Functional programming in F#: Data Structures

Prof. Michele Loreti

Programmazione Avanzata

Corso di Laurea in Informatica (L31)

Scuola di Scienze e Tecnologie

Summary of previous lectures

In the previous lecture we have. . . :

- introduced basic principles of programming paradigms;
- introduced basic concept of a functional language, F#:
  - elementary types;
  - expressions;
  - function definitions;
  - type inference;
  - lists.

Exercises: solutions

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

Exercises: solutions

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

```
let mcd (a: uint32) (b: uint32) =
    let rec _mcd x y =
        if y=0u then x
        else _mcd y (x%y)
    in
        _mcd (max a b) (min a b)
```

Exercises: solutions

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

Exercises: solutions

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

```
let isPrime n =
    let limit = int(sqrt(float(n)))+1
    let rec test n x =
        if x>=limit then true
        else if n%x = 0 then false
        else test n (x+1)
    in
        test n 2
```

Exercises: solutions

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

Exercises: solutions

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

```
let primeFactors n =
    let rec divP x y =
        if x%y<>0 then (x,0)
        else
            let (z,p) = divP (x/y) y
            in
                (z,p+1)
    let rec combP x y =
        if x<2 then []
        else
            match divP x y with
            | (_,0) -> combP x (y+1)
            | (z,p) -> y::(combP z (y+1))
    in
        combP n 2
```

Imperative statements

To simplify coding, standard imperative statements are often included in functional languages.

Imperative statements

To simplify coding, standard imperative statements are often included in functional languages.

You can use the keyword mutable to specify a variable that can be changed.

Imperative statements

To simplify coding, standard imperative statements are often included in functional languages.

You can use the keyword mutable to specify a variable that can be changed.

```
let mutable x = 1 //declaration
x <- x + 1 //assignment
```

Imperative statements

To simplify coding, standard imperative statements are often included in functional languages.

You can use the keyword mutable to specify a variable that can be changed.

```
let mutable x = 1 //declaration
x <- x + 1 //assignment
```

Mutable variables in F# should generally have a limited scope, either as a field of a type or as a local value.

Imperative statements

To simplify coding, standard imperative statements are often included in functional languages.

You can use the keyword mutable to specify a variable that can be changed.

```
let mutable x = 1 //declaration
x <- x + 1 //assignment
```

Mutable variables in F# should generally have a limited scope, either as a field of a type or as a local value.

Use of mutable variables may introduce side effects!

Imperative statements

To simplify coding, standard imperative statements are often included in functional languages.

You can use the keyword mutable to specify a variable that can be changed.

```
let mutable x = 1 //declaration
x <- x + 1 //assignment
```

Mutable variables in F# should generally have a limited scope, either as a field of a type or as a local value.

Use of mutable variables may introduce side effects!

```
let mutable counter = 0

let step () = x <- x+1;x
```

Loop expressions...

The for ... to expression is used to iterate in a loop over a range of values of a loop variable:

```
for identifier = start [ to | downto ] finish do
    body-expression
```

Loop expressions...

The for ... to expression is used to iterate in a loop over a range of values of a loop variable:

```
for identifier = start [ to | downto ] finish do
    body-expression
```

The while ... do expression is used to perform iterative execution (looping) while a specified test condition is true:

```
while test-expression do
    body-expression
```

Examples: list of prime factors...

```
let primeFactors2 n =
    let compP x y =
        let mutable count = 0
        let mutable v = x
        while v%y = 0 do
            v <- v/y
            count <- count+1
        (v,count)
    let mutable v = n
    let mutable count = 2
    let mutable res = []
    while v >= 2 do
            let (r,p) = divP v count
            v <- r;
            if p>0 then res <- count::res;
            count <- count+1;
    res
```

Custom data type: Records...

Records represent simple aggregates of named values

Custom data type: Records...

Records represent simple aggregates of named values

```
type typename = {
    [ mutable ] label1 : type1;
    [ mutable ] label2 : type2;
    ...
}
```

Custom data type: Records...

Records represent simple aggregates of named values

```
type typename = {
    [ mutable ] label1 : type1;
    [ mutable ] label2 : type2;
    ...
}
```

Example:

```
type MyPoint = { x: float ; y: float }
```

Custom data type: Records...

Records represent simple aggregates of named values

```
type typename = {
    [ mutable ] label1 : type1;
    [ mutable ] label2 : type2;
    ...
}
```

Example:

```
type MyPoint = { x: float ; y: float }
```

You can initialize records by using the labels that are defined in the record:

```
let p = { x=10.0 ; y=10.0 }
```

Custom data type: Records...

Fields in a record are accessible via the standard `name.field` notation:

Custom data type: Records...

Fields in a record are accessible via the standard name.field notation:

```
let distance p1 p2 =
        ((p1.x-p2.x)**2.0+(p1.y-p2.y)**2.0)**0.5
```

Custom data type: Records...

Fields in a record are accessible via the standard `name.field` notation:

```
let distance p1 p2 =
        ((p1.x-p2.x)**2.0+(p1.y-p2.y)**2.0)**0.5
```

Records can be used with pattern matching. You can specify some fields explicitly and provide variables for other fields:

```
let pointInfo p =
    match p with
    | { x=0.0 ; y=0.0 } ->
        printf "This is the origin!"
    | { x=0.0 ; y=_ } ->
        printf "This point is located on the x-axes!"
    | { x=_ ; y=0.0 } ->
        printf "This point is located on the y-axes!"
    | { x=xval ; y=yval } ->
        printf "This point is located at (%f,%f)" xval yval
```

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)
```

Exercises...

Ex. 4 Implement function `size` that given a tree `t` computes the number of elements stored in `t`.

Ex. 5 Implement function `height` that given a tree `t` computes its height (an empty BST has height equal to 0).

Ex. 6 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. 7 Implement AVL data structure.

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!