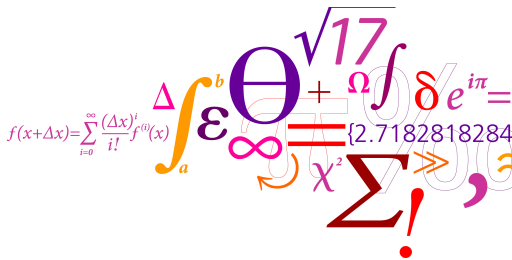


# 02157 Functional Programming

## Lecture 1: Introduction and Getting Started

Michael R. Hansen



**DTU Compute**

Department of Applied Mathematics and Computer Science

# WELCOME to 02157 Functional Programming

**Teacher:** Michael R. Hansen, DTU Compute      [mire@dtu.dk](mailto:mire@dtu.dk)

**Teaching assistants:**

Kristian Hedemark Krarup  
Maliina Bolethe Skytte Hammeken  
Matias Daugaard Holst-Christensen  
Per Lange Laursen  
Van Anh Thi Trinh

**Homepage:** [www.compute.dtu.dk/courses/02157](http://www.compute.dtu.dk/courses/02157)

# The Functional Setting

A program  $f$  is a function

$$f : \textit{Argument} \rightarrow \textit{Result}$$

- $f$  takes **one argument** and produces **one result**
- arguments and results can be **composite values**
- **computation** is governed by **function application**

For example:

```
insert(4, [1; 3; 7; 9]) = [1; 3; 4; 7; 9]
```

- the argument is a **pair**:  $(4, [1; 3; 7; 9])$
- the result is a **list**:  $[1; 3; 4; 7; 9]$
- the computation is guided by **repeated function application**

```

                insert(4, [1; 3; 7; 9])
    ~~~~~>      1::insert(4, [3; 7; 9])
    ~~~~~>      1::3::insert(4, [7; 9])
    ~~~~~>      1::3::4::[7; 9]
    ~~~~~>      = [1; 3; 4; 7; 9]
```

# A simple problem solving technique

Solve a complex problem by

- partitioning it into smaller well-defined parts
- compose the parts to solve the original problem.

The main goal:

A program is constructed by  
combining simple well-understood pieces

- A general technique that is natural in functional programming.

Insertion in an ordered list is a well-understood program.

- Can you use this in the construction of program for sorting a list?

# A typed functional programming language

## Supports

- Composite values like lists and trees
- Functions as "first-class citizens"
- Recursive functions
- Patterns
- A strong polymorphic type system
- Type inference

A small collection of powerful concepts used to explain the meaning of a program

- binding
- environment
- evaluation of expressions

A value-oriented (declarative) programming approach where you focus on *what* properties a solution should rather than on the individual computation steps.

does **not** support

- assignable variables, assignments

```
a[i] = a[i] + 1, x++, ...
```

- imperative control statements

```
while i<10 do ...  
if b then a[i]=a[i]+1,...
```

- object-oriented state-changing features

```
child.age = 5
```

- ...

In imperative and object-oriented programming approaches focus is on **how** a solution is obtained in terms of a sequence of state changing operations.

Some advantages of functional programming:

- Supports fast development based on abstract concepts  
more advanced applications are within reach
- Supplements modelling and problem solving techniques
- Supports parallel execution on multi-core platforms

F# is as efficient as C#

Testimonials, quotes and real-world applications of FP languages:

- <http://fsharp.org>
- [homepages.inf.ed.ac.uk/wadler/realworld](http://homepages.inf.ed.ac.uk/wadler/realworld)

Functional programming techniques once mastered are useful for the design of programs in other programming paradigms as well.

# Some functional programming background

- Introduction of  $\lambda$ -calculus around 1930 by Church and Kleene when investigating function definition, function application, recursion and computable functions. For example,  $f(x) = x + 2$  is represented by  $\lambda x.x + 2$ .
- Introduction of the type-less functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Introduction of functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) in the 1970s.
- Functional languages (SML, Haskell, OCAML, F#, ...) have now applications far away from their origin: Compilers, Artificial Intelligence, Web-applications, Financial sector, ...
- Declarative aspects are now sneaking into "main stream languages"
- [Peter Sestoft: Programming Language concepts, Springer 2012.](#) Uses F# as a meta language.



- Textbook: [Functional Programming using F#](#)  
M.R. Hansen and H. Rischel. Cambridge Univ. Press, 2013.

[www.imm.dtu.dk/~mrh/FSharpBook](http://www.imm.dtu.dk/~mrh/FSharpBook)

Available at DTU's bookstore and library.

- F# (principal designer: Don Syme) is an [open-source](#) functional language. F# is integrated in the Visual Studio platform and with access to all features in the .NET program library. The language is also supported on Linux, MAC, ... systems
- Look at <http://fsharp.org> concerning installations for your own laptops (Windows, Linux, Mac, Android, iPhone, ...).  
Suggestions: Windows: Visual Studio and for Mac and Linux: Visual Studio Code
- Lectures: Friday 8.15 - 10:00 in Building 341, **Auditorium 21**
- Exercises classes: Friday 10:00 - 12:00 in Building 341  
Rooms 003, 015 and 019

- Four mandatory assignments.      Dates will be announced soon
- Three must be approved in order to participate in the exam.
- They do not have a role in the final grade.
- Group af size 1, 2 and 3 are allowed.

Approvals of mandatory assignments from earlier years  
DO NOT apply this year

- **Prerequisites for 02157:**
  - Programming in an imperative/object-oriented language
  - Discrete mathematics (previously or at this semester)
- 02157 is a prerequisite for 02141 Computer Science Modelling
- 02141 is a prerequisite for 02257 Applied Functional Prog.
- **02257 Applied Functional Programming** January course
  - efficient use of functional programming in connection with activities at the **M.Sc. programmes** and in “**practical applications**”.

One project every week. Last year's projects:

- Computer science applications.
  - Compiler for a rich imperative programming language.
- “Practical applications”.
  - A game with a reactive graphical user-interface
- Functional pearls.
  - Layout of trees.

## Part 1 Getting Started:

- The interactive environment
- Values, expressions, types, patterns
- Declarations of values and recursive functions
- Binding, environment and evaluation
- Type inference

Main ingredients of F#

## Part 2 Lists:

- Lists: values and constructors
- Recursions following the structure of lists
- Polymorphism

A value-oriented approach

**GOAL:** By the end of the day you have constructed **succinct**, **elegant** and **understandable** F# programs.

```
2*3 + 4;;
```

⇐ Input to the F# system

```
val it : int = 10
```

⇐ Answer from the F# system

- The *keyword* `val` indicates a value is computed
- The *integer* `10` is the computed value
- `int` is the *type* of the computed value
- The *identifier* `it` names the (last) computed value

The notion *binding* explains which entities are named by identifiers.

`it ↦ 10`      reads: “`it` is bound to 10”

A value declaration has the form: `let identifier = expression`

```
let price = 25 * 5;;
```

← A declaration as input

```
val price : int = 125
```

← Answer from the F# system

The effect of a declaration is a binding: `price ↦ 125`

Bound identifiers can be used in expressions and declarations, e.g.

```
let newPrice = 2*price;;
```

```
val newPrice : int = 250
```

```
newPrice > 500;;
```

```
val it : bool = false
```

A collection of bindings

price	↦	125
newPrice	↦	250
it	↦	false

is called an environment

## Function Declarations 1: `let f x = e`

- `x` is called the *formal parameter*
- the defining expression `e` is called the *body* of the declaration

Declaration of the circle area function:

```
let circleArea r = System.Math.PI * r * r;;
```

- `System.Math` is a program library
- `PI` is an identifier (with type `float`) for  $\pi$  in `System.Math`

The type is *automatically inferred* in the answer:

```
val circleArea : float -> float
```

Applications of the function:

```
circleArea 1.0;; (* this is a comment *)  
val it : float = 3.141592654
```

```
circleArea(3.2);; // A comment: optional brackets  
val it : float = 32.16990877
```

`1.0` and `3.2` are also called *actual parameters*

Recursion. Example  $n! = 1 \cdot 2 \cdot \dots \cdot n$ ,  $n \geq 0$

Mathematical definition:

recursion formula

$$\begin{aligned} 0! &= 1 & (i) \\ n! &= n \cdot (n-1)!, \quad \text{for } n > 0 & (ii) \end{aligned}$$

- $n!$  is defined **recursively** in terms of  $(n-1)!$  when  $n > 0$

Computation:

$$\begin{aligned} &3! \\ = &3 \cdot (3-1)! & (ii) \\ = &3 \cdot 2 \cdot (2-1)! & (ii) \\ = &3 \cdot 2 \cdot 1 \cdot (1-1)! & (ii) \\ = &3 \cdot 2 \cdot 1 \cdot 1 & (i) \\ = &6 \end{aligned}$$



- the function *f* occurs in the body *e* of a *recursive declaration*

A recursive function declaration:

```
let rec fact n =  
    if n=0 then 1                (* i *)  
    else n * fact(n-1);;        (* ii *)  
val fact : int -> int
```

Evaluation:

```
fact(3)  
↪ 3 * fact(3 - 1)      (ii) [n ↦ 3]  
↪ 3 * 2 * fact(2 - 1)  (ii) [n ↦ 2]  
↪ 3 * 2 * 1 * fact(1 - 1) (ii) [n ↦ 1]  
↪ 3 * 2 * 1 * 1        (i)  [n ↦ 0]  
↪ 6
```

$e_1 \rightsquigarrow e_2$  reads: *e*<sub>1</sub> evaluates to *e*<sub>2</sub>

- An environment is used to bind the formal parameter *n* to actual parameters 3, 2, 1, 0 during evaluation

Type name `bool`

Values `false`, `true`

Operator	Type	
<code>not</code>	<code>bool -&gt; bool</code>	negation

<code>not true = false</code> <code>not false = true</code>
--

Expressions

`e1 && e2`

“conjunction  $e_1 \wedge e_2$ ”

`e1 || e2`

“disjunction  $e_1 \vee e_2$ ”

— are lazily evaluated, e.g.

<code>1 &lt; 2    5 / 0 = 1</code> <code>↗ true</code>
---

Precedence: `&&` has higher than `||`

# If-then-else expressions

Form:

if  $b$  then  $e_1$  else  $e_2$

Evaluation rules:

if true then  $e_1$  else  $e_2 \rightsquigarrow e_1$

if false then  $e_1$  else  $e_2 \rightsquigarrow e_2$

Notice:

- Both then and else branches must be present (because it is an expression that gives a value)
- either  $e_1$  or  $e_2$  is evaluated provided that (but never both) provided that  $b$  terminates.

Type name `string`

Values `"abcd"`, `" "`, `" "`, `"123\"321"` (escape sequence for `"`)

Operator	Type	
<code>String.length</code>	<code>string -&gt; int</code>	length of string
<code>+</code>	<code>string*string -&gt; string</code>	concatenation
<code>= &lt; &lt;= ...</code>	<code>string*string -&gt; bool</code>	comparisons
<code>string</code>	<code>obj -&gt; string</code>	conversions

## Examples

```
- "auto" < "car";  
> val it = true : bool  
  
- "abc"+"de";  
> val it = "abcde": string
```

```
- String.length("abc"^"def");  
> val it = 6 : int  
  
- string(6+18);  
> val it = "24": string
```

A pattern is composed from **identifiers**, **constants** and the **wildcard pattern** `_` using **constructors** (considered soon)

Examples of patterns are: `3.1`, `true`, `n`, `x`, `5`, `_`

- A pattern may match a value, and if so it results in an environment with bindings for every identifier in the pattern.
- The wildcard pattern – matches any value (resulting in no binding)

Examples:

- The value `3.1` matches pattern `x` resulting in environment:  
`[x ↦ 3.1]`
- The value `true` matches pattern `true` resulting in the empty environment `[]`

# Match expressions

A match expression  $e_m$  has the following form:

$$\begin{array}{l} \text{match } e \text{ with} \\ | pat_1 \rightarrow e_1 \\ \vdots \\ | pat_n \rightarrow e_n \end{array}$$

A match expression  $e_m$  is evaluated as follows:

If

- $v$  is the value of  $e$  and
- $pat_i$  is the first matching pattern for  $v$  and  $env$  is the environment obtained from the pattern matching,

then

$$e_m \rightsquigarrow (e_i, env)$$

# Match expressions: An example

Function declaration:

```
let rec fact n =
  match n with
  | 0 -> 1                                (* i *)
  | n -> n * fact(n-1)                  (* ii *)
val fact : int -> int
```

Evaluation:

```
fact(3)
~> 3 * fact(3 - 1)      (ii) [n ↦ 3]
~> 3 * 2 * fact(2 - 1)  (ii) [n ↦ 2]
~> 3 * 2 * 1 * fact(1 - 1) (ii) [n ↦ 1]
~> 3 * 2 * 1 * 1       (i)  [n ↦ 0]
~> 6
```

A match with a **when** clause and an **exception**:

```
let rec fact n =
  match n with
  | 0 -> 1
  | n when n > 0 -> n * fact(n-1)
  | _ -> failwith "Negative argument"
```

Recursion. Example  $x^n = x \cdot \dots \cdot x$ ,  $n$  occurrences of  $x$

Mathematical definition:

recursion formula

$$x^0 = 1 \quad (1)$$

$$x^n = x \cdot x^{n-1}, \quad \text{for } n > 0 \quad (2)$$

Function declaration:

```
let rec power(x,n) =
  match (x,n) with
  | (_,0) -> 1.0                (* 1 *)
  | (x,n) -> x * power(x,n-1)  (* 2 *)
```

Patterns:

$(_, 0)$  matches any **pair** of the form  $(u, 0)$ .

$(x, n)$  matches any pair  $(u, i)$  **yielding** the bindings

$x \mapsto u, n \mapsto i$



Evaluation. Example: `power(4.0, 2)`

## Function declaration:

```

let rec power(x,n) =
  match (x,n) with
  | (_,0) -> 1.0                (* 1 *)
  | (x,n) -> x * power(x,n-1)   (* 2 *)

```

## Evaluation:

<code>power(4.0,2)</code>	
$\rightsquigarrow$ <code>4.0 * power(4.0, 2 - 1)</code>	Clause 2, $[x \mapsto 4.0, n \mapsto 2]$
$\rightsquigarrow$ <code>4.0 * power(4.0, 1)</code>	
$\rightsquigarrow$ <code>4.0 * (4.0 * power(4.0, 1 - 1))</code>	Clause 2, $[x \mapsto 4.0, n \mapsto 1]$
$\rightsquigarrow$ <code>4.0 * (4.0 * power(4.0, 0))</code>	
$\rightsquigarrow$ <code>4.0 * (4.0 * 1)</code>	Clause 1
$\rightsquigarrow$ <code>16.0</code>	

Types — every expression has a type  $e : \tau$ 

Basic types:

	type name	example of values
Integers	int	~27, 0, 15, 21000
Floats	float	~27.3, 0.0, 48.21
Booleans	bool	true, false

Pairs:

If  $e_1 : \tau_1$  and  $e_2 : \tau_2$ then  $(e_1, e_2) : \tau_1 * \tau_2$ 

pair (tuple) type constructor

Functions:

if  $f : \tau_1 \rightarrow \tau_2$  and  $a : \tau_1$ then  $f(a) : \tau_2$ 

function type constructor

Examples:

```
(4.0, 2): float*int
power: float*int -> float
power(4.0, 2): float
```

\* has higher precedence than  $\rightarrow$

Type inference: `power`

```
let rec power (x,n) =
  match (x,n) with
  | (_,0) -> 1.0                (* 1 *)
  | (x,n) -> x * power(x,n-1)  (* 2 *)
```

- The type of the function must have the form:  $\tau_1 * \tau_2 \rightarrow \tau_3$ , because argument is a pair.
- $\tau_3 = \text{float}$  because `1.0:float` (Clause 1, function value.)
- $\tau_2 = \text{int}$  because `0:int`.
- `x*power(x,n-1):float`, because  $\tau_3 = \text{float}$ .
- multiplication can have  
`int*int -> int` or `float*float -> float`  
 as types, but no “mixture” of `int` and `float`
- Therefore `x:float` and  $\tau_1 = \text{float}$ .

The F# system determines the type `float*int -> float`

# Functions expressions

We have in previous examples exploited pattern matching in match expressions  $e_m$ :

```
match  $e$  with  
  |  $pat_1$  →  $e_1$   
    ⋮  
  |  $pat_n$  →  $e_n$ 
```

A function expression  $e_f$  has a similar pattern matching feature:

```
function  
  |  $pat_1$  →  $e_1$   
    ⋮  
  |  $pat_n$  →  $e_n$ 
```

- A function expression  $e_f$  denotes an **anonymous function**
- An application  $e_f e$  is equivalent to the match expression  $e_m$

An anonymous function computing the number of days in a month:

```
function
| 2  -> 28    // February
| 4  -> 30    // April
| 6  -> 30    // June
| 9  -> 30    // September
| 11 -> 30    // November
| _  -> 31;; // All other months
  val it : int -> int = <fun:clo@17-2>

it 2;;
val it : int = 28
```

An anonymous circle-area function:

```
function r -> System.Math.PI * r * r
```

## Alternative declarations of the power function:

```
let rec power = function
  | (_, 0) -> 1.0
  | (x, n) -> x * power(x, n-1);;
```

```
let rec power a = match a with
  | (_, 0) -> 1.0
  | (x, n) -> x * power(x, n-1);;
```

```
let rec power(x, n) = match n with
  | 0 -> 1.0
  | n' -> x * power(x, n'-1);;
```

A **curried** version:

what is the type?

```
let rec power x = function
  | 0 -> 1.0
  | n when n < 0 -> failwith "power: negative exponent"
  | n -> x * power x (n-1);;
```

- The interactive environment
- Values, expressions, types, patterns
- Declarations of values and recursive functions
- Binding, environment and evaluation
- Type inference

Breathe first round through many concepts aiming at program construction from the first day.

We will go deeper into each of the concepts later in the course.

- Lists: values and constructors
- Recursions following the structure of lists
- Polymorphism

The purpose of this lecture is to give you an (as short as possible) introduction to lists, so that you can solve a problem which can illustrate some of F#'s high-level features.

This part is *not* intended as a comprehensive presentation on lists, and we will return to the topic again later.



A list is a finite sequence of elements having the same type:

$[v_1; \dots; v_n]$  ( $[]$  is called the empty list)

```
[2;3;6];;
val it : int list = [2; 3; 6]

["a"; "ab"; "abc"; ""];;
val it : string list = ["a"; "ab"; "abc"; ""]

[sin; cos];;
val it : (float->float) list = [<fun:...>; <fun:...>]

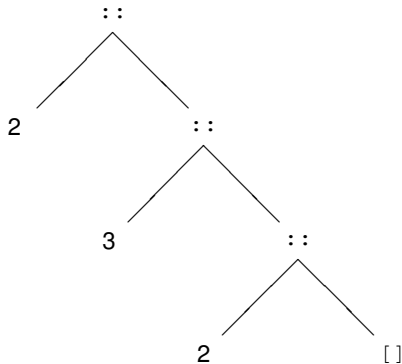
[(1,true); (3,true)];;
val it : (int * bool) list = [(1, true); (3, true)]

[[]; [1]; [1;2]];;
val it : int list list = [[]; [1]; [1; 2]]
```

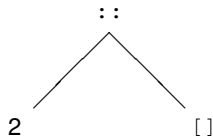
# Trees for lists

A non-empty list  $[x_1, x_2, \dots, x_n]$ ,  $n \geq 1$ , consists of

- a *head*  $x_1$  and
- a *tail*  $[x_2, \dots, x_n]$



Graph for  $[2, 3, 2]$



Graph for  $[2]$

# Recursion on lists – a simple example

$$\text{suml } [x_1; x_2; \dots; x_n] = \sum_{i=1}^n x_i = x_1 + x_2 + \dots + x_n = x_1 + \sum_{i=2}^n x_i$$

Constructors are used in list patterns

```
let rec suml xs =
  match xs with
  | []          -> 0
  | x::tail    -> x + suml tail;;
val suml : int list -> int
```

```
suml [1;2]
~> 1 + suml [2]      (x ↦ 1 and tail ↦ [2])
~> 1 + (2 + suml []) (x ↦ 2 and tail ↦ [])
~> 1 + (2 + 0)       (the pattern [] matches the value [])
~> 1 + 2
~> 3
```

Recursion follows the structure of lists

# The Polymorphic Append function

The built-in function `append @` joins two lists:

$$[x_1; x_2; \dots; x_m] @ [y_1; y_2; \dots; y_n] \\ = [x_1; x_2; \dots; x_m; y_1; y_2; \dots; y_n]$$

It has a **polymorphic type**

```
val (@) : 'a list -> 'a list -> 'a list
```

- `'a` is a *type variable*

Append has **many forms**

`'a = int`: Appending integer lists

```
[1;2] @ [3;4];;
val it : int list = [1;2;3;4]
```

`'a = int list`: Appending lists of integer list

```
[[1];[2;3]] @ [[4]];;
val it : int list list = [[1]; [2; 3]; [4]]
```

## Another polymorphic list function

The function `remove(y,xs)` gives the list obtained from `xs` by deleting every occurrence of `y`, e.g. `remove(2,[1;2;0;2;7]) = [1;0;7]`.

Recursion is following the structure of the list:

```
let rec remove(y,xs) =  
  match xs with  
  | []                -> []  
  | x::tail when x=y -> remove(y,tail)  
  | x::tail           -> x::remove(y,tail);;
```

List elements can be of **any type** that supports **equality**

```
remove : 'a * 'a list -> 'a list when 'a : equality
```

For example

```
remove("a", [""; "a"; "ab"; "a"; "bc"]);;  
val it : string list = [""; "ab"; "bc"]
```

## Exploiting structured patterns: the `isPrefix` function

The function `isPrefix(xs, ys)` tests whether the list `xs` is a prefix of the list `ys`, for example:

```
isPrefix([1;2;3],[1;2;3;8;9]) = true
isPrefix([1;2;3],[1;2;8;3;9]) = false
```

The function is declared as follows:

```
let rec isPrefix(xs, ys) =
  match (xs,ys) with
  | ([],_)          -> true
  | (_,[])          -> false
  | (x::xtail,y::ytail) -> x=y && isPrefix(xtail, ytail);;

isPrefix([1;2;3], [1;2]);;
val it : bool = false
```

A each clause expresses succinctly a natural property:

- The empty list is a prefix of any list
- A non-empty list is not a prefix of the empty list
- A non-empty list (...) is a prefix of another non-empty list (...) if ...

- Lists
- Polymorphism
- Constructors (`::` and `[]` for lists)
- Patterns
- Recursion on the structure of lists
- Constructors used in **patterns** to **decompose** structured values
- Constructors used in **expressions** to **compose** structured values

## Blackboard exercises

- `memberOf(x, ys)` is true iff `x` occurs in the list `ys`
- `insert(x, ys)` is the *ordered list* obtained from the *ordered list* `ys` by insertion of `x`
- `sort(xs)` gives a ordered version of `xs`