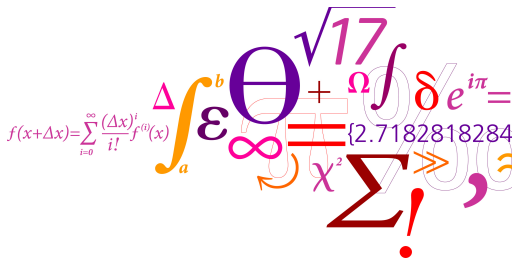


# 02157 Functional Programming

## 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:**

Jonas Dahl Larsen

Mathias Spezia

Mikael Hjermitslev Hoffmann

Oliwia Pindel

Shuokai Ma

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

## Advanced Engineering Mathematics 1

- eNotes: <https://01006.compute.dtu.dk/enoter>

## Advanced Engineering Mathematics 1

- eNotes: <https://01006.compute.dtu.dk/enoter>

For a function, like

$$f(x) = x^2$$

we often mention its **domain** and **range**:

$$f : \mathbb{R} \rightarrow \mathbb{R}$$

## Advanced Engineering Mathematics 1

- eNotes: <https://01006.compute.dtu.dk/enoter>

For a function, like

$$f(x) = x^2$$

we often mention its **domain** and **range**:

$$f : \mathbb{R} \rightarrow \mathbb{R}$$

For a **typed functional language** like F#, a function like:

```
let f x = x ** 2.0;;
```

has an associated type:

```
f:float -> float
```

where **float** is the type of both the domain and the range.

# A Simple Functional Programming Setting

A program  $f$  is a function

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

that takes **one argument** and produces **one result**.

# A Simple Functional Programming Setting

A program  $f$  is a function

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

that takes **one argument** and produces **one result**.

Consider

```
let f x = 2*x + 3;;
```

# A Simple Functional Programming Setting

A program  $f$  is a function

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

that takes **one argument** and produces **one result**.

Consider

```
let f x = 2*x + 3;;
```

Every function has a **type** specifying types of argument and result:

```
f: int -> int
```

- argument and result of  $f$  have type `int` (for integers).



# A Simple Functional Programming Setting

A program  $f$  is a function

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

that takes **one argument** and produces **one result**.

Consider

```
let f x = 2*x + 3;;
```

Every function has a **type** specifying types of argument and result:

```
f: int -> int
```

- argument and result of  $f$  have type `int` (for integers).

**Computation** is governed by **function application**

	$f(1 + 2)$	
=	$f(3)$	evaluate argument
=	$2 * 3 + 3$	substitute <b>3</b> in for $x$ in $f$ 's body
=	<b>9</b>	

# A Simple Functional Programming Setting

A program  $f$  is a function

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

that takes **one argument** and produces **one result**.

Consider

```
let f x = 2*x + 3;;
```

Every function has a **type** specifying types of argument and result:

```
f: int -> int
```

- argument and result of  $f$  have type `int` (for integers).

**Computation** is governed by **function application**

```
f(1 + 2)
= f(3)      evaluate argument
= 2 * 3 + 3 substitute 3 in for x in f's body
= 9
```

F# has **eager** evaluation: Compute argument before making the call

## Prerequisites

- You have used an editor to create programs
- You have installed a program on your laptop
- You have had (or have in the same semester) a course on Discrete Mathematics

## Prerequisites

- You have used an editor to create programs
- You have installed a program on your laptop
- You have had (or have in the same semester) a course on Discrete Mathematics

The course is a part of educations leading to the MSc programme in Computer Science and Engineering.

- candidates contributing to the development of high-quality, advanced software products

## Prerequisites

- You have used an editor to create programs
- You have installed a program on your laptop
- You have had (or have in the same semester) a course on Discrete Mathematics

The course is a part of educations leading to the MSc programme in Computer Science and Engineering.

- candidates contributing to the development of high-quality, advanced software products

It is an aim to contribute to the fundament for educations leading to the MSc education in CS&E

## Prerequisites

- You have used an editor to create programs
- You have installed a program on your laptop
- You have had (or have in the same semester) a course on Discrete Mathematics

The course is a part of educations leading to the MSc programme in Computer Science and Engineering.

- candidates contributing to the development of high-quality, advanced software products

It is an aim to contribute to the fundament for educations leading to the MSc education in CS&E

May sound good; but what does it mean?

# There is no magic

It is possible to understand everything:

# There is no magic

It is possible to understand everything:

- The syntax (notation) of the programming language



# There is no magic

It is possible to understand everything:

- The syntax (notation) of the programming language
- The semantics (meaning) of programs

# There is no magic

It is possible to understand everything:

- The syntax (notation) of the programming language
- The semantics (meaning) of programs
- The evaluation of programs

# There is no magic

It is possible to understand everything:

- The syntax (notation) of the programming language
- The semantics (meaning) of programs
- The evaluation of programs
- The properties of programs

# There is no magic

It is possible to understand everything:

- The syntax (notation) of the programming language
- The semantics (meaning) of programs
- The evaluation of programs
- The properties of programs

Functional programming is a simple setting supporting

- declaration of clear, concise programs at a high level of abstraction
- understanding and analysis of programs

due to the basis on mathematical functions (no side-effects)

An archetypical example  $n! = 1 \cdot 2 \cdot \dots \cdot n$ ,  $n \geq 0$

Mathematical definition:

recursion formula

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

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

An archetypical 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:

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

Declaring recursive functions: `let rec f x = e`

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

Declaring recursive functions: `let rec f x = e`

- 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
```



## Declaring recursive functions: `let rec f x = e`

- 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)
~> 3 * 2 * 1 * fact(1 - 1)   (ii)
~> 3 * 2 * 1 * 1             (i)
~> 6
```

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

## Declaring recursive functions: `let rec f x = e`

- 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)
~> 3 * 2 * 1 * 1        (i)
~> 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

## Declaring recursive functions: `let rec f x = e`

- 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)
~> 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

## Declaring recursive functions: `let rec f x = e`

- 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

# Some functional programming background

- The  $\lambda$ -calculus was introduced 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  $\lambda x.x + 2$ .

## Some functional programming background

- The  $\lambda$ -calculus was introduced 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  $\lambda x.x + 2$ .
- The untyped functional-like programming language LISP was developed by McCarthy in the late 1950s.

## Some functional programming background

- The  $\lambda$ -calculus was introduced 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  $\lambda x.x + 2$ .
- The untyped functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) were introduced in the 1970s.

## Some functional programming background

- The  $\lambda$ -calculus was introduced 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  $\lambda x.x + 2$ .
- The untyped functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) were introduced in the 1970s.
- Functional languages (SML, Haskell, OCAML, F#, ...) have now applications far away from their origin: Compilers, Artificial Intelligence, Web-applications, Financial sector, ...



## Some functional programming background

- The  $\lambda$ -calculus was introduced 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  $\lambda x.x + 2$ .
- The untyped functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) were introduced 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"

# Some functional programming background

- The  $\lambda$ -calculus was introduced 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  $\lambda x.x + 2$ .
- The untyped functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) were introduced 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"
- Functional programming should be a mandatory element of every BSc. education in Computer Science according to ACM's and IEEE's curricula recommendations, 2013.

The untyped Lambda Calculus has just three kinds of expressions  $e$ :

- variables  $x$
- abstractions  $\lambda x.e$
- applications  $e_1 e_2$

The untyped Lambda Calculus has just three kinds of expressions  $e$ :

- variables  $x$
- abstractions  $\lambda x.e$
- applications  $e_1 e_2$

where

- $\lambda x.e$  reads: “the function of  $x$  given by  $e$ ”

The untyped Lambda Calculus has just three kinds of expressions  $e$ :

- variables  $x$
- abstractions  $\lambda x.e$
- applications  $e_1 e_2$

where

- $\lambda x.e$  reads: “the function of  $x$  given by  $e$ ”

An application like  $(\lambda x.e) e_2$  may be **evaluated** as follows:

$$(\lambda x.e) e_2 \rightsquigarrow e'_2$$

where  $e'_2$  is obtained from  $e_2$  by

- substituting every free occurrence of  $x$  in  $e$  by  $e_2$

like we did in the previous examples

The untyped Lambda Calculus has just three kinds of expressions  $e$ :

- variables  $x$
- abstractions  $\lambda x.e$
- applications  $e_1 e_2$

where

- $\lambda x.e$  reads: “the function of  $x$  given by  $e$ ”

An application like  $(\lambda x.e) e_2$  may be **evaluated** as follows:

$$(\lambda x.e) e_2 \rightsquigarrow e'_2$$

where  $e'_2$  is obtained from  $e_2$  by

- substituting every free occurrence of  $x$  in  $e$  by  $e_2$

like we did in the previous examples

No magic: A full explanation can be given in terms of few concepts

The untyped Lambda Calculus has just three kinds of expressions  $e$ :

- variables  $x$
- abstractions  $\lambda x.e$
- applications  $e_1 e_2$

where

- $\lambda x.e$  reads: “the function of  $x$  given by  $e$ ”

An application like  $(\lambda x.e) e_2$  may be **evaluated** as follows:

$$(\lambda x.e) e_2 \rightsquigarrow e'_2$$

where  $e'_2$  is obtained from  $e_2$  by

- substituting every free occurrence of  $x$  in  $e$  by  $e_2$

like we did in the previous examples

No magic: A full explanation can be given in terms of few concepts

The part of **F#** we will use is based on **typed lambda calculus**

# Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...



# Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...

- Patterns:

$x$  -  $(p_1, \dots, p_n)$   $p_1 :: p_2$   $p_1 | p_2$   $p \text{ when } e$   $p \text{ as } x$   $p : t \dots$

# Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...

- Patterns:

$x$  -  $(p_1, \dots, p_n)$   $p_1 :: p_2$   $p_1 | p_2$   $p$  when  $e$   $p$  as  $x$   $p:t...$

- Expressions:

$x$   $(e_1, \dots, e_n)$   $e_1 :: e_2$   $e_1 e_2$   $e_1 \oplus e_2$  let  $p_1 = e_1$  in  $e_2$   $e:t$

if  $e$  then  $e_1$  then  $e_2$  match  $e$  with *clauses*

fun  $p_1 \dots p_n \rightarrow e$  function *clauses* ...

where the construct *clauses* has the form:

|  $p_1 \rightarrow e_1$  | ... |  $p_n \rightarrow e_n$

# Overview: Syntactical constructs in “our part of” F#

- Constants:  $0$ ,  $1$ ,  $1$ ,  $true$ , ...

- Patterns:

$x$  -  $(p_1, \dots, p_n)$   $p_1 :: p_2$   $p_1 | p_2$   $p$  when  $e$   $p$  as  $x$   $p : t \dots$

- Expressions:

$x$   $(e_1, \dots, e_n)$   $e_1 :: e_2$   $e_1 e_2$   $e_1 \oplus e_2$   $\text{let } p_1 = e_1 \text{ in } e_2$   $e : t$

$\text{if } e \text{ then } e_1 \text{ then } e_2$   $\text{match } e \text{ with } \textit{clauses}$

$\text{fun } p_1 \dots p_n \rightarrow e$   $\text{function } \textit{clauses}$  ...

- Declarations  $\text{let } f \ p_1 \dots p_n = e$   $\text{let rec } f \ p_1 \dots p_n = e, n \geq 0$

where the construct *clauses* has the form:

$| \ p_1 \rightarrow e_1 \ | \ \dots \ | \ p_n \rightarrow e_n$

# Overview: Syntactical constructs in “our part of” F#

- Constants: `0`, `1.1`, `true`, ...

- Patterns:

`x` - `(p1, ..., pn)` `p1::p2` `p1|p2` `p` when `e` `p` as `x` `p:t...`

- Expressions:

`x` `(e1, ..., en)` `e1::e2` `e1e2` `e1⊕e2` `let p1 = e1 in e2` `e:t`

`if e then e1 then e2` `match e with clauses`

`fun p1 ... pn -> e` `function clauses` ...

- Declarations `let f p1 ... pn = e` `let rec f p1 ... pn = e`,  $n \geq 0$

- Types

`int` `float` `bool` `string` `'a...`

`t1*t2*...*tn` `t list` `t1->t2...`

where the construct *clauses* has the form:

`| p1 -> e1 | ... | pn -> en`

# Overview: Syntactical constructs in “our part of” F#

- Constants: `0`, `1.1`, `true`, ...

- Patterns:

`x` - `(p1, ..., pn)` `p1::p2` `p1|p2` `p` when `e` `p` as `x` `p:t...`

- Expressions:

`x` `(e1, ..., en)` `e1::e2` `e1e2` `e1⊕e2` `let p1 = e1 in e2` `e:t`

`if e then e1 then e2` `match e with clauses`

`fun p1 ... pn -> e` `function clauses` ...

- Declarations `let f p1 ... pn = e` `let rec f p1 ... pn = e, n ≥ 0`

- Types

`int` `float` `bool` `string` `'a...`

`t1*t2*...*tn` `t list` `t1->t2...`

where the construct *clauses* has the form:

`| p1 -> e1 | ... | pn -> en`

In addition to that

- type declarations, precedence and associativity rules, parenthesis around *p* and *e* and type correctness

Have a look at

- <http://homepages.inf.ed.ac.uk/wadler/realworld/>
- <https://fsharp.org/testimonials/>

concerning use of functional programming in the "real world".

- General information:

<http://courses.compute.dtu.dk/02157>

- General information:

<http://courses.compute.dtu.dk/02157>

- Practical Information:

[http://courses.compute.dtu.dk/02157/  
PracticalInfo.html](http://courses.compute.dtu.dk/02157/PracticalInfo.html)

**Exam form: Written exam, 4 hour – no aid allowed**



- General information:

<http://courses.compute.dtu.dk/02157>

- Practical Information:

[http://courses.compute.dtu.dk/02157/  
PracticalInfo.html](http://courses.compute.dtu.dk/02157/PracticalInfo.html)

**Exam form: Written exam, 4 hour – no aid allowed**

- Course plan:

<http://courses.compute.dtu.dk/02157/plan.html>

- General information:

<http://courses.compute.dtu.dk/02157>

- Practical Information:

[http://courses.compute.dtu.dk/02157/  
PracticalInfo.html](http://courses.compute.dtu.dk/02157/PracticalInfo.html)

**Exam form: Written exam, 4 hour – no aid allowed**

- Course plan:

<http://courses.compute.dtu.dk/02157/plan.html>

On DTU Learn you can find some material

- A brief course introduction
- A mini-project on polynomials
- Slides
- ....

# Course Infrastructure

- Syllabus (see introduction to the course)
- Weekly lectures
- Weekly exercise classes with fantastic TAs

a flipped classroom model

# Course Infrastructure

- Syllabus (see introduction to the course)
- Weekly lectures
- Weekly exercise classes with fantastic TAs

a flipped classroom model

Course design is based on an evenly distributed workload and “steady progress” throughout the semester

# Course Infrastructure

- Syllabus (see introduction to the course)
- Weekly lectures
- Weekly exercise classes with fantastic TAs

a flipped classroom model

Course design is based on an evenly distributed workload and “steady progress” throughout the semester

Mini-projects: Exercise FP concepts and techniques while

- telling a coherent story on a specific topic
- relating FP to neighbouring courses
- introducing fundamental CS concepts

# Course Infrastructure

- Syllabus (see introduction to the course)
- Weekly lectures
- Weekly exercise classes with fantastic TAs

a flipped classroom model

Course design is based on an evenly distributed workload and “steady progress” throughout the semester

Mini-projects: Exercise FP concepts and techniques while

- telling a coherent story on a specific topic
- relating FP to neighbouring courses
- introducing fundamental CS concepts

Nothing is mandatory

# Course Infrastructure

- Syllabus (see introduction to the course)
- Weekly lectures
- Weekly exercise classes with fantastic TAs

a flipped classroom model

Course design is based on an evenly distributed workload and “steady progress” throughout the semester

Mini-projects: Exercise FP concepts and techniques while

- telling a coherent story on a specific topic
- relating FP to neighbouring courses
- introducing fundamental CS concepts

Nothing is mandatory

It is your own responsibility to achieve a good use of Fridays’ teaching slot

- no online support
- no hotline support

# Course Infrastructure

- Syllabus (see introduction to the course)
- Weekly lectures
- Weekly exercise classes with fantastic TAs

a flipped classroom model

Course design is based on an evenly distributed workload and “steady progress” throughout the semester

Mini-projects: Exercise FP concepts and techniques while

- telling a coherent story on a specific topic
- relating FP to neighbouring courses
- introducing fundamental CS concepts

Nothing is mandatory

It is your own responsibility to achieve a good use of Fridays' teaching slot

- no online support
- no hotline support

You are always welcome to visit my office: Room 112, Building 322



## 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 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

```
2*3 + 4;;  
val it : int = 10
```

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

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

⇐ Input to the F# system

⇐ Answer from the F# system

```
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

```
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`

# Value Declarations

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 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

## 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
```

## 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
```

## 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*

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 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)

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:

- Value `3.1` matches pattern `x` resulting in environment:  $[x \mapsto 3.1]$



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:

- Value `3.1` matches pattern `x` resulting in environment:  $[x \mapsto 3.1]$
- Value `true` matches pattern `true` resulting in environment  $[]$

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:

- Value `3.1` matches pattern `x` resulting in environment:  $[x \mapsto 3.1]$
- Value `true` matches pattern `true` resulting in environment  $[]$
- The **pair** `(1, true)` matches pattern `(x, y)` resulting in environment  $[x \mapsto 1, y \mapsto true]$

# Match expressions

A match expression  $e_m$  has the following form:

```
match  $e$  with  
|  $pat_1 \rightarrow e_1$   
   $\vdots$   
|  $pat_n \rightarrow e_n$ 
```

# Match expressions

A match expression  $e_m$  has the following form:

```
match  $e$  with  
|  $pat_1 \rightarrow e_1$   
   $\vdots$   
|  $pat_n \rightarrow e_n$ 
```

A match expression  $e_m$  is evaluated as follows:

- 1 evaluate  $e$  to a value, say  $v$
- 2 search for the first pattern  $pat_i$  matching  $v$
- 3 evaluate  $e_i$  in an environment enriched with the bindings from the pattern matching

# Match expressions

```
match  $e$  with  
|  $pat_1 \rightarrow e_1$   
   $\vdots$   
|  $pat_n \rightarrow e_n$ 
```

A match expression  $e_m$  has the following form:

A match expression  $e_m$  is evaluated as follows:

- 1 evaluate  $e$  to a value, say  $v$
- 2 search for the first pattern  $pat_i$  matching  $v$
- 3 evaluate  $e_i$  in an environment enriched with the bindings from the pattern matching

If no pattern matches  $v$ , then the evaluation terminates abnormally.

## Example: Match on a pair

Let  $e_1$  be given by:

```
match (3+5, 3<5) with
| (0, _)    -> 0
| (n, false) -> -n
| (n, _)    -> 2*n
```

## Example: Match on a pair

Let  $e_1$  be given by:

```
match (3+5, 3<5) with
| (0, _)    -> 0
| (n, false) -> -n
| (n, _)    -> 2*n
```

Evaluation:

```
       $e_1$ 
  ~> (2 * n, [n ↦ 8])
  ~> (2 * 8, [n ↦ 8])
  ~> 16
```

## Example: Match expression in a declaration

Function declaration:

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



## Example: Match expression in a declaration

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)
~> 3 * 2 * fact(2 - 1)  (ii)
~> 3 * 2 * 1 * fact(1 - 1) (ii)
~> 3 * 2 * 1 * 1        (i)
~> 6
```

## Example: Match expression in a declaration

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)
~> 3 * 2 * 1 * fact(1 - 1) (ii)
~> 3 * 2 * 1 * 1        (i)
~> 6
```

# Example: Match expression in a declaration

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)
~> 3 * 2 * 1 * 1        (i)
~> 6
```

## Example: Match expression in a declaration

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)
~> 6
```

# Example: Match expression in a declaration

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
```

## Example: Match expression in a declaration

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:

$$x^0 = 1$$

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

recursion formula

(1)

(2)

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

Mathematical definition:

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

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

recursion formula

Function declaration:

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



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

$$x \mapsto u, n \mapsto i$$

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

$$x \mapsto u, n \mapsto i$$

Can you simplify the program?

# 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	<code>int</code>	~27, 0, 15, 21000
Floats	<code>float</code>	~27.3, 0.0, 48.21
Booleans	<code>bool</code>	true, false

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

Basic types:

	type name	example of values
Integers	<code>int</code>	~27, 0, 15, 21000
Floats	<code>float</code>	~27.3, 0.0, 48.21
Booleans	<code>bool</code>	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

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

Basic types:

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

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

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 *)
```



# 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.

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.)

# 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`.

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  $\tau_3 = \text{float}$ .

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`

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  $\tau_1 = \text{float}$ .

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`

## A higher-order version of the power function

We shall now look at a version of `power`  $x \ n = x^n$  with the type

```
power: float -> (int -> float)
```



## A higher-order version of the power function

We shall now look at a version of `power`  $x^n = x^n$  with the type

```
power: float -> (int -> float)
```

- the argument of `power` is the base `x`

## A higher-order version of the power function

We shall now look at a version of `power`  $x \ n = x^n$  with the type

```
power: float -> (int -> float)
```

- the argument of `power` is the base  $x$
- and `power`  $x$  is the function that maps exponent  $n$  to  $x^n$

## A higher-order version of the power function

We shall now look at a version of `power`  $x\ n = x^n$  with the type

```
power: float -> (int -> float)
```

- the argument of `power` is the base  $x$
- and `power  $x$`  is the function that maps exponent  $n$  to  $x^n$

The function may be evaluated in *stages*:

```
let pow2 = power 2.0;;
```

```
pow2 3;;
```

```
val it : float = 8.0
```

```
pow2 4;;
```

```
val it : float = 16.0
```

## A higher-order version of the power function

We shall now look at a version of `power`  $x\ n = x^n$  with the type

```
power: float -> (int -> float)
```

- the argument of `power` is the base  $x$
- and `power`  $x$  is the function that maps exponent  $n$  to  $x^n$

The function may be evaluated in *stages*:

```
let pow2 = power 2.0;;
```

```
pow2 3;;
```

```
val it : float = 8.0
```

```
pow2 4;;
```

```
val it : float = 16.0
```

This higher-order version of `power` is declared by

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

## A higher-order version of the power function

We shall now look at a version of `power x n = xn` with the type

```
power: float -> (int -> float)
```

- the argument of power is the base  $x$
- and `power x` is the function that maps exponent  $n$  to  $x^n$

The function may be evaluated in *stages*:

```
let pow2 = power 2.0;;
```

```
pow2 3;;
```

```
val it : float = 8.0
```

```
pow2 4;;
```

```
val it : float = 16.0
```

This higher-order version of power is declared by

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

The value of the function is a function

# A expression for anonymous functions

The function expression

```
function  
|  $pat_1 \rightarrow e_1$   
|  $\vdots$   
|  $pat_n \rightarrow e_n$ 
```

allows you to “tabulate” argument-value pairs of a function.

```
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>
```

# A expression for anonymous functions

The function expression

```
function
|  $pat_1 \rightarrow e_1$ 
|
|  $pat_n \rightarrow e_n$ 
```

allows you to “tabulate” argument-value pairs of a function.

```
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
```

## Another higher-order version of the power function

We now have another look at `power`  $x \ n = x^n$  with the type

```
power: float -> (int -> float)
```



## Another higher-order version of the power function

We now have another look at `power x n = xn` with the type

```
power: float -> (int -> float)
```

The following declaration explicitly reveals that `power x` is a function:

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

Type name `bool`

Values `false`, `true`

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

```
not true = false  
not false = true
```

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  $e_1 \vee e_2$ ”

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 `||`

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

- Lists: values and constructors
  - Recursions following the structure of lists
  - Polymorphism
- 
- The list concept is a natural, built-in ingredient of functional languages

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

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

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 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"; ""]
```

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:...>]
```

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)]
```

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]]
```

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

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

# List constructors

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

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

The list type has two constructors:

- The empty list  $[]$
  - The *cons* constructor  $x_1 :: [x_2; \dots; x_n] = [x_1; x_2; \dots; x_n]$
- they are used to *construct* and to *decompose* lists

## Recursion on lists – a simple example

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

## 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
```



# 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

## A polymorphic list function (I)

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;;
```

## A polymorphic list function (I)

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
```

## A **polymorphic** list function (I)

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
```

- `'a` is a *type variable*
- `'a : equality` is a type constraint

## A **polymorphic** list function (I)

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
```

- `'a` is a *type variable*
- `'a : equality` is a type constraint

The F# system infers the **most general type** for `remove`

## A polymorphic list function (II)

- A type containing type variables is called a **polymorphic type**
- The remove function is called a **polymorphic function**.

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

The function has **many forms**, one for each instantiation of 'a:

## A polymorphic list function (II)

- A type containing type variables is called a **polymorphic type**
- The remove function is called a **polymorphic function**.

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

The function has **many forms**, one for each instantiation of 'a:

Instantiating 'a with `int`:

```
remove 2 [1; 2; 0; 2; 7];;  
val it : int list = [1; 0; 7]
```

## A polymorphic list function (II)

- A type containing type variables is called a **polymorphic type**
- The remove function is called a **polymorphic function**.

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

The function has **many forms**, one for each instantiation of 'a:

Instantiating 'a with int:

```
remove 2 [1; 2; 0; 2; 7];;  
val it : int list = [1; 0; 7]
```

Instantiating 'a with int list:

```
remove [2] [[2;1]; [2]; [0;1]; [2]; [5;6;7]];;  
val it : int list list = [[2; 1]; [0; 1]; [5; 6; 7]]
```



## A polymorphic list function (II)

- A type containing type variables is called a **polymorphic type**
- The remove function is called a **polymorphic function**.

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

The function has **many forms**, one for each instantiation of 'a:

Instantiating 'a with int:

```
remove 2 [1; 2; 0; 2; 7];;
val it : int list = [1; 0; 7]
```

Instantiating 'a with int list:

```
remove [2] [[2;1]; [2]; [0;1]; [2]; [5;6;7]];;
val it : int list list = [[2; 1]; [0; 1]; [5; 6; 7]]
```

Notice that  $\rightarrow$  **associates to the right**:

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

## 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
```

## 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
```

## 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:

## 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

## 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

## 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



- Lists
- Polymorphism

- Lists
- Polymorphism
- Constructors (`::` and `[]` for lists)

- Lists
- Polymorphism
- Constructors (`::` and `[]` for lists)
- Patterns

- Lists
- Polymorphism
- Constructors (`::` and `[]` for lists)
- Patterns
- Recursion on the structure of lists

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

- 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

- 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`