F#: References and Arrays

Prof. Michele Loreti

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

Reference Cells...

Reference cells are storage locations that enable you to create mutable values with reference semantics.

Reference Cells...

Reference cells are storage locations that enable you to create mutable values with reference semantics.

The `ref` operator can be used before a value to create a new reference cell that encapsulates the value.

Reference Cells...

Reference cells are storage locations that enable you to create mutable values with reference semantics.

The `ref` operator can be used before a value to create a new reference cell that encapsulates the value.

Operator `!` is used to access the content of a cell.

Reference Cells...

Reference cells are storage locations that enable you to create mutable values with reference semantics.

The `ref` operator can be used before a value to create a new reference cell that encapsulates the value.

Operator `!` is used to access the content of a cell.

The content of the cell can be changed because it is mutable.

Reference Cells...

Reference cells are storage locations that enable you to create mutable values with reference semantics.

The ref operator can be used before a value to create a new reference cell that encapsulates the value.

Operator ! is used to access the content of a cell.

The content of the cell can be changed because it is mutable.

```
// Declare a reference.
let refVar = ref 6

// Change the value referred to by the reference.
refVar := 50

// Dereference by using the ! operator.
printfn "%d" !refVar
```

Arrays. . .

Arrays are fixed-size, zero-based, mutable collections of consecutive data elements that are all of the same type.

Arrays...

Arrays are fixed-size, zero-based, mutable collections of consecutive data elements that are all of the same type.
Arrays can be created in different way...

- by listing consecutive values between [| and |] and separated by semicolons:

```
let array1 = [| 1; 2; 3 |]
```

Arrays...

Arrays are fixed-size, zero-based, mutable collections of consecutive data elements that are all of the same type.
Arrays can be created in different way...

- by listing consecutive values between `[|` and `|]` and separated by semicolons:

```
let array1 = [| 1; 2; 3 |]
```

- ...listing each element on a separate line (semicolon is optional):

```
let array1 =
    [|
        1
        2
        3
     |]
```

Arrays. . .

Arrays are fixed-size, zero-based, mutable collections of consecutive data elements that are all of the same type.
Arrays can be created in different way. . .

- by listing consecutive values between [| and |] and separated by semicolons:

```
let array1 = [| 1; 2; 3 |]
```

- . . . listing each element on a separate line (semicolon is optional):

```
let array1 =
    [|
        1
        2
        3
     |]
```

- . . . by using sequence expressions

```
let array3 = [| for i in 1 .. 10 -> i * i |]
```

Sequence expressions. . .

A sequence expression is an expression that evaluates to a sequence.

Sequence expressions...

A sequence expression is an expression that evaluates to a sequence.

The simplest form specifies a range:

```
seq { 1 .. 5 }
```

Sequence expressions...

A sequence expression is an expression that evaluates to a sequence.

The simplest form specifies a range:

```
seq { 1 .. 5 }
```

You can also specify an increment (or decrement) between two double periods:

```
// Sequence that has an increment.
seq { 0 .. 10 .. 100 }
```

Sequence expressions. . .

A sequence expression is an expression that evaluates to a sequence.

The simplest form specifies a range:

```
seq { 1 .. 5 }
```

You can also specify an increment (or decrement) between two double periods:

```
// Sequence that has an increment.
seq { 0 .. 10 .. 100 }
```

Sequences can be also obtained from the evaluation of an expression:

```
seq { for i in 1 .. 10 -> i*i }
```

Sequence expressions...

Sequence expressions can be used in...

Sequence expressions...

Sequence expressions can be used in...

...iterators:

```
for i in 1 .. 10 do
  printf "%d\n" i
```

Sequence expressions...

Sequence expressions can be used in...

...iterators:

```
for i in 1 .. 10 do
  printf "%d\n" i
```

...list expressions:

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

Sequence expressions. . .

Sequence expressions can be used in. . .

. . . iterators:

```
for i in 1 .. 10 do
  printf "%d\n" i
```

. . . list expressions:

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

. . . array expressions:

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

Exercises: Binary Search Trees

Prof. Michele Loreti

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

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 (4/7Pm)

Ordered list of elements in the tree:

Example Binary Search Trees

Operations on trees (4/7Pm)

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 (4/7Pm)

Ordered list of elements in the tree:

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

Ordered array of elements in the tree:

Ordered list of elements in the tree:

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

Ordered array of elements in the tree:

```
let arrayOf t =
    List.toArray (listOf t)
```

Example Binary Search Trees

Operations on trees (5/7)

Create a tree from an array of sorted elements:

Example Binary Search Trees
Operations on trees (5/7)

Create a tree from an array of sorted elements:

```
let fromArray<'T when 'T:comparison> (a: 'T []) =
    let rec _fromArray i j =
        if j<=i then EMPTY
        else
            let m = i+(j-i)/2
            let v = a.[m]
            BSTREE(v,_fromArray i m, _fromArray (m+1) j)
    _fromArray 0 (Array.length a)
```

Example Binary Search Trees

Operations on trees (5/7)

Create a tree from an array of sorted elements:

```
let fromArray<'T when 'T:comparison> (a: 'T []) =
    let rec _fromArray i j =
        if j<=i then EMPTY
        else
            let m = i+(j-i)/2
            let v = a.[m]
            BSTREE(v,_fromArray i m, _fromArray (m+1) j)
    _fromArray 0 (Array.length a)
```

Balance a tree:

Example Binary Search Trees

Operations on trees (5/7)

Create a tree from an array of sorted elements:

```
let fromArray<'T when 'T:comparison> (a: 'T []) =
    let rec _fromArray i j =
        if j<=i then EMPTY
        else
            let m = i+(j-i)/2
            let v = a.[m]
            BSTREE(v,_fromArray i m, _fromArray (m+1) j)
    _fromArray 0 (Array.length a)
```

Balance a tree:

```
let balance t = fromArray (arrayOf 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)
```