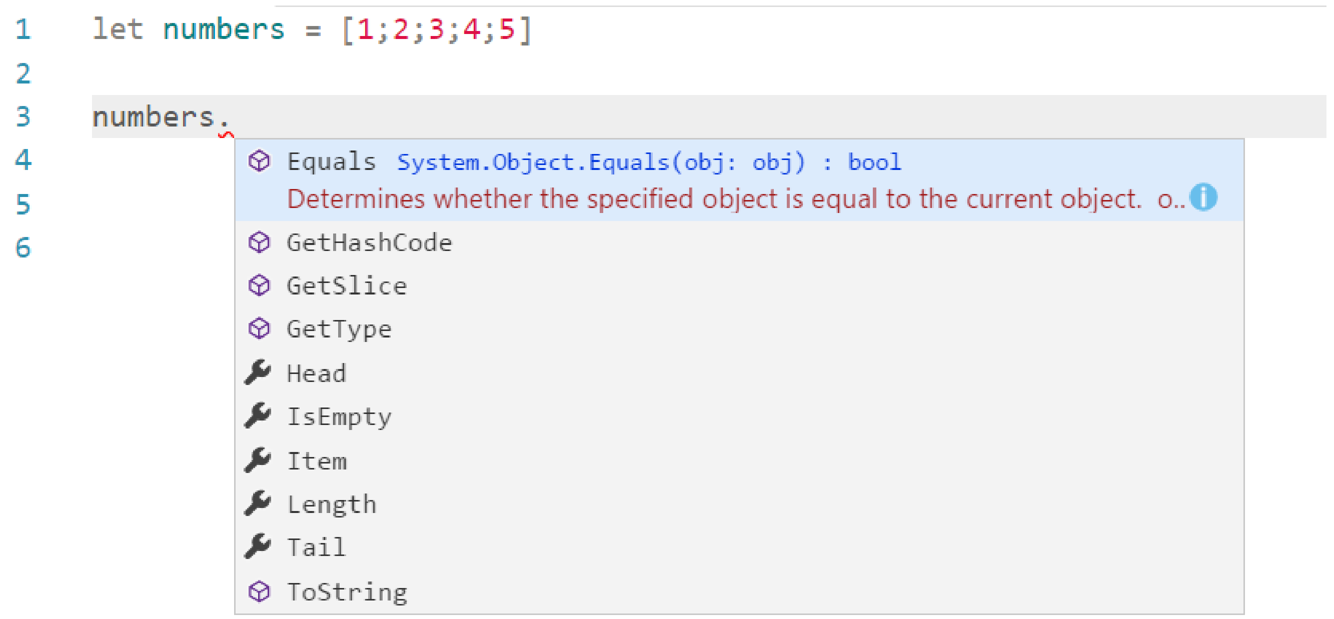
Ionide-FAKE 1.1.0
FAKE (F# Make) Support
Ionide Install

Ionide-Paket 1.2.2
Paket (alternative NuGet cli...
Ionide Install

Ionide-fsharp 2.4.3
F# Language Support
Ionide Install

IntelliSense and syntax highlighting

- Syntax highlighting
- IntelliSense provide auto-completion with smart suggestions based on variable types, function definitions, and imported modules.



```
let numbers = [1;2;3;4;5]

numbers.
```

```
Equals  System.Object.Equals(obj: obj) : bool
Determines whether the specified object is equal to the current object.  o..
GetHashCode
GetSlice
GetType
Head
IsEmpty
Item
Length
Tail
ToString
```

Binding and values

- Value binding
    - Always begin with keyword “let”:

```
let x = 0
let piConstant = 3.141
let name = “Eriawan”
```

F#

Mutability in F#

- Immutable by default:

```fsharp
let y = x + 1
```
F#

- Mutable is explicit:
  - Adding keyword “mutable”
  - Operation of assignment must use “<-”

```fsharp
let mutable y = 10
y <- y + 1
```
F#

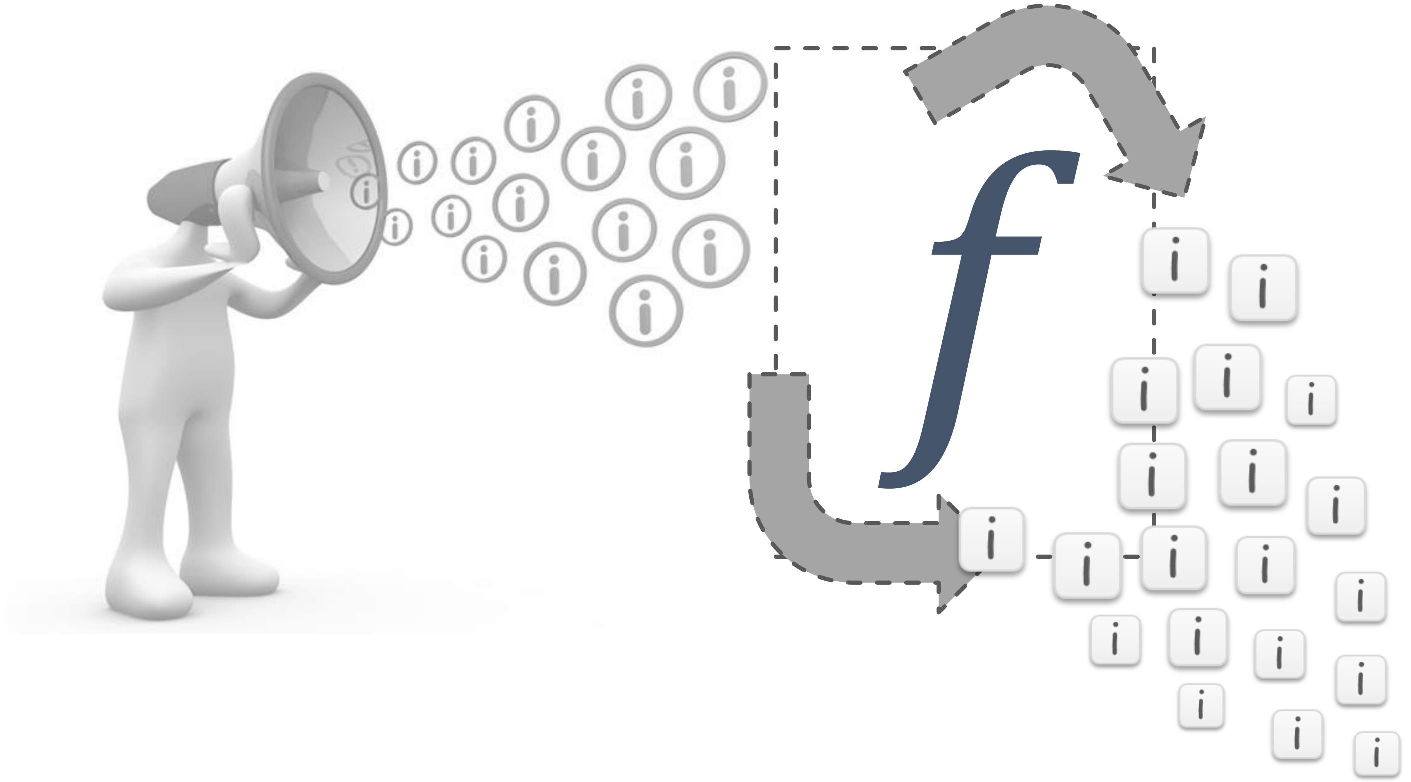
Succinct syntax of F#

- Commenting code
  - Comments are marked by “//” or “///”:

```
// some code
let x = x + 1

/// A square function
let sqr y = y * y
```

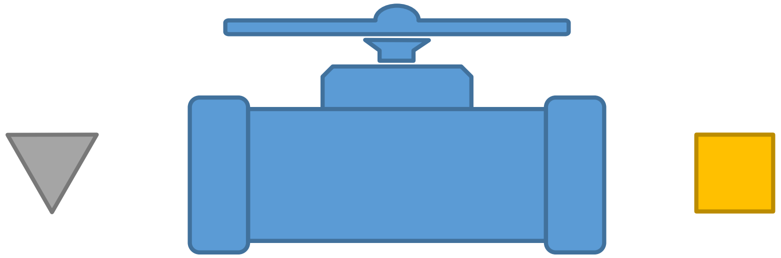
F#

Data types



- Data types specify the ‘shape’ of the information

Data types



Function:
triangle --> square

- Data types specify the ‘shape’ of the information

Primitive data types

- Data stored in memory is a sequence of bits (0 or 1)

1000010

- 66?
- 'B'?
- 9.2E-44?

- How the computer interprets the sequence of bits depends on the context?
- In a statically typed language, we must make the context explicit by specifying the type of the data

What is a Data types

- A data type is a domain of similar characteristics
- It defines the type of information stored in the computer memory
- Examples:
    - Positive integers: 1,2,3, …
    - Alphabetical characters: a, b, c, …
    - Days of week: Monday, Tuesday, …

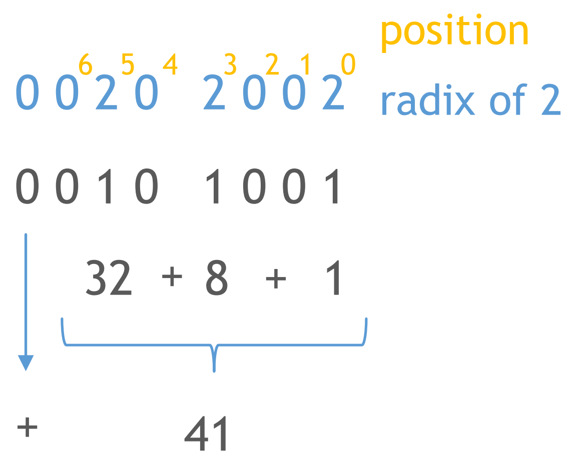
Data type characteristics

- A data type has:
  - Name
  - Size (how much memory is used)
  - Default value
- Example integer number:
  - Name int
  - Size 32 bits (4 bytes)
  - Default value 0

Integer types

- Represent whole number
- signed or unsigned
- Range is depending on the size of used memory
- Default value 0

```
// Bind an integer type
let i = 41
```



Binary number:
positional notation with a radix of 2

Floating-Point types

- Represent real numbers
- Signed or unsigned
- Range is depending on the size of used memory
  - 32-bits -> precision of 7 digits
  - 64-bits -> precision of 15-16 digits
- Default value 0.0
- ! Can behave abnormally in calculations

```
// Bind an float type
let pi = 3.14159265359
```

Floating-Point abnormalities

- Comparing floating-point numbers can not be safely compared with the = operator

```
// Inaccuracy in binary representation
let a = 1.1
let b = 0.1
let sum = 1.2

a + b = sum // ! returns false !
```

Decimal type

- Represent real numbers
- Precision of 28-29 digits
- Almost no loss of precision
- No round-off errors
- Default value 0.0M

Mostly used for financial calculations

Arithmetic operators

- An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
**	Exponentiation Operator, raises an operand to the power of another	B**A will give 20^{10}

let A = 10
and
let B = 20

Operator precedenc

- The operators *, /, % are evaluated before the operators +, - because *, /, % have higher precedence than +, -
  Example:

$$2 + \underbrace{4 * 5}$$

$$2 + \underbrace{20}$$

$$22$$

- To change the order use parentheses:
  Example: (2 + 4) * 5 evaluates to 30

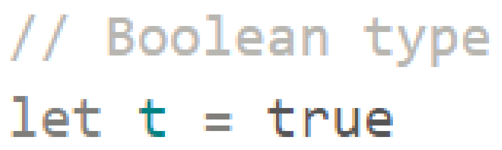
Comparison operators

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A <> B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

let A = 10
and
let B = 20

Boolean type

- Declared by the bool keyword
- Has two possible values: true and false
- Is useful in logical expressions
- Default value false



```
// Boolean type
let t = true
```

Boolean operators

Operator	Description	Example
&&	Called Boolean AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Boolean OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
not	Called Boolean NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	not (A && B) is true.

let A = true
and
let B = false

Char type

- Represents symbolic information
- Declared by using the char keyword
- Gives each symbol a corresspsonding integer code
- Default value ‚\\000‘

```
// Char type
let c = 'A'
```

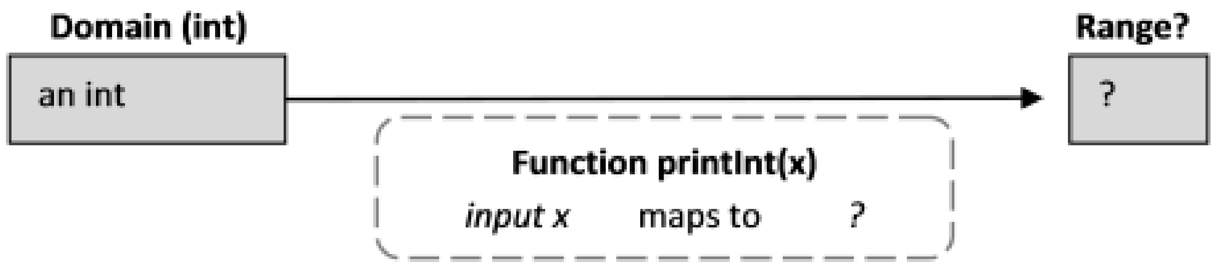
String type

- Represents a sequence of information
- Declared by using the string keyword
- Devault value *null*
- Strings are enclosed in quotes
- Strings can be concatenated using the + operator

```
// String type
let str = "text"

for c in str do
    printfn "%c" c
```

The ‘unit’ type

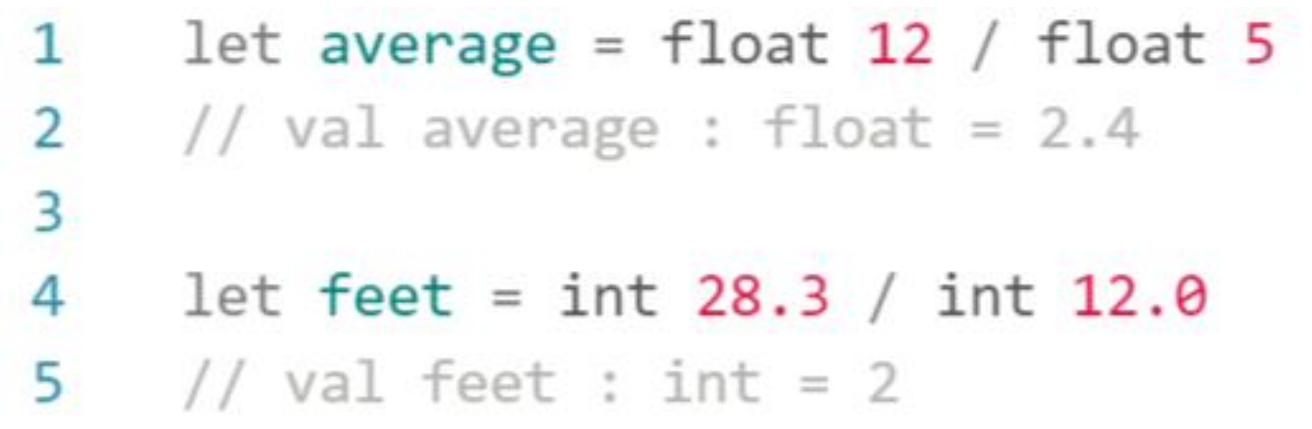
- Sometimes we want a function to do something without returning a value
- Example: PrintInt



```
// Unit() type
let u = ()
```

Type Casting

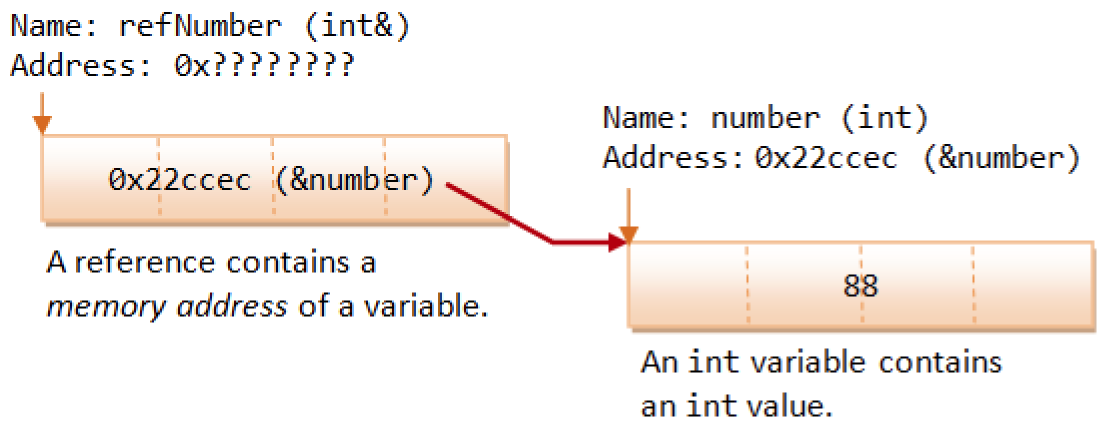
- Type casting tells how to convert one type to another
- Uses:
  - Convert an *int* to a *float* to force floating-point division.
  - Truncate a *float* to an *int*.



```
let average = float 12 / float 5
// val average : float = 2.4

let feet = int 28.3 / int 12.0
// val feet : int = 2
```

References

- A reference works as a pointer. A reference is declared as an alias of a variable. It stores the address of the variable, as illustrated:


Name: refNumber (int&)
Address: 0x????????
0x22ccec (&number)
A reference contains a *memory address* of a variable.
Name: number (int)
Address: 0x22ccec (&number)
88
An int variable contains an int value.

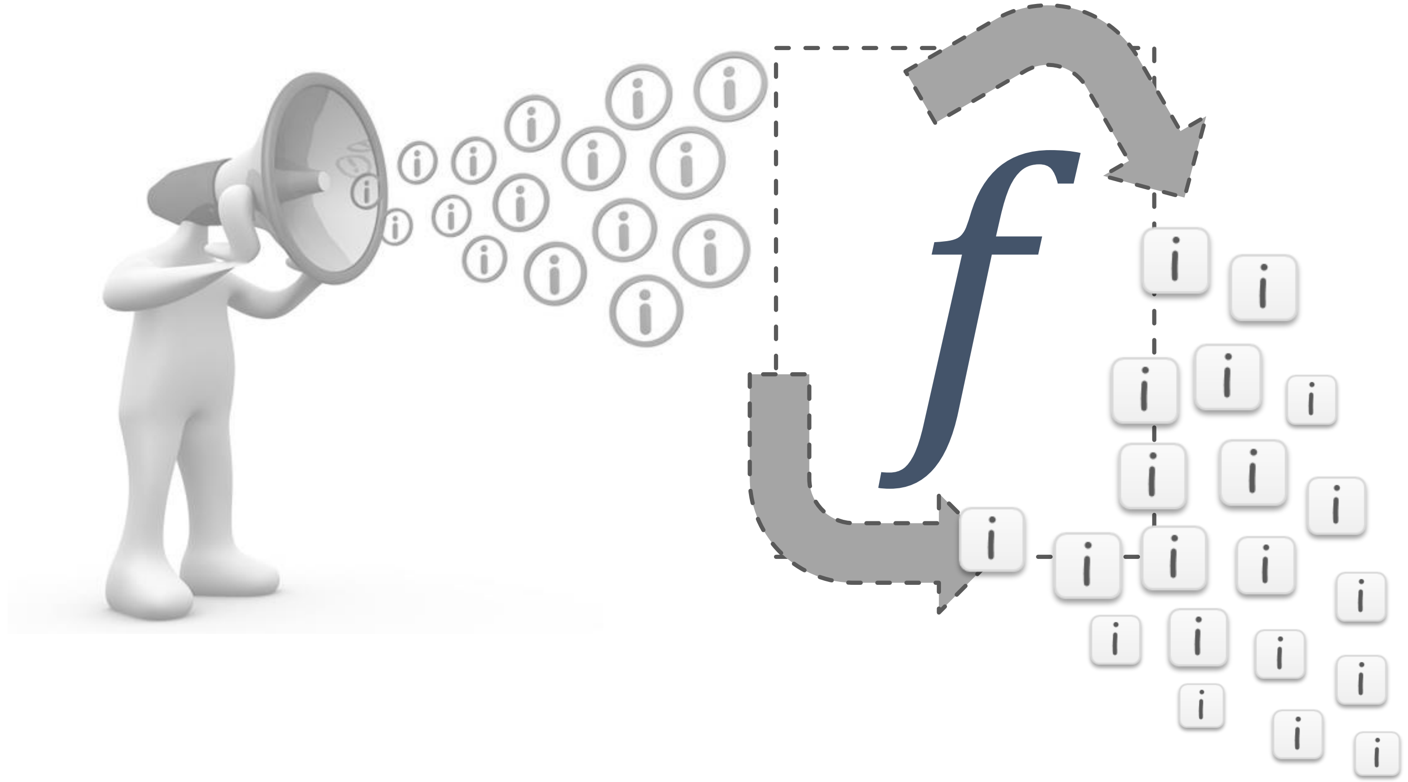
Common primitive types

Type	Domain
bool	Possible values are true and false.
byte	Values from 0 to 255.
int	Values from -2,147,483,648 to 2,147,483,647.
float	A 64-bit floating point type.
char	Unicode character values.
unit	Indicates the absence of an actual value. The type has only one formal value, which is denoted (). The unit value, (), is often used as a placeholder where a value is needed but no real value is available or makes sense.

Data types: primitive types

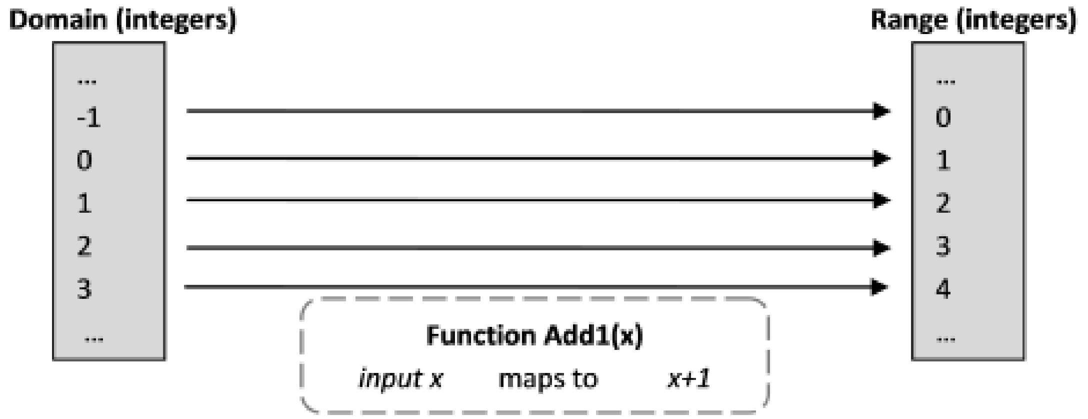
Type	Domain
bool	Possible values are true and false.
byte	Values from 0 to 255.
sbyte	Values from -128 to 127.
int	Values from -2,147,483,648 to 2,147,483,647.
uint32	Values from 0 to 4,294,967,295.
int64	Values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
uint64	Values from 0 to 18,446,744,073,709,551,615.
float32	A 32-bit floating point type.
float	A 64-bit floating point type.
decimal	A floating point data type that has at least 28 significant digits.
char	Unicode character values.
string	Unicode text.
unit	Indicates the absence of an actual value. The type has only one formal value, which is denoted (). The unit value, (), is often used as a placeholder where a value is needed but no real value is available or makes sense.

Functions (f)



- Model the input and output information -> domain modeling

A function as instruction



- Domain: The set of values that can be used as input to the function
- Range: The set of possible output values from the function
- **The function is said to map the domain to the range**

Succinct syntax of F#

- Function declaration
  - Always begin with keyword "let":

```
let f x       = x + 1                                      F#
let sqr y     = y * y
let force x = x * gravity
```

  - In general:

```
let functionName parameter1 (parameter2 …) = body
```

  - Shortcut: anonymous function

    function name is replace by keyword "fun"

```
fun parameter -> body
```

Succinct syntax of F#

- Multiple lines
  - Multiple lines of code is using indentation:

```
let rec fib x =
    if   x <= 0 then 0
    elif x =  1 then 1
    else fib (x-1) + fib (x-2)
```

F#

Calling a function

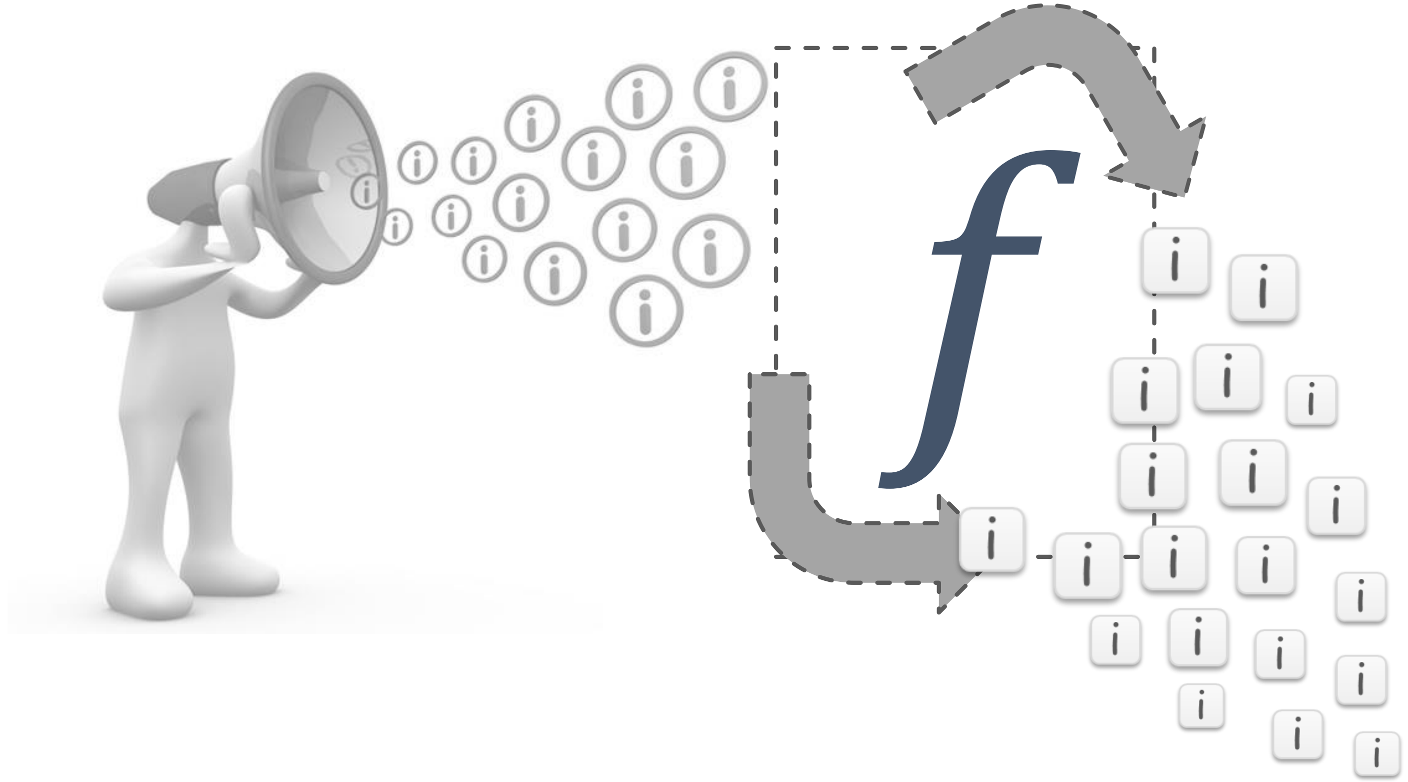
- Function declaration
  - Always begin with keyword “let”:

```
let sqr y = y * y
val sqr : y:int -> int
```
F#

- Function calling:

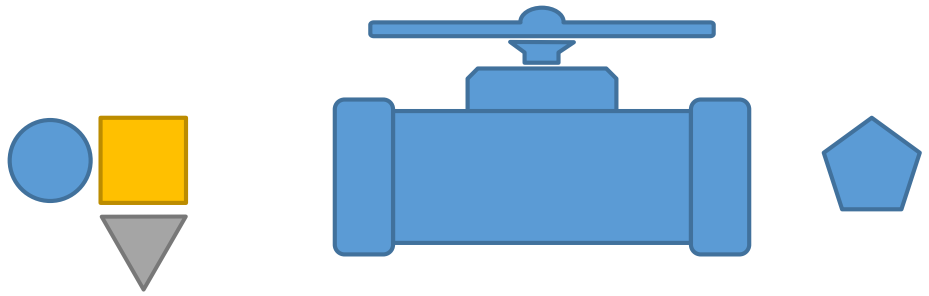
```
sqr 5
val it : int = 25
```
F#

Functional data types



- Model the input and output information -> domain modeling

Functional data types



Function:
union --> record

▶ Model the input and output information by combining primitive data types

Data types

- Definition: A type is an annotation to a value which defines the kind of the value you and the compiler has to deal with.

- The different types can be grouped in many different ways:
    - Primitive Types
    - Functional Types
    - Object Types

Data types: primitive types

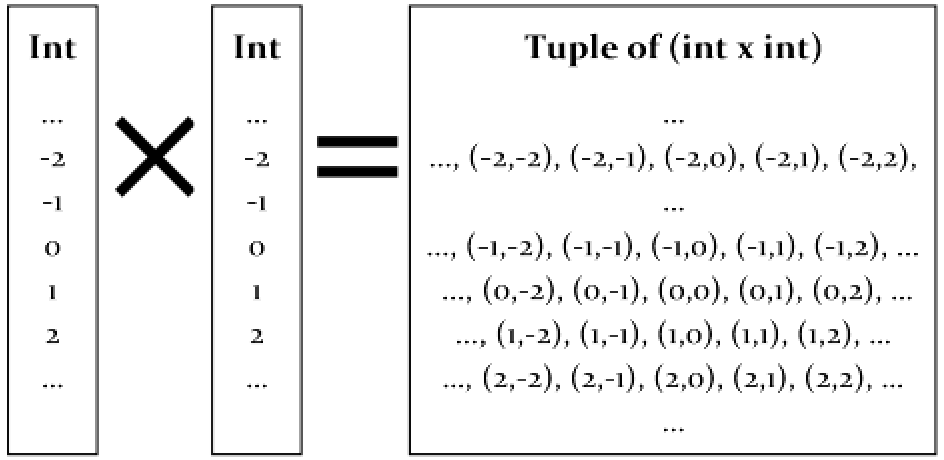
Type	Domain
bool	Possible values are true and false.
byte	Values from 0 to 255.
sbyte	Values from -128 to 127.
int	Values from -2,147,483,648 to 2,147,483,647.
uint32	Values from 0 to 4,294,967,295.
int64	Values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
uint64	Values from 0 to 18,446,744,073,709,551,615.
float32	A 32-bit floating point type.
float	A 64-bit floating point type.
decimal	A floating point data type that has at least 28 significant digits.
char	Unicode character values.
string	Unicode text.
unit	Indicates the absence of an actual value. The type has only one formal value, which is denoted (). The unit value, (), is often used as a placeholder where a value is needed but no real value is available or makes sense.

Functional types:

- You can create new types in F# by combining simple data types
- The way of combination can be subdivided in three functional type subclasses:
    - Product types
        - Tuples
        - Records
    - Sum types
        - Discriminated unions
        - Enum types
    - Collection types

Product type: Tuples

- Combining simple data types by multiplying :



F#

```
let t1 = (2,3)
```

- This "product" approach can be used to make tuples out of any mixture of types

Product type: Records

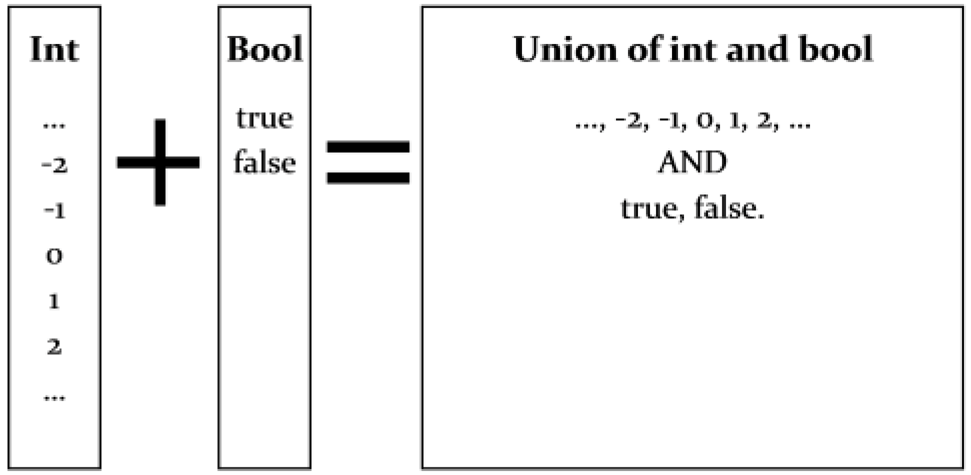
- A record type is like the tuple type where each element is labeled:

```fsharp
type ComplexNumber = { real: float; imaginary: float }

let myComplexNumber = { real = 1.1; imaginary = 2.2 }

myComplexNumber.real
```

F#

Sum Types: Discriminated unions

- Combining simple data types by summing :



F#

```
type IntOrBool =
  | I of int
  | B of bool
```

- This “sum" approach can be used to define a function that works with one type OR the other

Sum Types: Enum

- The enum types are very similar to the discriminant unions
- Enum types sum constant values of simple data types

F#

```
type ColorEnum =
   | Red    = 0
   | Yellow = 1
   | Blue   = 2
```

Collection types

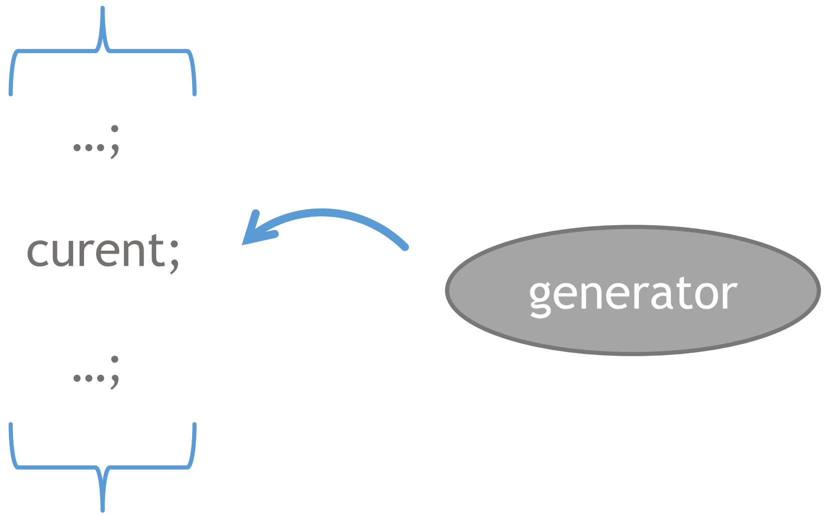
- Sequence types or collection types are a series of items or values
- Different collection types have different properties regarding their manipulation and efficiency
- The most important types are:
    - Seq (IEnumerable)
    - List
    - Array
    - Map
    - Set

Collection types: Seq / IEnumerable

- Seq is an abstract representation of any lazily evaluated sequences of values
- A lazily evaluated sequence only evaluates the elements when they are required

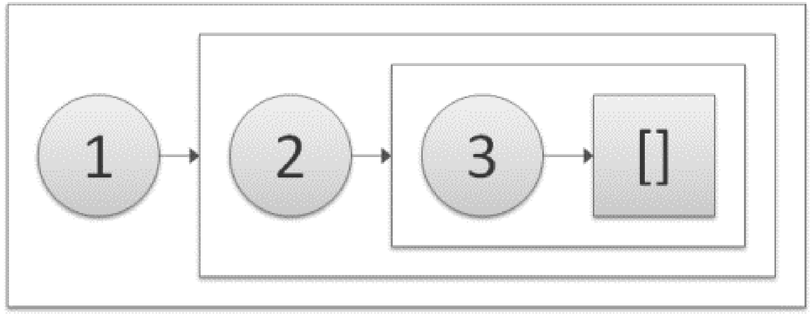
F#

```
seq { 0 .. 100 }

seq { for i in 1 .. 10 do yield i }
```



Collection types: Lists

- A linked list is a data structure consisting of a value and a pointer to the next:

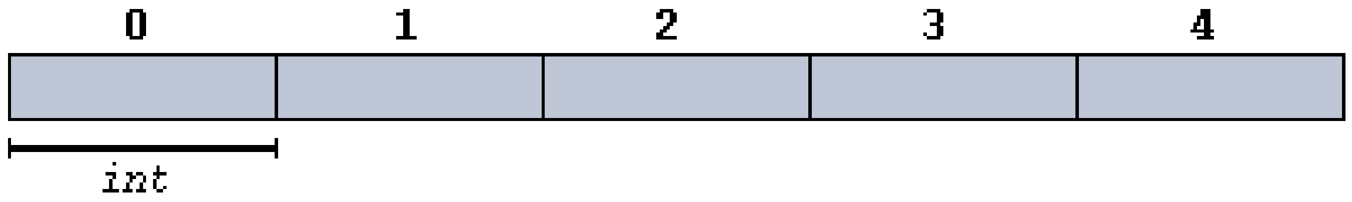


```fsharp
let list1_10 = [ 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; ]
```
F#

- List can be appended and accessed from head to tail very efficiently

Collection types: Arrays

- Arrays are ordered, integer-indexed collections of any elements:



```
let array1_10 = [| 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; |]
```

F#

- The size of the collection is fixed once created
- Arrays are mutable and the values can be accessed and changes efficiently

Collection types: Maps

- Maps are collections that associate keys with corresponding values and allow values to be looked up by key efficiently:

	Keys		Values
pair 1	"key_1"	.-->	1
pair 2	"key_2"	.-->	2
pair 3	"key_3"	.-->	3

```
let mass = Map [“Hydrogen”,1.0079; “Carbon”, 12.001]
```

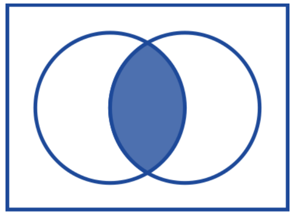
F#

Collection types: Sets

- Sets are immutable, sorted and unique (duplicates are removed) collections.
- They allow for fast:
  - insertion/deletion
  - membership
  - union/difference and intersection (**Set-theoretic operations**)

F#

```
let sample1 = Set [“Hydrogen”; “Carbon”;]
let sample2 = Set [“Carbon”; “Nitrogen”;]

Set.inter sample1 sample2
val it : Set<string> = [“Carbon”;]
```



$\{sample1\} \cap \{sample2\}$

Object oriented

- You can program imperative and object oriented in F# …
- … but you should always keep a functional mindset.

- F# object types seamlessly interoperate with OOP from .NET

Object Types in F#

- F# is fundamentally a functional language, but it is considered “impure” -> Object can also be first class members
- Objects consist of:
  - Constructor
  - Methods / Fields (Properties)
- Objects can inherit from other objects (abstract)

Creating objects

F#

```
type DNASequence(sequence:string) =
     member self.Sequence = sequence
     member self.Length   = sequence.Length

     member self.revers() =
         self.Sequence.ToCharArray()
         |> Array.rev
  ...
```

- declaration

F#

```
let dna = new DNASequence(“ATGGTC”)
```

- instantiation

Inheritance

F#

```
type Description(id:string) =
    member self.Id = id

    member self.Print()  = printfn “%s” id
```

F#

```
type DNASequence(id:string,sequence:string) =

    inherit Description(id)

    member self.Sequence = sequence
    member self.Length   = sequence.Length

    member self.revers() =
        self.Sequence.ToCharArray()
        |> Array.rev

    ...
```