

# Imperative programming in F#

Björn Lisper  
School of Innovation, Design, and Engineering  
Mälardalen University

`bjorn.lisper@mdh.se`  
`http://www.idt.mdh.se/~blr/`

---

# F# is a Multiparadigm Programming Language

So far we have used F# mainly as a functional language

But F# is really a *multi-paradigm* language

It supports both functional, imperative, and object-oriented programming

Although the focus of this course is functional programming, we will spend some time on the imperative and object-oriented parts in F#

---

## F# as an Imperative Language

We have already seen some limited side-effects (`printf`, file I/O), and sequencing

In addition, F# has:

- *mutable data* (that can be overwritten),
- *imperative control structures* (loops, conditionals), and
- *iteration over sequences, lists, and arrays* (similar to loops)

---

## Mutable Variables

F# has *mutable* variables (sometimes called *locations*)

Their contents can be changed

Declared with keyword `mutable`:

```
> let mutable x = 5;;
```

```
val mutable x : int = 5
```

Can be of any type:

```
> let mutable f = fun x -> x + 1;;
```

```
val mutable f : (int -> int)
```

---

## Updating Mutable Variables

Update (assignment) is done using the “<-” operator:

```
> let mutable x = 5;;
```

```
val mutable x : int = 5
```

```
> x <- x + 1;;
```

```
val it : unit = ()
```

```
> x;;
```

```
val it : int = 6
```

---

## Using the Values of Mutable Variables

The current value of a mutable variable is returned by its name. Thus, “`x`” refers to the current value of `x`. This has some consequences. An example:

```
> let mutable x = 5;;  
val mutable x : int = 5  
> let y = [x;x];;  
val y : int list = [5; 5]  
> x <- x + 1;;  
val it : unit = ()  
> y;;  
val it : int list = [5; 5]
```

So the list `y` is not changed when `x` is updated. This is because the current *value* of `x` was used when creating `y`. `y` is an ordinary, immutable list

---

## Mutable Records

We have already seen (immutable) records

Like variables, record fields can be declared `mutable` meaning that they can be updated

An example: an account record, having three fields: an account holder field (`string`, immutable), an account number field (`int`, immutable), an amount field (`int`, mutable), and a field counting the number of transactions (ditto)

(See next page)

---

```
type Account =  
    { owner : string;  
      number : int;  
      mutable amount : int;  
      mutable no_of_trans : int }
```

**A function to initialize an account record:**

```
let account_init own no =  
    { owner = own;  
      number = no;  
      amount = 0;  
      no_of_trans = 0 }
```



---

A mutable field can be updated with the “<-” operator:

```
account.no_of_trans <- account.no_of_trans + 1
```

**Example:** a function that adds an amount to an account:

```
let add_amount account money =  
    account.amount <- account.amount + money  
    account.no_of_trans <- account.no_of_trans + 1
```

---

## Mutable Reference Cells

They provide a third way to have mutable data in F#

Main difference to mutable variables is that the reference cells themselves can be referenced, not just the values held in them

Type `'a ref`, meaning “a cell that holds a value of type `'a`”. Initialized with function `ref : 'a -> 'a ref`:

```
let r = ref 5
```

Creates a reference cell `r : int ref` that holds the value 5

`r` is the cell itself. Its *contents* can be accessed with the “!” prefix operator:

```
!r ==> 5
```

Note the difference between `r` (the *cell*), and `!r` (the *contents* of the cell)

---

## Updating Reference Cells

The binary infix operator `(:=)` : `'a ref -> 'a -> unit` is used to update the contents of a reference cell

Creating/initializing, accessing, and updating a reference cell:

```
let r = ref 5
printf "Contents of r: %d\n" !r
r := !r - 2
printf "New contents of r: %d\n" !r
```

Resulting printout:

```
Contents of r: 5
New contents of r: 3
```

---

## Defining Mutable Reference Cells

Mutable reference cells can be defined in F# itself

They are simply records with one mutable field “contents”:

```
type ref<'a> = { mutable contents: 'a }  
let (!) r = r.contents  
let (:=) r v = r.contents <- v  
let ref v = { contents = v }
```

---

## Handling Reference Cells

Reference cells can be stored in data structures, and passed around. They can be accessed using the ordinary operations on data structures:

```
> let r = [ref 5;ref 3];;
```

```
val r : int ref list = [{contents = 5;}; {contents = 3;}]
```

```
> !(List.head r);;
```

```
val it : int = 5
```

```
> List.head r := !(List.head r) + 2;;
```

```
val it : unit = ()
```

```
> r;;
```

```
val it : int ref list = [{contents = 7;}; {contents = 3;}]
```

---

## Updating Reference Cells in Data Structures

Updating the contents of a reference cell will affect data structures where it is stored:

```
> let z = ref 5;;
val z : int ref = {contents = 5;}
> let r = [z;z];;
val r : int ref list = [{contents = 5;}; {contents = 5;}]
> z := !z + 1;;
val it : unit = ()
> r;;
val it : int ref list = [{contents = 6;}; {contents = 6;}]
```

Compare this with the mutable variable example! There, the *value* of `x` was stored in the list. Here, it is the *cell* `z` that is stored

---

## Why Two Types of Mutable Data?

Why are there both `mutable` variables and `ref` variables in F#?

They are stored differently. `Mutable` variables are stored on the *stack*, `ref` variables on the *heap*

This implies some restrictions on the use of `mutable` variables

---

## An Example that does not Work

A good way to use mutable data is to make them local to a function. Then the side-effects will be local, and the function is still pure. Alas, mutable variables cannot be used like this:

```
let f(x) =  
    let mutable y = 0  
    in let rec g(z) = if z = 0 then y else y <- y + 2;g(z-1)  
        in g(x)
```

```
/localhome/bjorn/unison/work/GRU/F#/test/locvar.fs(5,21): error  
FS0407: The mutable variable 'y' is used in an invalid way.  
Mutable variables cannot be captured by closures. Consider  
eliminating this use of mutation or using a heap-allocated  
mutable reference cell via 'ref' and '!'.  
.
```



---

## Using a `ref` Variable Instead

A `ref` variable works:

```
let f(x) =  
  let y = ref 0  
  in let rec g(z) = if z = 0 then !y else y := !y + 2;g(z-1)  
     in g(x)  
  
val f : int -> int
```

---

## Comparing Assignments in F# and C/C#/Java

In C/C#/Java:

```
x = x + y/z - 17
```

In F#, with mutable variables:

```
x <- x + y/z - 17
```

Very similar to C/C#/Java

In F#, with reference cells:

```
x := !x + !y/!z - 17
```

The main difference is that F# makes a difference between the cell itself (`x`) and the value it contains (`!x`)

---

# Arrays

Arrays are mutable in F#

Array elements can be updated similarly to mutable record fields:

```
let a = [|1; 3; 5|]  
a.[1] <- 7 + a.[1]
```

**Now, a = [|1; 10; 5|]**

---

## Control Structures in F#

F# has conditionals and loops

The conditional statement is just the usual `if-then-else`:

```
if b then s1 else s2
```

It first evaluates `b`, then `s1` or `s2` depending on the outcome of `b`

If side effects are added, then this is precisely how an imperative `if-then-else` should work

```
If s : unit, then
```

```
if b then s
```

is allowed, and is then equivalent to

```
if b then s else ()
```

---

## While Loops

F# has a quite conventional `while` loop construct:

```
while b do s
```

`s` must have type `unit`, and `while b do s` then also has type `unit`

An example:

```
let x = ref 3
while !x > 0 do
    printf "x=%d\n" !x
    x := !x - 1
```

Resulting printout:

x=3

x=2

x=1

---

## Simple For Loops

The simplest kind of for loop:

```
for v = start to stop do s
for v = start downto stop do s
```

The first form increments  $v$  by 1, the second decrements it by 1

Note that  $v$  cannot be updated by the code inside the loop

```
let blahonga n =
    for i = 1 to n do printf "Blahonga!\n"
```

---

## Iterated For Loops

These loops are iterated over the elements of a sequence (or list, or array). They have this general format:

```
for pat in sequence do s
```

The pattern `pat` is matched to each element in `sequence`, and `s` is executed for each matching in the order of the sequence

---

## Simple For Loops as Iterated For Loops

The simplest patterns are variables, and the simplest sequences are range expressions. With them, we can easily recreate simple for loops:

```
for i in 1 .. 10 do printf "Blahonga no. %d!\n" i
for i in 10 .. (-1) .. 1 do printf "Blahonga no. %d!\n" i
```

Also with non-unit stride:

```
for i in 1 .. 2 .. 10 do printf "Blahonga no. %d!\n" i
for i in 10 .. (-3) .. 1 do printf "Blahonga no. %d!\n" i
```



---

## More General Iterated Loops

More general use of patterns and sequences (lists, arrays) to iterate over:

```
for Some x in [Some 1; None; Some 2; Some 2] do printf "%d" x
```

Only the matching elements are selected. Printout will be “122”

```
let squares = seq { for i in 1 .. 100 -> (i,i*i) }
let sum = ref 0
let sum2 = ref 0
for (i,i2) in squares do
    sum := !sum + i
    sum2 := !sum2 + i2
printf "Sum = %d\nSquare sum = %d\n" !sum !sum2
```

This example illustrates a mix of matched variables, which stand for values, and reference variables which stand for cells that contain values

---

## Concluding Example: Iteration is Recursion

Let's finally see how we can define our own imperative control constructs through recursion

We will define a while loop construct

This shows that iteration is just a special case of recursion!

Since `while` is already a construct in F#, we define a function `repeat` that implements a `repeat-until` construct (like `while`, but executes the loop body once before making the test)

---

Idea: define a function `repeat b s`, where `b` is a condition (type `bool`), and `s` a loop body (executed only for the side effect)

Use sequencing to make executions of arguments happen in the right order:

- first execute `s`,
- then test if `b` is true. If yes then exit, else recursively call `repeat` again, with the same arguments `b` and `s`

If `s` has side effects, and `b` depends on these, the recursion can still terminate

---

## Repeat, First Attempt

Let's implement this idea right off:

```
let rec repeat b s = s; if b then () else repeat b s
repeat : bool -> unit -> unit
```

However, this solution has a problem! Consider this:

```
let n = ref 3
repeat (!n = 0) (printf "n=%d\n" !n; n := !n - 1)
```

If we evaluate the above then we get the printout “n=3”, and then the evaluation goes into infinite recursion.

*Why??*

---

## Why it Went Wrong

`repeat` is a function.

F# uses call by value.

Therefore, the arguments get evaluated the first time `repeat` is called.

Subsequent argument uses will not re-evaluate them, just reuse their previous values

Therefore, the side effects of `s` will only occur *once*

`b` will *always* return the value of the *first call*  $\implies$  infinite recursion, if `true`

*How can we fix this?*

---

## Repeat, Second Attempt

A way to have the arguments re-evaluated each time they are used is to *wrap each one into a function*

A function body is re-evaluated each time the function is called

This will give us the desired effect!

The functions will be given a dummy argument

We can use the value `()` as dummy argument

We obtain

```
b : unit -> bool
s : unit -> unit
```

---

## New Solution

```
let rec repeat b s = s (); if b () then () else repeat b s
repeat : (unit -> bool) -> (unit -> unit) -> unit
```

If we define

```
let n = ref 3
let b1 = (fun () -> !n = 0)
let s1 = (fun () -> printf "n=%d\n" !n; n := !n - 1)
```

And evaluate `repeat b1 s1` we get (in `fsi`):

```
n=3
n=2
n=1
val it : unit = ()
```