Programmazione Avanzata

Prof. Michele Loreti

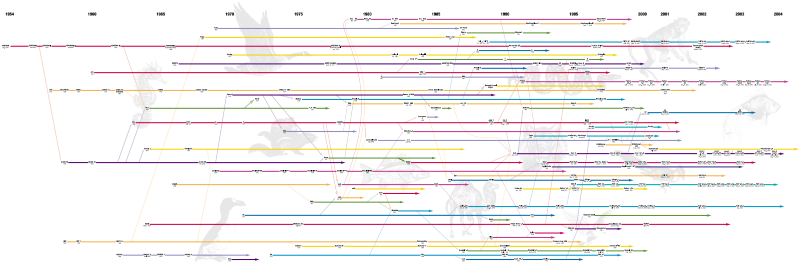
Programmazione Avanzata
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Programming paradigms

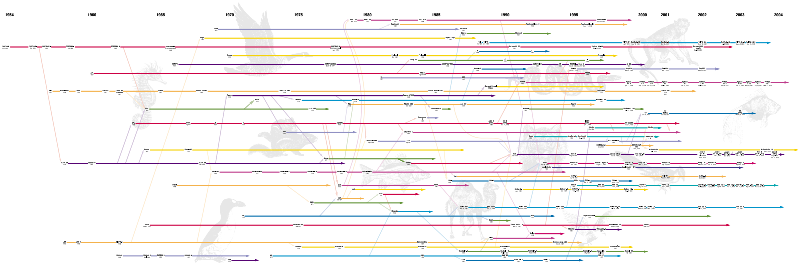
Prof. Michele Loreti

Programmazione Avanzata
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Programming Languages



Programming Languages



How we can classify all these languages?

Programming paradigms

Programming paradigms are a way to classify programming languages based on their features.

Programming paradigms

Programming paradigms are a way to classify programming languages based on their features.

A programming paradigm is an approach to programming a computer based on a mathematical theory or a coherent set of principles.

Programming paradigms

Programming paradigms are a way to classify programming languages based on their features.

A programming paradigm is an approach to programming a computer based on a mathematical theory or a coherent set of principles.

Each paradigm supports a set of concepts that makes it the best for a certain kind of problem.

Programming paradigms

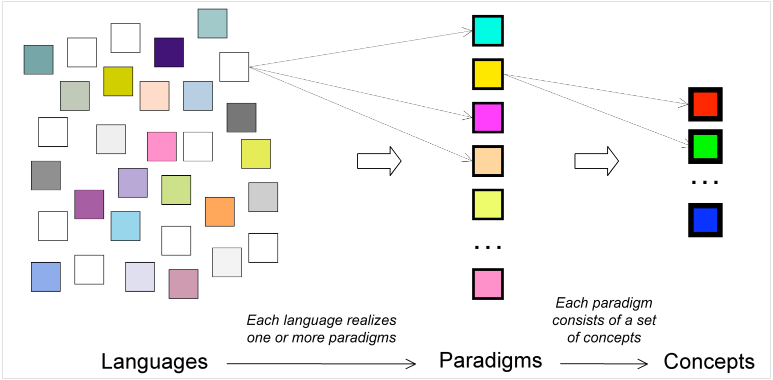
Programming paradigms are a way to classify programming languages based on their features.

A programming paradigm is an approach to programming a computer based on a mathematical theory or a coherent set of principles.

Each paradigm supports a set of concepts that makes it the best for a certain kind of problem.

Solving a programming problem requires choosing the right concepts!

Programming paradigms



Programming paradigms

Common programming paradigms include:

- imperative/procedural: statements are used to change program's state. Imperative programming focuses on describing how a program operates.

Programming paradigms

Common programming paradigms include:

- imperative/procedural: statements are used to change program's state. Imperative programming focuses on describing how a program operates.
- functional: computation is treated as the evaluation of mathematical functions and avoids changing-state and mutable data.

Programming paradigms

Common programming paradigms include:

- imperative/procedural: statements are used to change program's state. Imperative programming focuses on describing how a program operates.
- functional: computation is treated as the evaluation of mathematical functions and avoids changing-state and mutable data.
- declarative/logical: expresses the logic of a computation without describing its control flow. A program consists in a set of sentences in logical form, expressing facts and rules about some problem domain.

Programming paradigms

Common programming paradigms include:

- imperative/procedural: statements are used to change program's state. Imperative programming focuses on describing how a program operates.
- functional: computation is treated as the evaluation of mathematical functions and avoids changing-state and mutable data.
- declarative/logical: expresses the logic of a computation without describing its control flow. A program consists in a set of sentences in logical form, expressing facts and rules about some problem domain.
- object-oriented: it is based on the concept of objects, which may contain both data, the fields, and code, the methods.

This lecture...

In this lecture...

This lecture. . .

In this lecture. . .

. . . we will first introduce basic notions of functional programming. . .

This lecture...

In this lecture...

...we will first introduce basic notions of functional programming...

...then some basic notions of declarative programming is provided...

This lecture. . .

In this lecture. . .

. . . we will first introduce basic notions of functional programming. . .

. . . then some basic notions of declarative programming is provided. . .

. . . after that we focus on object-oriented programming. . .

This lecture. . .

In this lecture. . .

. . . we will first introduce basic notions of functional programming. . .

. . . then some basic notions of declarative programming is provided. . .

. . . after that we focus on object-oriented programming. . .

. . . finally an overview of modern programming languages is provided.

Functional programming in F#: Basic Concepts

Prof. Michele Loreti

Programmazione Avanzata
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Functional programming

Programming in a functional language consists of building definitions end using the computer to evaluate expressions.

Functional programming

Programming in a functional language consists of building definitions end using the computer to evaluate expressions.

Functions are first-class values and can be assigned to names (*variables*).

Functional programming

Programming in a functional language consists of building definitions end using the computer to evaluate expressions.

Functions are first-class values and can be assigned to names (*variables*).

Computations consist in the appropriate compositions of defined functions.

Functional programming

Programming in a functional language consists of building definitions end using the computer to evaluate expressions.

Functions are first-class values and can be assigned to names (*variables*).

Computations consist in the appropriate compositions of defined functions.

We will consider F#, a modern functional language integrated in the .Net framework.

F# programming language

F# (pronounced *F sharp*)...

... is a strongly typed programming language;

F# programming language

F# (pronounced *F sharp*)...

... is a strongly typed programming language;

... supports multi-paradigms (functional, imperative, and object-oriented);

F# programming language

F# (pronounced *F sharp*)...

... is a strongly typed programming language;

... supports multi-paradigms (functional, imperative, and object-oriented);

... is used as a cross-platform Common Language Infrastructure (CLI) language;

F# programming language

F# (pronounced *F sharp*). . .

. . . is a strongly typed programming language;

. . . supports multi-paradigms (functional, imperative, and object-oriented);

. . . is used as a cross-platform Common Language Infrastructure (CLI) language;

. . . can generate JavaScript and Graphics Processing Unit (GPU) code.

F# programming language

F# (pronounced *F sharp*)...

... is a strongly typed programming language;

... supports multi-paradigms (functional, imperative, and object-oriented);

... is used as a cross-platform Common Language Infrastructure (CLI) language;

... can generate JavaScript and Graphics Processing Unit (GPU) code.

F# programming language

F# (pronounced *F sharp*)...

... is a strongly typed programming language;

... supports multi-paradigms (functional, imperative, and object-oriented);

... is used as a cross-platform Common Language Infrastructure (CLI) language;

... can generate JavaScript and Graphics Processing Unit (GPU) code.

Here we will main consider the functional aspects!

F# programming language
Primitive Types (1/2)

- bool, Boolean values (true or false).
- byte, Unsigned byte (from 0 to $2^8 - 1$).
- sbyte, Signed byte (from -2^7 to $2^7 - 1$).
- int16, 16-bit integer (from -2^{15} to $2^{15} - 1$).
- uint16, 16-bit integer (from 0 to $2^{16} - 1$).
- int, 32-bit integer (from -2^{31} to $2^{31} - 1$).
- uint32, 32-bit unsigned (from 0 to $2^{32} - 1$).
- int64, 64-bit integer (from -2^{63} to $2^{63} - 1$).
- uint64, 64-bit unsigned int (from 0 to $2^{64} - 1$).
- char, Unicode character values.
- string, Unicode text.
- decimal, Floating point data type that has at least 28 significant digits.

F# programming language

Primitive Types (2/2)

- unit, Indicates the absence of an actual value.
- void, Indicates no type or value.
- float32, A 32-bit floating point type.
- float, A 64-bit floating point type.

F# programming language
Values (1/2)

- bool: true, false.
- byte, an integer with postfix y (86y).
- sbyte, an integer with postfix uy (86uy).
- int16, an integer with postfix s (86s).
- uint16, an integer with postfix us (86us).
- int, an integer with the optional postfix l (86 or 86l).
- uint32, an integer with postfix u or ul (86u or ul).
- int64, an integer with postfix L (86L).
- uint64, an integer with postfix UL (86UL).
- char, a single symbol surrounded by single quotes ('a').

- string, can be:
  - ... a sequence of characters surrounded by double quotes ("Hello\n\n World!");
  - ... a sequence of characters surrounded by double quotes and prefixed with @ (@"Hello\n\n World!");
  - ... a portion of text (possibly on multiple lines) surrounded by """

```
"""Hello

World!"""
```

- decimal, a floating point value postfixed with M (0.35M).
- unit, the value ().
- float32, a floating point postfixed with f or F (0.35f or 035F).
- float, a floating point in decimal or exponential form (0.35 or 3.5E−1).

F# programming language

Basic concepts

Basic construct in F# is let that can be used to associate a name with a value

```
let num = 10
let str = "F#"
```

F# programming language

Basic concepts

Basic construct in F# is let that can be used to associate a name with a value

```
let num = 10
let str = "F#"
```

Each name has a type that is inferred from the associated expression!

Basic construct in F# is `let` that can be used to associate a name with a value

```
let num = 10
let str = "F#"
```

Each name has a type that is inferred from the associated expression!

Above:

- `num` has type `int`;
- `str` has type `string`;

F# programming language
Operators

Arithmetic Operators: +, −, *, /, %, **;

Comparison Operators: =, <, <=, >, >=, <>;

Boolean Operators: not, ||, &&;

Bitwise Operators: &&&, |||, ^^^, ~~~, <<<, >>>;

F# programming language
Operators

Arithmetic Operators: +, −, *, /, %, **;

Comparison Operators: =, <, <=, >, >=, <>;

Boolean Operators: not, ||, &&;

Bitwise Operators: &&&, |||, ^^^, ~~~, <<<, >>>;

Arithmetic and Comparison operators are overloaded: the exact type depends on the type of their argument!

F# programming language
Operators

Arithmetic Operators: +, −, *, /, %, **;

Comparison Operators: =, <, <=, >, >=, <>;

Boolean Operators: not, ||, &&;

Bitwise Operators: &&&, |||, ^^^, ~~~, <<<, >>>;

Arithmetic and Comparison operators are overloaded: the exact type depends on the type of their argument!

Differently from `Java`, no implicit cast is done!

F# programming language

Simple type errors!

```
let x = 86u //x has type ubyte
let y = 86  //y has type int

let z = x+y //This is an error!!!!
```

Functions are first-class values and can be associated with names as any other built-in types:

Functions are first-class values and can be associated with names as any other built-in types:

```
let f1 = fun x -> x+1

let f2(x) = x+1
```

Functions are first-class values and can be associated with names as any other built-in types:

```
let f1 = fun x -> x+1

let f2(x) = x+1
```

Functions can be passed as arguments of other functions:

```
let f3(x,f) = f(x+2)+1

let y = f3(1,f)
```

F# programming language

Basic concepts

Functions are first-class values and can be associated with names as any other built-in types:

```
let f1 = fun x -> x+1

let f2(x) = x+1
```

Functions can be passed as arguments of other functions:

```
let f3(x,f) = f(x+2)+1

let y = f3(1,f)
```

The type of parameters and the type of returned value can be omitted

Functions are first-class values and can be associated with names as any other built-in types:

```
let f1 = fun x -> x+1

let f2(x) = x+1
```

Functions can be passed as arguments of other functions:

```
let f3(x,f) = f(x+2)+1

let y = f3(1,f)
```

The type of parameters and the type of returned value can be omittedwhen they can be inferred from the code!

F# programming language

Basic concepts

Functions are first-class values and can be associated with names as any other built-in types:

```
let f1 = fun x -> x+1
```

```
let f2(x) = x+1
```

Functions can be passed as arguments of other functions:

```
let f3(x,f) = f(x+2)+1
```

```
let y = f3(1,f)
```

The type of parameters and the type of returned value can be omittedwhen they can be inferred from the code!

Function types have the form: `type1 -> type2`

F# programming language

Type inference

The idea of type inference is that you do not have to specify the types of F# constructs except when the compiler cannot conclusively deduce the type.

F# programming language

Type inference

The idea of type inference is that you do not have to specify the types of F# constructs except when the compiler cannot conclusively deduce the type.

This does not mean that F# is a dynamically typed language or that values in F# are weakly typed.

F# programming language

Type inference

The idea of type inference is that you do not have to specify the types of F# constructs except when the compiler cannot conclusively deduce the type.

This does not mean that F# is a dynamically typed language or that values in F# are weakly typed.

F#, like almost all functional languages, is statically typed!

F# programming language
Type inference

The idea of type inference is that you do not have to specify the types of F# constructs except when the compiler cannot conclusively deduce the type.

This does not mean that F# is a dynamically typed language or that values in F# are weakly typed.

F#, like almost all functional languages, is statically typed!

Type annotations can be used to help the compiler to infer the expected type.

F# programming language
Type inference

```
//No annotation, inferred type: int*int -> int
let f(x,y) = x+y
```

F# programming language
Type inference

```
//No annotation, inferred type: int*int -> int
let f(x,y) = x+y


//Parameter x is annotated as float, inferred type: float*
  float -> float
let f(x: float, y) = x+y
```

F# programming language

Type inference

```
//No annotation, inferred type: int*int -> int
let f(x,y) = x+y

//Parameter x is annotated as float, inferred type: float*
  float -> float
let f(x: float, y) = x+y

//Name x is annotated as float, inferred type: float*float
 -> float
let f(x,y) = (x: float)+y
```

F# programming language
Type inference

```
//No annotation, inferred type: int*int -> int
let f(x,y) = x+y

//Parameter x is annotated as float, inferred type: float*
  float -> float
let f(x: float, y) = x+y

//Name x is annotated as float, inferred type: float*float
 -> float
let f(x,y) = (x: float)+y

//Return type of f is float,
//    inferred type: float*float -> float
let f(x,y):float = x+y
```

F# programming language

Partial evaluation

Let us consider the following functions:

```
let f1(x,y) = x+y

let f2 x y = x+y
```

F# programming language

Partial evaluation

Let us consider the following functions:

```
let f1(x,y) = x+y

let f2 x y = x+y
```

Function f1 has type:

Let us consider the following functions:

```
let f1(x,y) = x+y
```

```
let f2 x y = x+y
```

Function f1 has type:

```
val f1 : x:int * y:int -> int
```

F# programming language

Partial evaluation

Let us consider the following functions:

```
let f1(x,y) = x+y
```

```
let f2 x y = x+y
```

Function f1 has type:

```
val f1 : x:int * y:int -> int
```

Function f2 has type:

F# programming language

Partial evaluation

Let us consider the following functions:

```
let f1(x,y) = x+y

let f2 x y = x+y
```

Function f1 has type:

```
val f1 : x:int * y:int -> int
```

Function f2 has type:

```
val f2 : x:int -> y:int -> int
```

F# programming language

Partial evaluation

Let us consider the following functions:

```
let f1(x,y) = x+y

let f2 x y = x+y
```

Function f1 has type:

```
val f1 : x:int * y:int -> int
```

Function f2 has type:

```
val f2 : x:int -> y:int -> int
```

Function f2 can be partially evaluate:

```
let inc = f2 1
```

F# programming language

Partial evaluation

Let us consider the following functions:

```
let f1(x,y) = x+y

let f2 x y = x+y
```

Function f1 has type:

```
val f1 : x:int * y:int -> int
```

Function f2 has type:

```
val f2 : x:int -> y:int -> int
```

Function f2 can be partially evaluate:

```
let inc = f2 1
```

The two approaches are in fact equivalent! The second one is the standard (and more efficient).

Type parameters...

Let us consider the following definition:

```
let select x y z w = if (x=y) then z else w
```

Type parameters...

Let us consider the following definition:

```
let select x y z w = if (x=y) then z else w
```

The compiler has not info for inferring as type for x, y, z and w.

Type parameters. . .

Let us consider the following definition:

```
let select x y z w = if (x=y) then z else w
```

The compiler has not info for inferring as type for x, y, z and w. However, we know that:

1. x and y must have the same type (say it *a*);

Type parameters. . .

Let us consider the following definition:

```
let select x y z w = if (x=y) then z else w
```

The compiler has not info for inferring as type for x, y, z and w.
However, we know that:

1. x and y must have the same type (say it *a*);
2. z and w must have the same type (say it *b*);

Type parameters...

Let us consider the following definition:

```
let select x y z w = if (x=y) then z else w
```

The compiler has not info for inferring as type for x, y, z and w. However, we know that:

1. x and y must have the same type (say it *a*);
2. z and w must have the same type (say it *b*);
3. values of type *a* must support *equality*.

Type parameters...

Let us consider the following definition:

```
let select x y z w = if (x=y) then z else w
```

The compiler has not info for inferring as type for x, y, z and w.
However, we know that:

1. x and y must have the same type (say it *a*);
2. z and w must have the same type (say it *b*);
3. values of type *a* must support *equality*.

Type parameters...

Let us consider the following definition:

```
let select x y z w = if (x=y) then z else w
```

The compiler has not info for inferring as type for x, y, z and w.
However, we know that:

1. x and y must have the same type (say it *a*);
2. z and w must have the same type (say it *b*);
3. values of type *a* must support *equality*.

Any type satisfying the expected properties (equality for *a*) can be used in place of *a* and *b*, that can be considered as type parameters!

Type parameters. . .

Let us consider the following definition:

```
let select x y z w = if (x=y) then z else w
```

The compiler has not info for inferring as type for x, y, z and w.
However, we know that:

1. x and y must have the same type (say it *a*);
2. z and w must have the same type (say it *b*);
3. values of type *a* must support *equality*.

Any type satisfying the expected properties (equality for *a*) can be used in place of *a* and *b*, that can be considered as type parameters!

The following type is inferred for function select :

```
val select : x:'a -> y:'a -> z:'b -> w:'b -> 'b
    when 'a : equality
```

Recursive functions. . .

In functional programming the use of recursive definition is crucial.

F# programming language

Recursive functions. . .

In functional programming the use of recursive definition is crucial.

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

F# programming language

Recursive functions. . .

In functional programming the use of recursive definition is crucial.

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

This definition is not correct! The symbol fib is not defined when the body of the function is evaluated!

F# programming language

Recursive functions...

In functional programming the use of recursive definition is crucial.

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

This definition is not correct! The symbol fib is not defined when the body of the function is evaluated!

```
let rec fib(x) = //Note here the use of 'rec'
  if x<=2 then
    1
  else
    (fib x-1)+(fib x-2)
```

F# programming language

Tuples...

A tuple is a grouping of unnamed but ordered values, possibly of different types.

```
(element ,  ...  ,  element)
```

F# programming language

Tuples. . .

A tuple is a grouping of unnamed but ordered values, possibly of different types.

```
(element, ... , element)
```

Example:

```
let fib(x) =
  let rec _fib(x) =
    if x<=2 then
      (1,1)
    else
      let (a,b)=_fib(x-1)
      in
        (a+b,a)
  in
    let (a,_) = _fib(x)
    in
      a
```

F# programming language

Lists. . .

A list in F# is an ordered, immutable series of elements of the same type. Lists have type `'a list` .

F# programming language

Lists. . .

A list in F# is an ordered, immutable series of elements of the same type. Lists have type `'a list`.

You can define a list by explicitly listing out the elements, separated by semicolons and enclosed in square brackets;

```
let list123 = [ 1; 2; 3 ] //Type int list
let emptylist = [] //Type 'a list!
```

F# programming language

Lists...

A list in F# is an ordered, immutable series of elements of the same type. Lists have type `'a list`.

You can define a list by explicitly listing out the elements, separated by semicolons and enclosed in square brackets;

```
let list123 = [ 1; 2; 3 ] //Type int list
let emptylist = [] //Type 'a list!
```

You can also define list elements by using a range indicated by integers separated by the range operator `..`:

```
let list1 = [ 1..10 ]
```

F# programming language

Lists. . .

A list in F# is an ordered, immutable series of elements of the same type. Lists have type 'a list .

You can define a list by explicitly listing out the elements, separated by semicolons and enclosed in square brackets;

```
let list123 = [ 1; 2; 3 ] //Type int list
let emptylist = [] //Type 'a list!
```

You can also define list elements by using a range indicated by integers separated by the range operator ..:

```
let list1 = [ 1..10 ]
```

List operations:

- :: is used to add an element at the beginning of the list: a :: list1
- @ is used to concatenate two lists: l1@l2

Pattern matching...

The `match` expression provides branching control that is based on the comparison of an expression with a set of patterns.

Pattern matching...

The match expression provides branching control that is based on the comparison of an expression with a set of patterns.

```
// Match expression.
match test-expression with
| pattern1 [ when condition ] -> result-expression1
| pattern2 [ when condition ] -> result-expression2
| ...
```

Pattern matching...

The match expression provides branching control that is based on the comparison of an expression with a set of patterns.

```
// Match expression.
match test-expression with
| pattern1 [ when condition ] -> result-expression1
| pattern2 [ when condition ] -> result-expression2
| ...
```

Pattern can be used to inspect the structure of a value and bind values to variables:

```
match lst with
| [] -> exp_1
| v::tail -> exp_2
```

Pattern matching...

The match expression provides branching control that is based on the comparison of an expression with a set of patterns.

```
// Match expression.
match test-expression with
| pattern1 [ when condition ] -> result-expression1
| pattern2 [ when condition ] -> result-expression2
| ...
```

Pattern can be used to inspect the structure of a value and bind values to variables:

```
match lst with
| [] -> exp_1
| v::tail -> exp_2
```

Conditions are boolean expressions that can be used to limit the selection.

Example: Polynomial evaluation

A polynomial in a single indeterminate x is an expression o the form:

$$a_n x^n + \cdots + a_1 x + a_0$$

Example: Polynomial evaluation

A polynomial in a single indeterminate x is an expression o the form:

$$a_n x^n + \cdots + a_1 x + a_0$$

A polynomial can be represented as the list of its coefficients:

```
let poly = [ an; ... a1; a0 ]
```

Example: Polynomial evaluation

A polynomial in a single indeterminate x is an expression o the form:

$$a_n x^n + \cdots + a_1 x + a_0$$

A polynomial can be represented as the list of its coefficients:

```
let poly = [ an; ... a1; a0 ]
```

Write a function `eval` that received in input a list of coefficients and a value `x` computes the value of the polynomial.

Example: Polynomial evaluation

Solution 1:

```
let rec eval clist (x: float) =
    match clist with
    | [] -> 0.0
    | c::tail -> c*(x**float(clist.Length-1))+(eval tail x)
```

Example: Polynomial evaluation

Solution 1:

```
let rec eval clist (x: float) =
    match clist with
    | [] -> 0.0
    | c::tail -> c*(x**float(clist.Length-1))+(eval tail x)
```

Solution 2:

```
let eval2 clist (x: float) =
    let rec _eval2 clist v =
        match clist with
        | [] -> v
        | c::tail -> _eval2 tail (v*x+c)
    in
        _eval2 clist 0.0
```

Option type...

The option type is used when an actual value might not exist for a named value or variable.

Option type...

The option type is used when an actual value might not exist for a named value or variable.
An option has an underlying type and can hold a value of that type, or it might not have a value

```
'a option
```

Option type...

The option type is used when an actual value might not exist for a named value or variable.
An option has an underlying type and can hold a value of that type, or it might not have a value

```
'a option
```

Find the first element in a list matching a predicate:

```
let rec findFirstMatching pred l =
    match l with
    | [] -> None
    | v::tail -> if (pred v) then Some v
                 else findFirstMatching pred tail
```

Option type...

The option type is used when an actual value might not exist for a named value or variable.
An option has an underlying type and can hold a value of that type, or it might not have a value

```
'a option
```

Find the first element in a list matching a predicate:

```
let rec findFirstMatching pred l =
    match l with
    | [] -> None
    | v::tail -> if (pred v) then Some v
                 else findFirstMatching pred tail
```

The type of findFirstMatching is:

```
pred:('a -> bool) -> l:'a list -> 'a option
```

Excercises...

Ex. 0: Download and install F# developing environment. See instructions available here:

`https://docs.microsoft.com/en-us/dotnet/fsharp/get-started/`

Ex. 1: Write a function that given in input a and b computes their *mcd*.

Ex. 2: Write a function that given in input n returns true if n is a *prime number* and false otherwise.

Ex. 3: Write a function that given in input an integer n computes the list of its prime factors.