Exercises: Funcitonal Programming in Action

Prof. Michele Loreti

Programmazione Avanzata

Corso di Laurea in Informatica (L31)

Scuola di Scienze e Tecnologie

Excercise 1: Computing GCD...

The *greatest common divisor* (gcd) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers.

Excercise 1: Computing GCD...

The *greatest common divisor* (gcd) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers.

Euclidean Algorithm: Given two integers a and b...

1. Compute the sequence of numbers a_i, b_i such that:

$$a_i = \begin{cases} a & (i = 0) \\ b_{i-1} & (i > 0) \end{cases} \qquad b_i = \begin{cases} b & (i = 0) \\ a_{i-1} \mod b_{i-1} & (i > 0) \end{cases}$$

2. $gcd(a, b) = a_i$, where i is the least index such that $b_i = 0$.

Excercise 1: Computing GCD...

GCD of two integers:

```
let rec gcd2 a b =
      if b=0 then a
      else gcd2 b (a%b)
```

Excercise 1: Computing GCD...

GCD of two integers:

```
let rec gcd2 a b =
      if b=0 then a
      else gcd2 b (a%b)
```

GCD of a list of integers

Excercise 1: Computing GCD...

GCD of two integers:

```
let rec gcd2 a b =
    if b=0 then a
    else gcd2 b (a%b)
```

GCD of a list of integers (solution 1):

```
let rec gcd alist =
    match alist with
    [] -> None
    | a::tail ->
        match gcd tail with
        None -> Some a
        | Some b -> Some (gcd2 a b)
```

Excercise 1: Computing GCD...

GCD of two integers:

```
let rec gcd2 a b =
      if b=0 then a
      else gcd2 b (a%b)
```

GCD of a list of integers (solution 1):

```
let rec gcd alist =
    match alist with
    [] -> None
    | a::tail ->
        match gcd tail with
        None -> Some a
        | Some b -> Some (gcd2 a b)
```

GCD of a list of integers (solution 2):

Excercise 1: Computing GCD...

GCD of two integers:

```
let rec gcd2 a b =
    if b=0 then a
    else gcd2 b (a%b)
```

GCD of a list of integers (solution 1):

```
let rec gcd alist =
    match alist with
    [] -> None
    | a::tail ->
        match gcd tail with
        None -> Some a
        | Some b -> Some (gcd2 a b)
```

GCD of a list of integers (solution 2):

```
let gcdlist alist =
    List.reduce gcd2 alist
```

Excercise 2: Prime numbers...

Given a number n checks if its prime. A number n is prime if and only if it can be divided by 1 and itself.

Excercise 2: Prime numbers...

Given a number n checks if its prime. A number n is prime if and only if it can be divided by 1 and itself. If n is not prime, at least a divisor of n is between 1 and $\sqrt{n}$.

Excercise 2: Prime numbers...

Given a number n checks if its prime. A number n is prime if and only if it can be divided by 1 and itself. If n is not prime, at least a divisor of n is between 1 and $\sqrt{n}$.

Solution 1:

```
let isPrime n =
    let rec _isPrime n v =
        if v=1 then true
        else if n%v=0 then false
        else _isPrime n (v-1)
    in _isPrime n (int (sqrt (float n)))
```

Excercise 2: Prime numbers...

Given a number n checks if its prime. A number n is prime if and only if it can be divided by 1 and itself. If n is not prime, at least a divisor of n is between 1 and $\sqrt{n}$.

Solution 1:

```
let isPrime n =
    let rec _isPrime n v =
        if v=1 then true
        else if n%v=0 then false
        else _isPrime n (v-1)
    in _isPrime n (int (sqrt (float n)))
```

Solution 2:

```
let isPrime2 n =
    (n>1) && (not (List.exists
       (fun i -> n%i=0) [ 2 .. (int (sqrt (float n)))]))
```

Excercise 3: Prime factors...

Compute the list of prime factors of an integer n (1 and n excluded).

Excercise 3: Prime factors...

Compute the list of prime factors of an integer n (1 and n excluded).

Solution:

```
let primeFactors n =
    List.filter
      (fun i -> n%i=0)
      (List.filter isPrime [2 .. n-1])
```

Example Binary Search Trees

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search...

... when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf;

... making comparisons to keys stored in the nodes of the tree and deciding, on the basis of the comparison, to continue searching in the left or right subtrees.

Example Binary Search Trees

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search...

- ... when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf;
- ... making comparisons to keys stored in the nodes of the tree and deciding, on the basis of the comparison, to continue searching in the left or right subtrees.

We can use an enumeration to define the set of Binary Search Trees:

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

Example Binary Search Trees

Operations on trees (1/7)

Adding an element:

Example Binary Search Trees

Operations on trees (1/7)

Adding an element:

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

Example Binary Search Trees

Operations on trees (1/7)

Adding an element:

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

Check if an element is in the tree:

Example Binary Search Trees

Operations on trees (1/7)

Adding an element:

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

Check if an element is in the tree:

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

Example Binary Search Trees

Operations on trees (2/7)

Get min element in the tree:

Example Binary Search Trees

Operations on trees (2/7)

Get min element in the tree:

```
let rec getMin t =
    match t with
    EMPTY -> None
    | BSTREE(v1,EMPTY,_) -> Some v1
    | BSTREE(v1,t1,_) -> getMin t1
```

Example Binary Search Trees

Operations on trees (2/7)

Get min element in the tree:

```
let rec getMin t =
    match t with
    EMPTY -> None
    | BSTREE(v1,EMPTY,_) -> Some v1
    | BSTREE(v1,t1,_) -> getMin t1
```

Get max element in the tree:

Example Binary Search Trees

Operations on trees (2/7)

Get min element in the tree:

```
let rec getMin t =
    match t with
    EMPTY -> None
    | BSTREE(v1,EMPTY,_) -> Some v1
    | BSTREE(v1,t1,_) -> getMin t1
```

Get max element in the tree:

```
let rec getMax t =
    match t with
    EMPTY -> None
    | BSTREE(v1,_,EMPTY) -> Some v1
    | BSTREE(v1,_,t1) -> getMax t1
```

Example Binary Search Trees

Operations on trees (3/7)

Number of elements in the tree:

Example Binary Search Trees

Operations on trees (3/7)

Number of elements in the tree:

```
let rec size t =
    match t with
    EMPTY -> 0
    | BSTREE(_,l,r) -> 1+(size l)+(size r)
```

Example Binary Search Trees

Operations on trees (3/7)

Number of elements in the tree:

```
let rec size t =
    match t with
    EMPTY -> 0
    | BSTREE(_,l,r) -> 1+(size l)+(size r)
```

Height of the tree:

Example Binary Search Trees

Operations on trees (3/7)

Number of elements in the tree:

```
let rec size t =
    match t with
    EMPTY -> 0
    | BSTREE(_,l,r) -> 1+(size l)+(size r)
```

Height of the tree:

```
let rec height t =
    match t with
    EMPTY -> 0
    | BSTREE(_,l,r) -> 1+(max (height l) (height r))
```

Example Binary Search Trees

Operations on trees (4/7)

Ordered list of elements in the tree:

Example Binary Search Trees

Operations on trees (4/7)

Ordered list of elements in the tree:

```
let rec listOf t =
    match t with
    EMPTY -> []
    | BSTREE(v1,l,r) -> (listOf l)@(v1::(listOf r))
```

Example Binary Search Trees

Operations on trees (5/7)

Balance a tree:

Example Binary Search Trees

Operations on trees (5/7)

Balance a tree:

```
let balance t =
    let rec _fromOrderedList lst =
        match lst with
        [] -> EMPTY
        | [ v ] -> BSTREE(v,EMPTY,EMPTY)
        | _ ->
            let l1,l2 = List.splitAt (lst.Length/2) lst
            in
                match l2 with
                [] -> _fromOrderedList l1
                | v::tail ->
                    BSTREE(v,
                        (_fromOrderedList l1),
                        (_fromOrderedList tail) )
    in
        _fromOrderedList (listOf t)
```

Example Binary Search Trees

Operations on trees (6/7)

Filtering elements:

Example Binary Search Trees

Operations on trees (6/7)

Filtering elements:

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

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

Example Binary Search Trees

Operations on trees (7/7)

Merging two trees:

Example Binary Search Trees

Operations on trees (7/7)

Merging two trees:

```
let rec merge t1 t2 =
    match t1,t2 with
    | EMPTY,_ -> t2
    | _,EMPTY -> t1
    | BSTREE(v1,l1,r1),BSTREE(v2,l2,r2) when v1<v2 ->
      let l11 = getAllLessThan v2 r1
      let l12 = getAllGreaterThan v2 r1
      let l21 = getAllLessThan v1 l2
      let l22 = getAllGreaterThan v1 l2
         BSTREE(v2, BSTREE(v1,merge l1 l21,merge l11 l21),
             merge l12 r2)
    | BSTREE(v1,l1,r1),BSTREE(v2,l2,r2) -> //v1 >= v2
      let l11 = getAllLessThan v2 r1
      let l12 = getAllGreaterThan v2 r1
      let l21 = getAllLessThan v1 l2
      let l22 = getAllGreaterThan v1 l2
         BSTREE(v1,BSTREE(v2,l2,merge l11 l21),merge l12 r1)
```

Concluding remarks...

What we learnt...

Concluding remarks...

What we learnt...

1. Type, type inference and type checking;

What we are able to do now...

Concluding remarks...

What we learnt...

1. Type, type inference and type checking;
2. Generic types and constraints;

What we are able to do now...

Concluding remarks...

What we learnt...

1. Type, type inference and type checking;
2. Generic types and constraints;
3. Functions are first order data;

What we are able to do now...

Concluding remarks...

What we learnt...

1. Type, type inference and type checking;
2. Generic types and constraints;
3. Functions are first order data;
4. Programming with pattern matching;

What we are able to do now...

Concluding remarks...

What we learnt...

1. Type, type inference and type checking;
2. Generic types and constraints;
3. Functions are first order data;
4. Programming with pattern matching;
5. Filter-map-reduce as a pattern for handling collections of data.

What we are able to do now...

Concluding remarks...

What we learnt...

1. Type, type inference and type checking;
2. Generic types and constraints;
3. Functions are first order data;
4. Programming with pattern matching;
5. Filter-map-reduce as a pattern for handling collections of data.

What we are able to do now...

1. Infer types of simple expressions/programs;

Concluding remarks...

What we learnt...

1. Type, type inference and type checking;
2. Generic types and constraints;
3. Functions are first order data;
4. Programming with pattern matching;
5. Filter-map-reduce as a pattern for handling collections of data.

What we are able to do now...

1. Infer types of simple expressions/programs;
2. Understand F# code;

Concluding remarks...

What we learnt...

1. Type, type inference and type checking;
2. Generic types and constraints;
3. Functions are first order data;
4. Programming with pattern matching;
5. Filter-map-reduce as a pattern for handling collections of data.

What we are able to do now...

1. Infer types of simple expressions/programs;
2. Understand F# code;
3. Write simple F# functions implementing simple algorithms;

Concluding remarks...

What we learnt...

1. Type, type inference and type checking;
2. Generic types and constraints;
3. Functions are first order data;
4. Programming with pattern matching;
5. Filter-map-reduce as a pattern for handling collections of data.

What we are able to do now...

1. Infer types of simple expressions/programs;
2. Understand F# code;
3. Write simple F# functions implementing simple algorithms;
4. Apply *filter-map-reduce* patter.

To be continued...