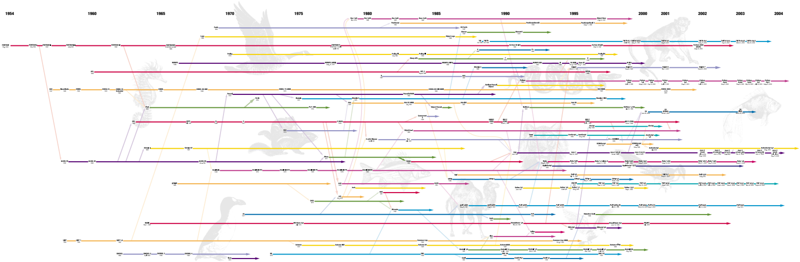
Programming paradigms

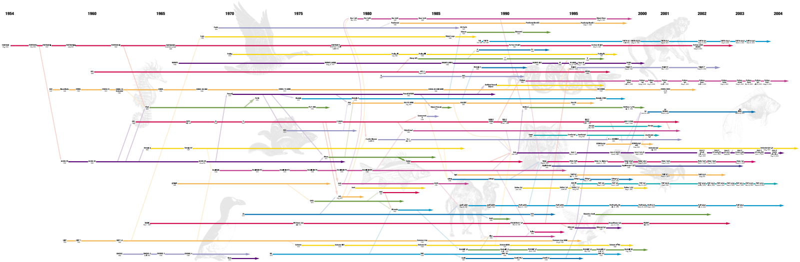
Prof. Michele Loreti

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

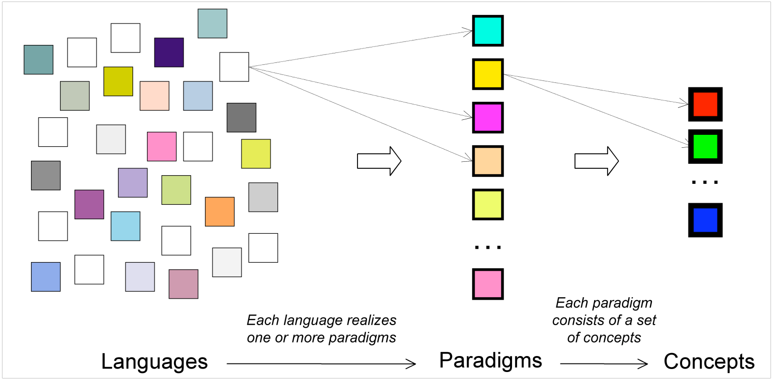
Programming paradigms are a way to classify programming languages based on their features.

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



Each language realizes one or more paradigms

Each paradigm consists of a set of concepts

Languages ⟶ Paradigms ⟶ Concepts

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. . .

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

This lecture. . .

In this lecture. . .

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

. . . 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'`).

F# programming language

Values (2/2)

- 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!

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!

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!!!!
```

F# programming language

Basic concepts

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

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
```

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)
```

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

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!

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:

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
```

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
```

F# programming language

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!
```

List operations:

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

F# programming language

Lists. . .

F# programming language

Lists. . .

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

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

List can be generated in a symbolic way expressions:

```
   let fiblist = [ for i in 1 .. 10 -> fib(i) ];;
```

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
```

Custom data type: Discriminated Unions...

Discriminated unions provide support for values that can be one of a number of named cases, possibly each with different values and types:

```
type type−name =
    | case−identifier1 [of [ fieldname1 : ] type1 [ ∗ [
    fieldname2 : ] type2 ...]
    | case−identifier2 [of [fieldname3 : ]type3 [ ∗ [
    fieldname4 : ]type4 ...]
```

Custom data type: Discriminated Unions...

Discriminated unions provide support for values that can be one of a number of named cases, possibly each with different values and types:

```
type type-name =
    | case-identifier1 [of [ fieldname1 : ] type1 [ * [
   fieldname2 : ] type2 ...]
    | case-identifier2 [of [fieldname3 : ]type3 [ * [
   fieldname4 : ]type4 ...]
```

Example:

```
type Shape =
    | Rectangle of width : float * length : float
    | Circle of radius : float
    | Prism of width : float * float * height : float
```

Example: Binary Search Trees!

Binary search trees keep their keys in sorted order:

- elements are inserted/removed from the tree by following the principle of binary search;
- elements traverse the tree from root to leaf by making decisions on the base of comparison.

Example: Binary Search Trees!

Binary search trees keep their keys in sorted order:

- elements are inserted/removed from the tree by following the principle of binary search;
- elements traverse the tree from root to leaf by making decisions on the base of comparison.

Exercise:

1. develop a data type for BST;
2. implement basic operations on BST...
    - insertion;
    - search;
    - deletion.

Example: Binary Search Trees!

Data type:

```
type bstree =
    EMPTY
  | NODE of value: int * left: bstree * right: bstree
```

Example: Binary Search Trees!

Data type:

```
type bstree =
    EMPTY
  | NODE of value: int * left: bstree * right: bstree
```

Add a value in the tree:

```
let rec add v t =
    match t with
    | EMPTY -> NODE(v,EMPTY,EMPTY)
    | NODE(v1,l,r) when v1<v -> NODE(v1,l,add v r)
    | NODE(v1,l,r) -> NODE(v1,add v l, r)
```

Example: Binary Search Trees!

Search for an element:

```
let rec contains v t =
    match t with
    | EMPTY -> false
    | NODE(v1,_,_) when v1=v -> true
    | NODE(v1,l,r) when v1<v -> contains v r
    | NODE(v1,l,r) -> contains v l
```

Example: Binary Search Trees!

Search for an element:

```
let rec contains v t =
    match t with
    | EMPTY -> false
    | NODE(v1,_,_) when v1=v -> true
    | NODE(v1,l,r) when v1<v -> contains v r
    | NODE(v1,l,r) -> contains v l
```

Merging trees:

```
let rec merge t1 t2 =
    match t1,t2 with
    | EMPTY,_ -> t2
    | _,EMPTY -> t1
    | NODE(v1,l1,r1),NODE(v2,l2,r2) when v1<v2 ->
            NODE(v1,l1,merge r1 t2)
    | NODE(v1,l1,r1),NODE(v2,l2,r2) ->
            NODE(v2,l2,merge r2 t1)
```

Example: Binary Search Trees!

Removing an element:

```
let rec remove v t =
    match t with
    | EMPTY -> EMPTY
    | NODE(v1,l,r) when v1=v -> merge l r
    | NODE(v1,l,r) when v1<v -> NODE(v1,l,remove v r)
    | NODE(v1,l,r) -> NODE(v1,remove v l,r)
```

Remarks. . .

The type `bstree` can only contain integer values.

Remarks. . .

The type `bstree` can only contain integer values.

It could be convenient, like we already observed for lists, define this type as parametrised!

Remarks. . .

The type `bstree` can only contain integer values.

It could be convenient, like we already observed for lists, define this type as parametrised!

The exact type of elements in a `bstree` could be chosen by the programmer!

Remarks. . .

The type `bstree` can only contain integer values.

It could be convenient, like we already observed for lists, define this type as parametrised!

The exact type of elements in a `bstree` could be chosen by the programmer!

We can use Generics!

Generics. . .

Generic programming is a style of computer programming in which elements of a program (procedures, functions, data-types,. . .) are written in terms of types to-be-specified-later.

Generics. . .

Generic programming is a style of computer programming in which elements of a program (procedures, functions, data-types,. . .) are written in terms of types to-be-specified-later.

These type parameters are then instantiated when needed for specific types provided as parameters.

Generics. . .

Generic programming is a style of computer programming in which elements of a program (procedures, functions, data-types,. . .) are written in terms of types to-be-specified-later.

These type parameters are then instantiated when needed for specific types provided as parameters.

A generic indicates values, methods, properties, and aggregate types such as classes, records, and discriminated unions can be generic.

Generics. . .

Generic programming is a style of computer programming in which elements of a program (procedures, functions, data-types,. . .) are written in terms of types to-be-specified-later.

These type parameters are then instantiated when needed for specific types provided as parameters.

A generic indicates values, methods, properties, and aggregate types such as classes, records, and discriminated unions can be generic.

In F# function values, methods, properties, and aggregate types such as classes, records, and discriminated unions can be generic.

Generics. . .

Generics. . .

Generic constructs contain at least one type parameter, which is usually supplied by the user of the generic construct.

Generics...

Generic constructs contain at least one type parameter, which is usually supplied by the user of the generic construct.

Generic functions and types enable you to write code that works with a variety of types without repeating the code for each type:

```
/ Explicitly generic function.
let function-name<type-parameters> parameter-list =
function-body

// Explicitly generic type.
type type-name<type-parameters> type-definition
```

Generics. . .

Automatic Generation: Type Inference

The F# compiler, when it performs type inference on a function, determines whether a given parameter can be generic.

Generics. . .

Automatic Generation: Type Inference

The F# compiler, when it performs type inference on a function, determines whether a given parameter can be generic.

The compiler examines each parameter and determines whether the function has a dependency on the specific type of that parameter. If it does not, the type is inferred to be generic.

Generics. . .

Automatic Generation: Type Inference

The F# compiler, when it performs type inference on a function, determines whether a given parameter can be generic.

The compiler examines each parameter and determines whether the function has a dependency on the specific type of that parameter. If it does not, the type is inferred to be generic.

Example:

```
let max a b = if a > b then a else b
```

This function has type `'a -> 'a -> 'a when 'a comparison`.

Generics. . .

Automatic Generation: Type Inference

The F# compiler, when it performs type inference on a function, determines whether a given parameter can be generic.

The compiler examines each parameter and determines whether the function has a dependency on the specific type of that parameter. If it does not, the type is inferred to be generic.

Example:

```
let max a b = if a > b then a else b
```

This function has type `'a -> 'a -> 'a when 'a comparison`.

Above `when 'a comparison` is a constraint.

Example...

We can change the definition of bstree as follows:

```
type bstree<'T when 'T: comparison> =
  EMPTY
  | NODE of value: 'T * left: 'T bstree * right: 'T bstree
```

Example. . .

We can change the definition of `bstree` as follows:

```
type bstree<'T when 'T: comparison> =
  EMPTY
  | NODE of value: 'T * left: 'T bstree * right: 'T bstree
```

We have not to change the functions `add`, `contains`, and `remove`!

Namespace. . .

A namespace lets you organize code into areas of related functionality by enabling you to attach a name to a grouping of program elements.

Namespace...

A namespace lets you organize code into areas of related functionality by enabling you to attach a name to a grouping of program elements.

```
namespace [parent-namespaces.]identifier
```

Namespace...

A namespace lets you organize code into areas of related functionality by enabling you to attach a name to a grouping of program elements.

```
namespace [parent-namespaces.]identifier
```

If you want to put code in a namespace, the first declaration in the file must declare the namespace. The contents of the entire file then become part of the namespace.

Namespace...

A namespace lets you organize code into areas of related functionality by enabling you to attach a name to a grouping of program elements.

```
namespace [parent-namespaces.]identifier
```

If you want to put code in a namespace, the first declaration in the file must declare the namespace. The contents of the entire file then become part of the namespace.

Namespaces cannot directly contain values and functions. Instead, values and functions must be included in modules, and modules are included in namespaces. Namespaces can contain types, modules.

Modules

A module is a grouping of F# code, such as values, types, and function values.

Modules

A module is a grouping of F# code, such as values, types, and function values.

Grouping code in modules helps keep related code together and helps avoid name conflicts in your program.

Modules

A module is a grouping of F# code, such as values, types, and function values.

Grouping code in modules helps keep related code together and helps avoid name conflicts in your program.

```
module [accessibility-modifier] module-name =
    declarations
```

Modules

A module is a grouping of F# code, such as values, types, and function values.

Grouping code in modules helps keep related code together and helps avoid name conflicts in your program.

```
module [accessibility-modifier] module-name =
    declarations
```

We can build the module of Bstrees!

Modules: List...

Modules: List. . .

average: Returns the average of the elements in the list.

Modules: List. . .

average: Returns the average of the elements in the list.

```
// Signature:
List.average : ^T list -> ^T
  (requires ^T with static member (+)
    and ^T with static member DivideByInt
    and ^T with static member Zero)

// Usage:
List.average list

// Example
average([1.0 .. 10.0])
```

Modules: List...

averageBy: Returns the average of the elements generated by applying the function to each element of the list.

```
// Signature:
List.averageBy : ('T -> ^U) -> 'T list -> ^U
  (requires ^U with static member (+)
    and ^U with static member DivideByInt
    and ^U with static member Zero)

// Usage:
List.averageBy projection list


// Example
List.averageBy (fun x -> x**2.0) [ 1.0 .. 10.0 ];;
```

Modules: List. . .

filter : Returns a new collection containing only the elements of the collection for which the given predicate returns true.

```
// Signature:
List.filter : ('T -> bool) -> 'T list -> 'T list

// Usage:
List.filter predicate list

// Example
List.filter (fun x -> x%3=0) [ 1 .. 100 ];;
```

Modules: List. . .

map: Creates a new collection whose elements are the results of applying the given function to each of the elements of the collection.

```
// Signature:
List.map : ('T -> 'U) -> 'T list -> 'U list

// Usage:
List.map mapping list

// Example
List.map (fun x -> x*x) [ 1 .. 10 ];;
```

Modules: List. . .

reduce: Applies a function to each element of the collection, threading an accumulator argument through the computation.

Modules: List...

`reduce`: Applies a function to each element of the collection, threading an accumulator argument through the computation.

Given a function `f` and a list containing `i0,i1,i2,...,ik` computes:

```
f (... (f i0 i1) i2 ...) ik
```

Modules: List. . .

reduce: Applies a function to each element of the collection, threading an accumulator argument through the computation.

Given a function `f` and a list containing `i0,i1,i2,...,ik` computes:

```
    f (... (f i0 i1) i2 ...) ik
```

```
// Signature:
List.reduce : ('T -> 'T -> 'T) -> 'T list -> 'T

// Usage:
List.reduce reduction list

// Example:
List.reduce (fun x y -> x+y) [1..100]
```

Map-Reduce. . .

Map-Reduce is a programming pattern for processing and generating (possibly big) data sets.

Map-Reduce. . .

Map-Reduce is a programming pattern for processing and generating (possibly big) data sets.

Elements in the data set are not processed in isolation but as part of a gruop.

Map-Reduce. . .

Map-Reduce is a programming pattern for processing and generating (possibly big) data sets.

Elements in the data set are not processed in isolation but as part of a gruop.

The Map-Reduce patter relies on three main functions:

- a filter that restricts the dataset to the elements satisfying a predicate;

Map-Reduce. . .

Map-Reduce is a programming pattern for processing and generating (possibly big) data sets.

Elements in the data set are not processed in isolation but as part of a gruop.

The Map-Reduce patter relies on three main functions:

- a filter that restricts the dataset to the elements satisfying a predicate;
- a map function that processes elements dataset;

Map-Reduce. . .

Map-Reduce is a programming pattern for processing and generating (possibly big) data sets.

Elements in the data set are not processed in isolation but as part of a gruop.

The Map-Reduce patter relies on three main functions:

- a filter that restricts the dataset to the elements satisfying a predicate;
- a map function that processes elements dataset;
- a reduce function that combines result.

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 input *a* and *b* computes their *mcd*.

Ex. 2: Write a function that given an 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.

Ex. 4: Can *map-filter-reduce* be used to simplify the code we have considered so far?

BS Trees. . .

Ex. 5 Implement function `size` that given a tree `t` computes the number of elements stored in `t`.

Ex. 6 Implement function `height` that given a tree `t` computes its height (an empty BST has height equal to 0).

Ex. 7 Implement function `balance` that given a tree `t` computes a tree `t1` with the same elements its height (an empty BST has height equal to 0).

Ex. 8 Implement AVL data structure.

To be continued. . .