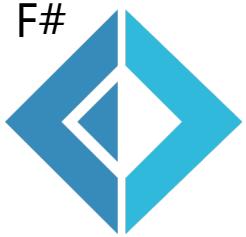
Introduction to Functional Programming

Teachers:

- Jon Sporring, Professor, UCPH
- Sarah Hyatt, Teaching Assistant, UCPH
- Ma Mengfan, Teaching Assistant, UESTC
- Tao Binglin Teaching Assistant, UESTC
- Li Mengying, Teaching Assistant, UESTC

F#



Notes:

- "Learning to program with F#"
- "Report template"
- Assignments + code + data

Hand-ins:

- Exercise 1: Individual
- Exercise 2: Groups of 2

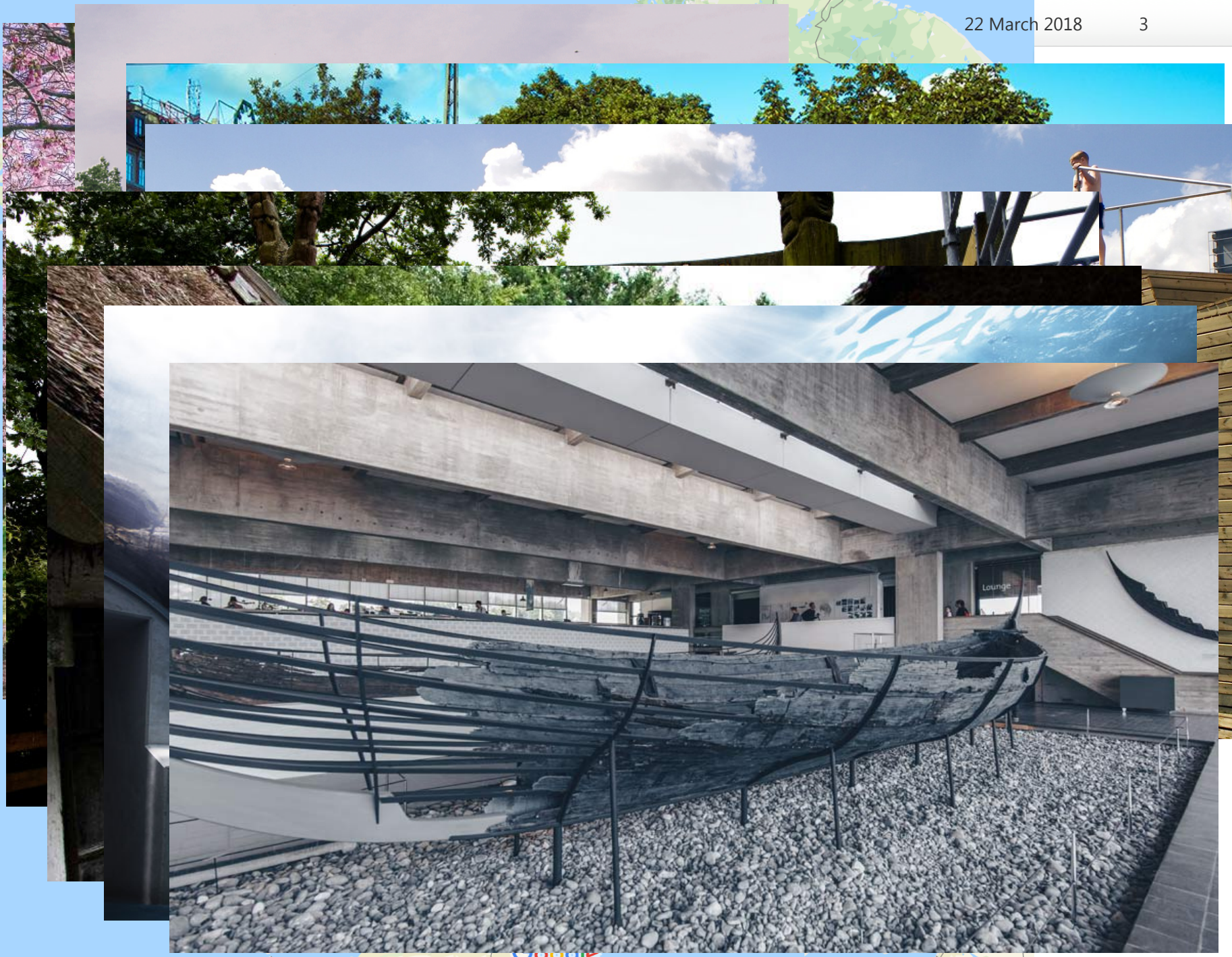
Time	Monday	Tuesday	Wednesday	Thursday	Friday
9:00--10:20	Introduction to F#	Pattern matching	Report writing	Open question session	Exercise hand-in 2
10:35--11:55	Imperative versus Functional programming	Working with lists	Exercise hand-in 1	Exercise hand-in 2	Exercise hand-in 2
14:30--15:50	Exercise F#	Exercise functional programming	Exercise hand-in 1	Exercise hand-in 2	Exercise hand-in 2
16:05--17:25	Programming with recursion	Exercise functional programming	Exercise hand-in 1	Exercise hand-in 2	Exercise hand-in 2
22:00			Deadline 1		Deadline 2

Introduction to
The University of Copenhagen

Jon Sporring,
Ph.d., Professor,
Deputy Head of Department for Research

Department of Computer Science,
University of Copenhagen
Universitetsparken 1,
DK-2100 Copenhagen, Denmark
Email: sporring@di.ku.dk

UNIVERSITY OF COPENHAGEN

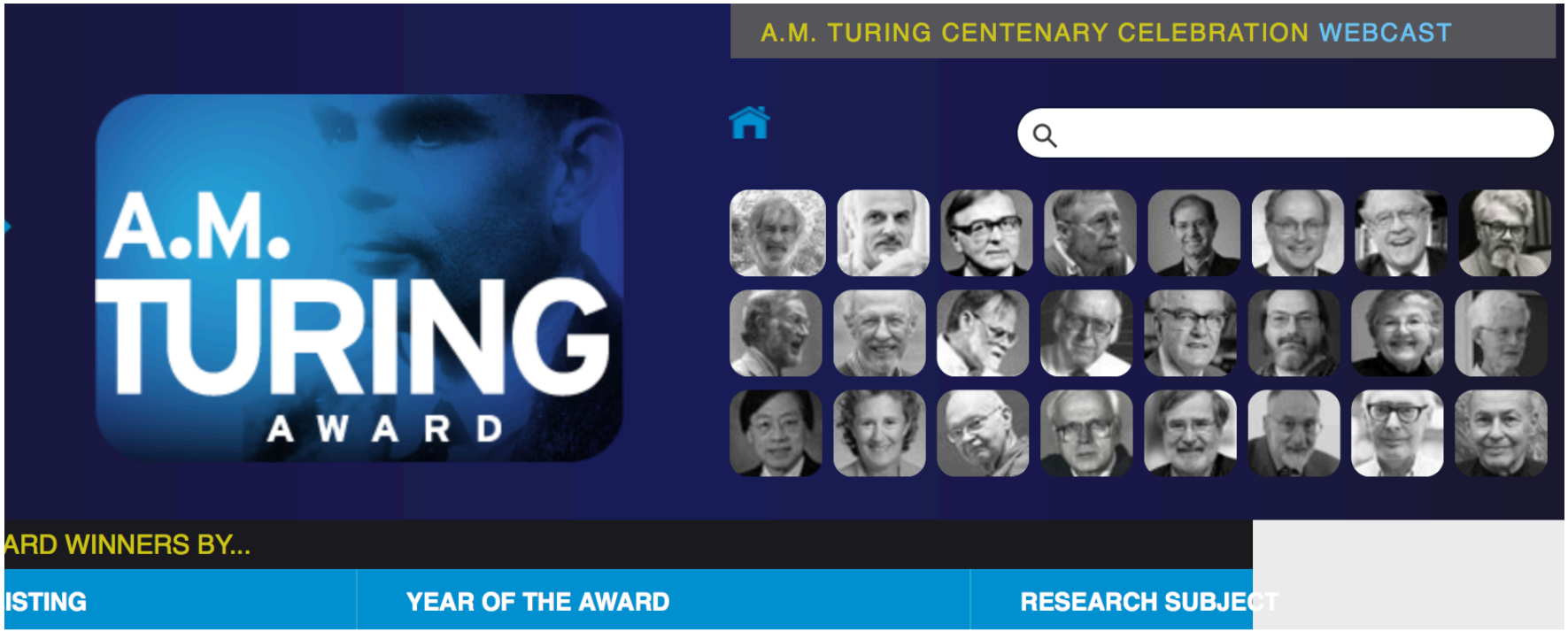

Lounge

University of Copenhagen

- The largest university in Scandinavia
- Founded in 1479 by King Christian I as a Catholic Seminary
- 40,000 students
- 9,000 employees, including 5,000 scientists
- 9 Nobel prizes

Niels Bohr (1885-1962), Nobel prize winner in Physics 1922

Department of Computer Science (DIKU)



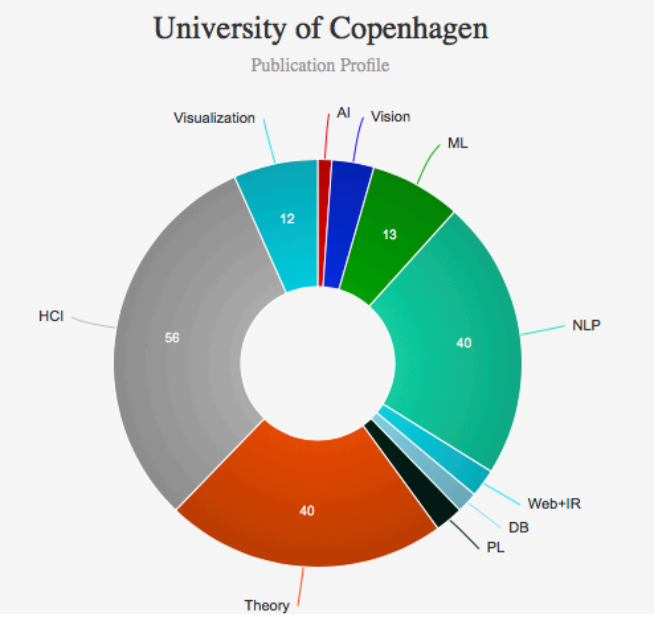
PETER NAUR

Denmark – 2005

CITATION

For fundamental contributions to programming language design and the definition of Algol 60, to compiler design, and to the art and practice of computer programming.

CSRanking, Scopus, Google Scholar citations: 2013-2018



#	Institution	Count	Faculty
1	Carnegie Mellon University	48.2	60
2	Stanford University	28.1	27
3	University of Washington	22.1	35
4	Cornell University	19.4	31
5	Massachusetts Institute of Technology	18.7	32
6	University of California - Berkeley	17.9	31
7	University of Copenhagen	17.8	16
8	Georgia Institute of Technology	17.5	30
9	University of Toronto	16.7	24
10	University of Michigan	14.8	28

Computer Science at UCPH 2013-2018	Avg. Citations per Publication	Avg. Scholarly Output
University of California at Berkeley	9,8	985
ETH Zurich	8,0	1096
University of Copenhagen	7,1	281
University College London	6,4	899
McGill University	4,5	580

Researcher	Citations 2013-2018
Ingemar Cox	9467
Mikkel Thorup	4820
Christian Igel	4282
Kasper Hornbæk	4074
Marleen de Bruijne	3155

Main programmes at Dept. of Computer Science

Year	Age	School
1	6	Primary School
2	7	
3	8	
4	9	
5	10	
6	11	
7	12	
8	13	
9	14	
10	15	
11	16	High school
12	17	
13	18	
14	19	Bachelor
15	20	
16	21	
17	22	Master
18	23	
19	24	
20	25	Ph.D.
21	26	
22	27	

5 Bachelor educations w. Computer Science

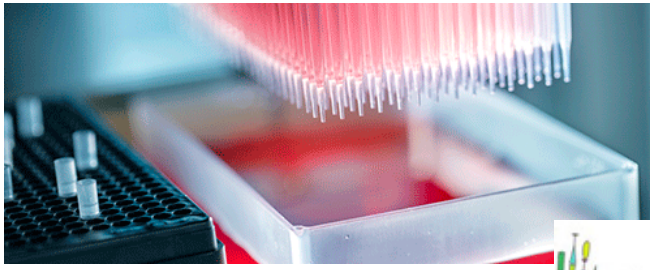
- Mainly taught in Danish
- 3 years full study – 180 ECTS
- 15 ECTS bachelor project
- Programs:
  - Computer science
  - Computer science and economy
  - Machine learning and data science
  - IT and health
  - IT and communication

3 Master educations w. Computer Science

- English
- 2 years full time study – 120 ECTS
- 30 ECTS master thesis
- Programs:
  - Computer Science
  - Communication and IT
  - Health Informatics
  - IT and Cognition

Note, 60 ETCS = 1 years full time study, Typical course size 7.5 or 15 ECTS

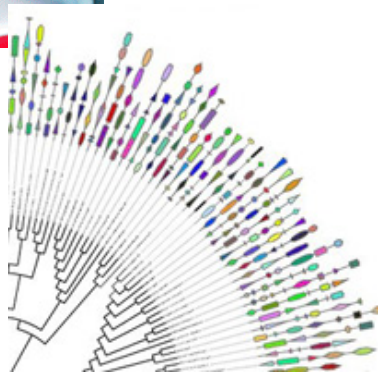
Collaborations at the Faculty of Science

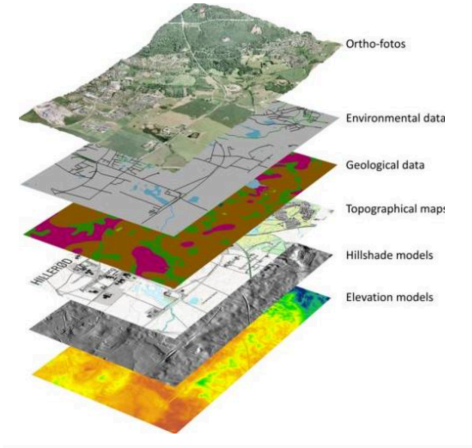


Center for Quantum Devices

QMATH

STATION Q





Collaborations in Greater Copenhagen Area

REGION Region Hovedstaden

Functional Programming

Crash course

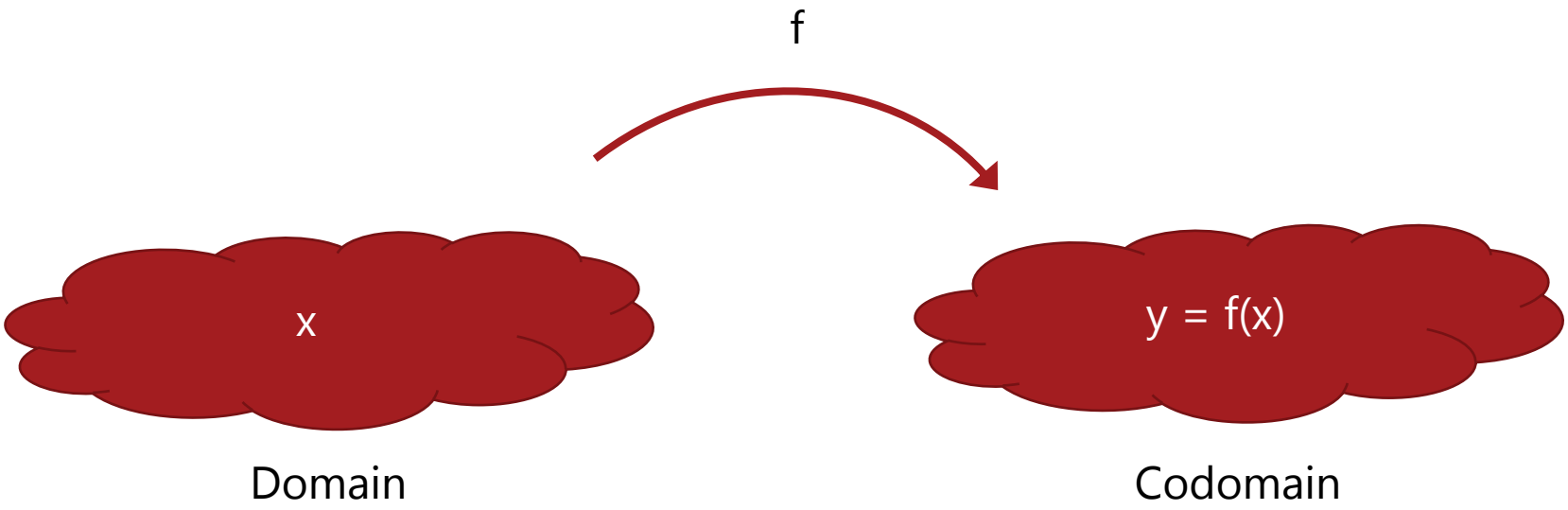
Jon Sporring

2019/07/08

UNIVERSITY OF COPENHAGEN

Functional Programming

A programming paradigm using functions and avoids changing-state and mutable data.



For every element in the domain, there is exactly one element in the codomain!

Functional code:

- has no side-effects,
- typically use recursion, and
- treats functions as values

F# is a functional first programming language

- It primarily supports functional programming, but also imperative, object-oriented, and event-driven programming.
- It is part of the .NET family (C# etc.)
- Its lightweight syntax requires indentation like Python
- It is well suited for programming parallel and concurrent programs.
- It has an interactive and a compile mode

Demo: myFirstFsharp.fsx

```
let a = 357
let b = 864
let c = a + b
do printfn "%A" c
```

- fsharpi -> indtast myFirstFsharp.fsx
- fsharpi myFirstFsharp.fsx
- fsharpc myFirstFsharp.fsx && mono myFirstFsharp.exe

F# is a strongly typed language

Type	int	float	char	string	float	float
Three	3	3.0	'3'	"3"	3e0	3.0e0

Type	syntax	Examples	Value
int, int32	<int or hex> <int or hex>1	3, 0x3 31, 0x31	3

Metatype	Type name
Boolean	bool
Integer	int
	byte
	sbyte
	int8
	uint8
	int16
	uint16
	int32
	uint32
	int64
	uint64
Real	float
	double
	single
	float32
	decimal
Character	char
	string
None	unit
Object	obj
Exception	exn

Operator	Associativity	Description
+<expr>, -<expr>, ~~~<expr>	Left	Unary identity, negation, and bitwise negation operator
f <expr>	Left	Function application
<expr> ** <expr>	Right	Exponent
<expr> * <expr>, <expr> / <expr>, <expr> % <expr>	Left	Multiplication, division and remainder
<expr> + <expr>, <expr> - <expr>	Left	Addition and subtraction binary operators
<expr> ^^^ <expr>	Right	bitwise exclusive or
<expr> < <expr>, <expr> <= <expr>, <expr> > <expr>, <expr> >= <expr>, <expr> = <expr>, <expr> <> <expr>, <expr> <<< <expr>, <expr> >>> <expr>, <expr> &&& <expr>, <expr> ||| <expr> ,	Left	Comparison operators, bitwise shift, and bitwise 'and' and 'or'.
<expr> && <expr>	Left	Boolean and
<expr> || <expr>	Left	Boolean or

Operators, types, precedence, and association

Operators	Precendens & association	Typecasting	Unære operators
3 + 4	exp 0.0	float 3	2 - 3
3.0 + 4.0	exp 1.0	int 3.2	-3
~~3 + 4.0~~	exp 0.0 + 1.0	int 3.6	char (int 'c' + -int 'a' + int 'A')
5 / 2	2.0 **(3.0 ** 4.0)	int (3.2 + 0.5) = 3	~~char (int 'c' -int 'a' + int 'A')~~
5 % 2	(2.0 / 3.0)/ 4.0	int (3.6 + 0.5) = 4	
2 * (5 / 2) + 5 % 2		int 'a'	
2.0 ** 3.0		int 'A'	
pown 2 3		char 65	
”hello " + ”world"		char (int 'c' - int 'a' + int 'A')	

String slicing, Boolean values and operators

Slicing

"abcdefghijkl".[1] = 'b'

"abcdefghijkl".[1..4] = "bcde"

"abcdefghijkl".[..4] = "abcde"

"abcdefghijkl".[4..] = "efghijkl"

"abcdefghijkl".Length = 12

"abcdefghijkl".[0..11] = "abcdefghijkl"

Boolean values and operators

true = 1

false = 0

a && b

a || b

not a

Comparison operators

3 < 4

3 > 4

3 <> 4

3 = 4

~~not 3 = 4~~

not (3 = 4)

Bindings of values

verbose syntax

```
let name = "World” in do printfn "Hello %A" name
```

Lightweight syntax

```
let name = "World”
do printfn "Hello %A" name
```

Optional ‘do’

```
let name = "World”
printfn "Hello %A" name
```

Sequence of statements

```
let name = "World" in do printfn "Hello %A" name; do printfn "Goodbye %A" name
```

Scope

Names (in outer scope) can not be overwritten

```
let name = "World"
let name = "Jon"
do printfn "Hello %s" name
```

Scope can be controlled with parantheses

```
let greeting = "Hello"                                  0
let name = "Jon"
do printfn "%s %s" greeting name
(
    let name = "Anders"                                 1
    do printfn "%s %s" greeting name
)
do printfn "%s %s" greeting name
```

Functions are the work-horse of F#

Example: Find the solutions to a second degree polynomial equation

```
let discriminant a b c =
  b ** 2.0 - 4.0 * a * c

let solution a b c sgn =
  let d = discriminant a b c
  (-b + sgn * sqrt d) / (2.0 * a)

let a = 1.0
let b = 0.0
let c = -1.0
let xp = (solution a b c +1.0)
printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
```

$$ax^2 + bx + c = 0$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Demo: square.fsx

And now with types

Example: Find the solutions to a second degree polynomial equation

$$ax^2 + bx + c = 0$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
let discriminant (a:float) (b:float) (c:float) : float=
  b ** 2.0 - 4.0 * a * c

let solution (a:float) (b:float) (c:float) (sgn:float) : float =
  let d = discriminant a b c
  (-b + sgn * sqrt d) / (2.0 * a)

let a = 1.0
let b = 0.0
let c = -1.0
let xp = (solution a b c +1.0)
printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
```

F# will infer types for you. Being explicit can help you catch errors as you program!

Mutable values and loops: Imperative features

```
for i = 1 to 10 do
  printf "%d " i
printfn ""
```

```
let mutable x = 5
printfn "%d" x
x <- 3
printfn "%d" x
```

```
let mutable i = 1
while i <= 10 do
  printf "%d " i
  i <- i + 1
printf "\n"
```

Tuples (product types)

```
$fsharpi
...
> let a = (1, 1.0);;
val a : int * float = (1, 1.0)
> printfn "%A %A" (fst a) (snd a);;
1 1.0
val it : unit = ()
> let b = 1, "en", '\049'
val b : int * string * char = (1, "en", '1')
```

Product type

Indexing functions for pairs

Parenthes are unnecessary but recommended

Pattern matching

```
> let (b1, b2, b3) = b;;
val b3 : char = '1'
val b2 : string = "en"
val b1 : int = 1
> let mutable c = (1,2)
- c <- (2,3)
- printfn "%A" c;;
(2, 3)
val mutable c : int * int = (2, 3)
val it : unit = ()
```

A tuple can be mutable, not its elements

Fibonacci: $f_n = f_{n-1}+f_{n-2}$, $f_1=f_2=1$

For-loops

```
let mutable m = 1
let mutable n = 1
let N = 5
for i = 3 to N do
  let p = m + n
  m <- n
  n <- p
printfn "%d: %d" N n
```

While-loops

```
let mutable m = 1
let mutable n = 1
let mutable i = 3
let N = 5
while i <= 5 do
  let p = m + n
  m <- n
  n <- p
  i <- i + 1;
printfn "%d: %d" N n
```

Tuple + for-loops

```
let mutable pair = (1,1)
let N = 5
for i = 3 to N do
  pair <- (snd pair, fst pair + snd pair)
printfn "%d: %d" N (snd pair)
```

```
let fib N =
  let mutable pair = (1,1)
  for i = 3 to N do
    pair <- (snd pair, fst pair + snd pair)
  snd pair


let N = 5
printfn "%d: %d" N (fib N)
```

Conditions

If-then-else

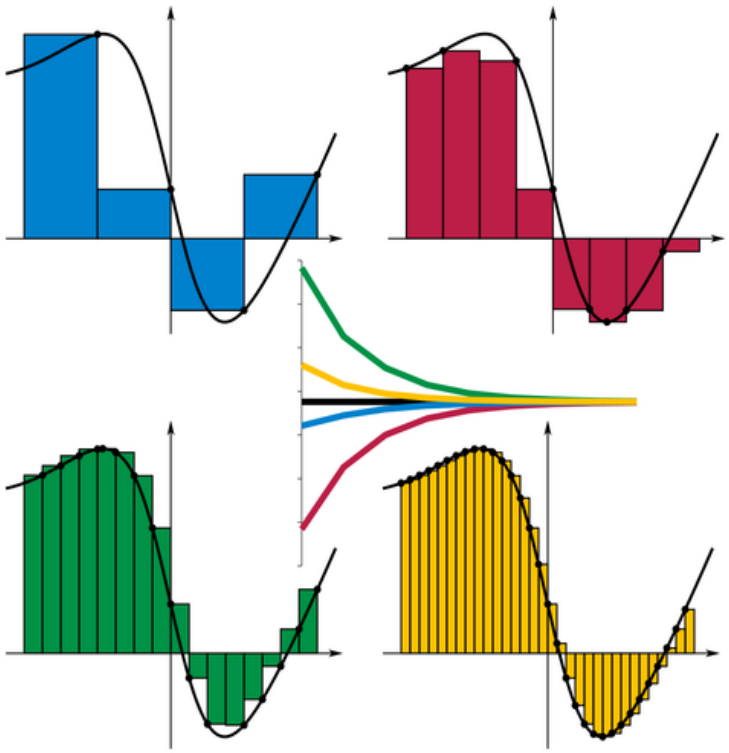
```
> if 3 < 2 then
-   printfn "3 < 2"
- else
-   printfn "3 >= 2";;
3 >= 2
val it : unit = ()

> let str =
-   if 3 < 2 then
-     "3 < 2"
-   else
-     "3 >= 2";;
val str : string = "3 >= 2"
```

If-chains

```
> let str =
-   if 3 < 2 then
-     "3 < 2"
-   elif 3 = 2 then
-     "3 = 2"
-   else
-     "3 > 2";;
val str : string = "3 > 2"
```

Higher-order functions: Numerical integration



By I, KSmrq, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=2347919

```
/// Estimate the integral of f
/// from a to b with stepsize d
let integrate f a b d =
  let mutable sum = 0.0
  let mutable x = a
  while x < b do
    sum <- sum + d * (f x)
    x <- x + d
  sum

let a = 0.0
let b = 1.0
let d = 0.01
let result = integrate exp a b d
printfn "Int_%g^%g exp(x) dx = %g" a b result
```

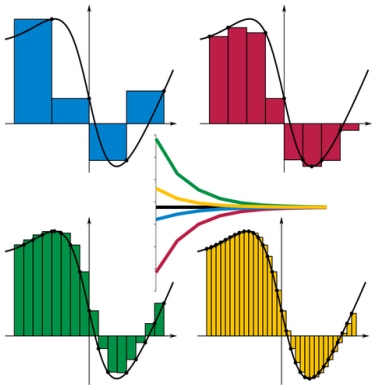
Anonymous functions

```
let f x = x * exp(x)
f 3.0
```

```
let f = fun x -> x * exp(x)
f 3.0
```

```
/// Estimate the integral of f
/// from a to b with stepsize d
let integrate f a b d =
  let mutable sum = 0.0
  let mutable x = a
  while x < b do
    sum <- sum + d * (f x)
    x <- x + d
  sum

let a = 0.0
let b = 1.0
let d = 1e-5
let result = integrate (fun x -> x * exp(x)) a b d
printfn "Int_%g^%g f(x) dx = %g" a b result
```



By I, KSmrq, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2347919

Recursive functions

Example:

Faculty function n!

5! = 1*2*3*4*5

```
/// faculty function using a for-loop
let facFor n =
  let mutable prod = 1
  for i = 2 to n do
    prod <- prod * i
  prod

let n = 5
printfn "fac %d = %d" n (facFor n)
```

facRec 5 → 5 * (facRec 4)
→ 5 * (4 * (facRec 3))
→ 5 * (4 * (3 * (facRec 2)))
→ 5 * (4 * (3 * (2 * (facRec 1))))
→ 5 * (4 * (3 * (2 * 1)))
→ 5 * (4 * (3 * 2))
→ 5 * (4 * 6)
→ 5 * 24
→ 120

```
/// faculty function using recursion
let rec facRec n =
  if n > 1 then
    n * facRec (n-1)
  else
    1

let n = 5
printfn "fac %d = %d" n (facRec n)
```