Imperative programming in F#

Jon Sporring

July 3, 2016

1 Introduction

Imperativ programming focusses on *how* a problem is to be solved as a list of *statements* and and a set of *states*, where states may change over time. An example is a baking recipe, e.g.,

1. Mix yeast with water
2. Stir in salt, oil, and flour
3. Knead until the dough has a smooth surface
4. Let the dough rise until it has double size
5. Shape dough into a loaf
6. Let the loaf rise until double size
7. Bake in oven until the bread is golden brown

Each line in this example consists of one or more statements that are to be executed, and while executing them states such as size of the dough, color of the bread changes, and some execution will halt execution until certain conditions of these states are fulfilled, e.g., the bread will not be put into the oven for baking before it has risen sufficiently.

Statements may be grouped into procedures, and structuring imperative programs heavily into procedures is called *Procedural programming*, which is sometimes considered as a seperate paradigm from imperative programming. *Object oriented programming* is an extension of imperative programming, where statements and states are grouped into classes and will be treated elsewhere.

Almost all computer hardware is designed for *machine code*, which is a common term used for many low-level computer programming language, and almost all machine langues follow the imperative programming paradigm.

Functional programming may be considered a subset of imperative programming, in the sense that functional programming does not include the concept of a state, or one may think of functional programming as only have one unchanging state. Functional programming has also a bigger focus on *what* should be solved, by declaring rules but not explicitly listing statements describing how these rules should be combined and executed in order to solve a given problem. Functional programming will be treated elsewhere.

2 Mutable Data

The most common syntax for a value definition is

```
let [mutable] ident [: type] = expr in expr
```

or alternatively

```
let [mutable] ident [: type] = expr [in]
  expr
```

In the above, `ident` may be replaced with a more complicated pattern, but this is outside the scope of this text. If a value has been defined as mutable, then it's value may be changed using the following syntax,

```
expr <- expr
```

Mutable data is synonymous with the term *variable*. A variable an area in the computers working memory associated with a name and a type, and this area may be read from and written to during program execution. For example,

mutableAssignReassing.fsx

```
let mutable x = Unchecked.defaultof<int> // Declare a variable x of
    type int and assign the corresponding default value to it.
printfn "%d" x
x <- 5 // Assign a new value 5 to x
printfn "%d" x
x <- -3 // Assign a new value -3 to x
printfn "%d" x
```

```
> fsharpi mutableAssignReassing.fsx
0
5
-3
```

Here a area in memory was denoted `x`, declared as type integer and assigned a default value. Later, this value of of `x` was replaced with another integer and yet another integer. The operator '`<-`' is used to distinguish the statement from the mathematical concept of equality. A short-hand for the above is available as,

mutableAssignReassingShort.fsx

```
let mutable x = 5 // Declare a variable x and assign the value 5 to it
printfn "%d" x
x <- -3 // Assign a new value -3 to x
printfn "%d" x
```

```
> fsharpi mutableAssignReassingShort.fsx
5
-3
```

where the assignment of the default value was skipped, and the type was infered from the assignment operation. However, it's important to note, that when the variable is declared, then the '`=`' operator must be used, while later reassingments must use the '`<-`' operator. Type mismatches will result in an error,

mutableAssignReassingTypeError.fsx

```
let mutable x = 5 // Declare a variable x and assign the value 5 to it
printfn "%d" x
x <- -3.0 // This is illegal, since -3.0 is a float, while x is of type
    int
printfn "%d" x
```

```
> fsharpi mutableAssignReassingTypeError.fsx


/Users/sporring/Desktop/fsharpNotes/mutableAssignReassingTypeError.fsx
   (3,6): error FS0001: This expression was expected to have type
    int
but here has type
    float
```

A typical variable is a counter of type integer, and a typical use of counters is to increment them, i.e., reassing a new value to be one more that its previous value. For example,

mutableAssignIncrement.fsx

```
let mutable x = 5 // Declare a variable x and assign the value 5 to it
printfn "%d" x
x <- x + 1 // Assign a new value -3 to x
printfn "%d" x
```

```
> fsharpi mutableAssignIncrement.fsx
5
6
```

An function that elegantly implements the incrementation operation may be constructed as,

mutableAssignIncrementEncapsulation.fsx

```
let incr =
  let mutable counter = 0
  fun () ->
    counter <- counter + 1
    counter
printfn "%d" (incr ())
printfn "%d" (incr ())
printfn "%d" (incr ())
```

```
> fsharpi mutableAssignIncrementEncapsulation.fsx
1
2
3
```

[**Jon says: Explain why this works!**] Here the output of `incr` is an anonymous function, that takes no argument, increments the variable of `incr` and returns the new value of the counter. This construction is called *encapsulation*, since the variable `counter` is hidden by the function `incr` from the user, i.e., the user need not be concerned with how the increment operator is implemented and the variable name used by `incr` does not clutter the scope where it is used.

Variables cannot be returned from functions, that is,

mutableAssignReturnValue.fsx

```
let g () =
  let x = 0
  x
printfn "%d" (g ())
```

```
> fsharpi mutableAssignReturnValue.fsx
0
```

declares a function that has no arguments and returns the value 0, while the same for a variable is illegal,

mutableAssignReturnVariable.fsx

```
let g () =
  let mutual x = 0
  x
printfn "%d" (g ())
```

```
> fsharpi mutableAssignReturnVariable.fsx


/Users/sporring/Desktop/fsharpNotes/mutableAssignReturnVariable.fsx
   (3,3): error FS0039: The value or constructor 'x' is not defined
```

There is a workaround for this by using *reference cells* by the build-in function `ref` and operators `!` and `:=`,

mutableAssignReturnRefCell.fsx

```
let g () =
  let x = ref 0
  x
let y = g ()
printfn "%d" !y
y := 3
printfn "%d" !y
```

```
> fsharpi mutableAssignReturnRefCell.fsx
0
3
```

That is, the `ref` function creates a reference variable, the '`!`' and the '`:=`' operators reads and writes its value. Reference cells are in some language called pointers, and their use is strongly discouraged, since they may cause *side-effects*, which is the effect that one function changes the state of another, such as the following example demonstrates,

mutableAssignReturnSideEffect.fsx

```
let updateFactor factor =
  factor := 2

let multiplyWithFactor x =
  let a = ref 1
  updateFactor a
  !a * x

printfn "%d" (multiplyWithFactor 3)
```

```
> fsharpi mutableAssignReturnSideEffect.fsx
6
```

In the example, the function `updateFactor` changes a variable in the scope of `multiplyWithFactor`, which is prone to errors, since the style of programming does not follow the usual assignment syntax. Better style of programming is,

mutableAssignReturnWithoutSideEffect.fsx

```
let updateFactor () =
  2

let multiplyWithFactor x =
  let a = ref 1
  a := updateFactor ()
  !a * x

printfn "%d" (multiplyWithFactor 3)
```

```
> fsharpi mutableAssignReturnWithoutSideEffect.fsx
6
```

Here there can be no doubt in `multiplyWithFactor` that the value of 'a' is changing. Side-effects do have their use, but should in general be avoided at almost all costs, and in general it is adviced to refrain from using ref cells.

3 Procedures

A *procedure* is a generalisation of the concept of functions, and in contrast to functions procedures need not return values. An example, we've already seen is the `printfn`, which is used to print text on the console, but does not return a value. Conincidentally, since the console is a state, printing to it is a side-effect. Above we examined

```
let updateFactor factor =
  factor := 2
```

which also does not have a return value. Procedural thinking is useful for encapsulation, but is prone to side-effects and should be minimized by being replaced by functional thinking.

4 Control Flow

4.1 Conditional expressions

```
if expr then expr
[{elif expr then expr}
else expr]
```

A basic flow control mechanism used both for functional and imperative programming is the `if-then-else` construction, e.g.,

flowIfThen.fsx

```
let printOnlyPostiveValues x =
  if x > 0 then
    printfn "%d" x
printOnlyPostiveValues 3
printOnlyPostiveValues -3
```

```
> fsharpi flowIfThen.fsx
3
```

I.e., if and only if the value of the argument is postive, then it will be printed on screen. More common is to include the `else`

flowIfThenElse.fsx

```
let abs x =
  if x < 0 then
    -x
  else
    x
printfn "%d" (abs 3)
printfn "%d" (abs -3)
```

```
> fsharpi flowIfThenElse.fsx
3
3
```

A common construction is a nested list of `if-then-else`,

flowIfThenElseNested.fsx

```
let digitToString x =
  if x < 1 then
    '0'
  else
    if x < 2 then
      '1'
    else
      '2'

printfn "%c" (digitToString 1)
printfn "%c" (digitToString 3)
printfn "%c" (digitToString -3)
```

```
> fsharpi flowIfThenElseNested.fsx
1
2
0
```

where the integers 0–2 are converted to characters, and integers outside this domain is converted to the nearest equivalent number. This construction is so common that a short-hand notation exists, and we may equivalently have written,

flowIfThenElseNestedShort.fsx

```
let digitToString x =
  if x < 1 then
    '0'
  elif x < 2 then
    '1'
  else
    '2'

printfn "%c" (digitToString 1)
printfn "%c" (digitToString 3)
printfn "%c" (digitToString -3)
```

```
> fsharpi flowIfThenElseNestedShort.fsx
1
2
0
```

A major difference between functional and imperative programming is how loops are expressed. Consider the problem of printing the numbers 1 to 5 on the console with a `while` loop can be done as follows,

flowWhile.fsx

```
let mutable i = 1
while i <= 5 do
  printf "%d " i
  i <- i + 1
printf "\n"
```

```
> fsharpi flowWhile.fsx
1 2 3 4 5
```

where the same result by recursion as

flowWhileRecursion.fsx

```
let rec prt a b =
  if a <= b then
    printf "%d " a
    prt (a + 1) b
  else
    printf "\n"
prt 1 5
```

```
> fsharpi flowWhileRecursion.fsx
1 2 3 4 5
```

The counting example is so often used that a special notation is availabel, the `for` loop, where the above could be implemented as

flowFor.fsx

```
for i = 1 to 5 do
  printf "%d " i
printf "\n"
```

```
> fsharpi flowFor.fsx
1 2 3 4 5
```

Note that `i` is a value and not a variable here. For a more complicated example, consider the problem of calculating average grades from a list of courses and grades. Using the above construction, this could be performed as,

flowForListsIndex.fsx

```
let courseGrades =
    ["Introduction to programming", 95;
    "Linear algebra", 80;
    "User Interaction", 85;]

let mutable sum = 0;
let mutable n = 0;
for i = 0 to (List.length courseGrades) - 1 do
  let (title, grade) = courseGrades.[i]
  printfn "Course: %s, Grade: %d" title grade
  sum <- sum + grade;
  n <- n + 1;
let avg =  (float sum) / (float n)
printfn "Average grade: %g" avg
```

```
> fsharpi flowForListsIndex.fsx
Course: Introduction to programming, Grade: 95
Course: Linear algebra, Grade: 80
Course: User Interaction, Grade: 85
Average grade: 86.6667
```

However, an elegant alternative is available as

flowForLists.fsx

```
let courseGrades =
    ["Introduction to programming", 95;
    "Linear algebra", 80;
    "User Interaction", 85;]

let mutable sum = 0;
let mutable n = 0;
for (title, grade) in courseGrades do
  printfn "Course: %s, Grade: %d" title grade
  sum <- sum + grade;
  n <- n + 1;
let avg =  (float sum) / (float n)
printfn "Average grade: %g" avg
```

```
> fsharpi flowForLists.fsx
Course: Introduction to programming, Grade: 95
Course: Linear algebra, Grade: 80
Course: User Interaction, Grade: 85
Average grade: 86.6667
```

This to be prefered, since we completely can ignore list boundary conditions and hence avoid out of range indexing. For comparison see a recursive implementation of the same,

flowForListsRecursive.fsx

```
let courseGrades =
    ["Introduction to programming", 95;
    "Linear algebra", 80;
    "User Interaction", 85;]

let rec printAndSum lst =
  match lst with
    | (title, grade)::rest ->
      printfn "Course: %s, Grade: %d" title grade
      let (sum, n) = printAndSum rest
      (sum + grade, n + 1)
    | _ -> (0, 0)
let (sum, n) = printAndSum courseGrades
let avg = (float sum) / (float n)
printfn "Average grade: %g" avg
```

```
> fsharpi flowForListsRecursive.fsx
Course: Introduction to programming, Grade: 95
Course: Linear algebra, Grade: 80
Course: User Interaction, Grade: 85
Average grade: 86.6667
```

Note how this implementation avoids the use of variables in contrast to the previous examples.

5 Arrays

Roughly speaking, arrays are mutable lists, and may be created and indexed in the same manner, e.g.,

arrayCreation.fsx

```
let A = [| 1; 2; 3; 4; 5 |]
let B = [| 1 .. 5 |]
let C = [| for a in 1 ..5 do yield a |]

let printArray (a : int array) =
  for i = 0 to a.Length - 1 do
    printf "%d " a.[i]
  printf "\n"

printArray A
printArray B
printArray C
```

```
> fsharpi arrayCreation.fsx
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

Notice that as for lists, arrays are indexed starting with 0, and that in this particular case it was necessary to specify the type of the argument for `printArray` as an array of integers with the `array` keyword. The `array` keyword is synomous with '`[]`'. Arrays do not support pattern matching, cannot be resized, but are mutable,

arrayReassign.fsx

```
let A = [| 1; 2; 3; 4; 5 |]

let printArray (a : int array) =
  for i = 0 to a.Length - 1 do
    printf "%d " a.[i]
  printf "\n"

let square (a : int array) =
  for i = 0 to a.Length - 1 do
    a.[i] <- a.[i] * a.[i]

printArray A
square A
printArray A
```

```
> fsharpi arrayReassign.fsx
1 2 3 4 5
1 4 9 16 25
```

Notice that in spite the missing `mutable` keyword, the function `square` still had the side-effect of squaring alle entries in `A`. Arrays support *slicing*, that is, indexing an array with a range results in a copy of array with values corresponding to the range, e.g.,

arraySlicing.fsx

```
let A = [| 1; 2; 3; 4; 5 |]
let B = A.[1..3]
let C = A.[..2]
let D = A.[3..]

let printArray (a : int array) =
  for i = 0 to a.Length - 1 do
    printf "%d " a.[i]
  printf "\n"

printArray A
printArray B
printArray C
printArray D
```

```
> fsharpi arraySlicing.fsx
1 2 3 4 5
2 3 4
1 2 3
4 5
```

As illustated, the missing start or end index implies from the first or to the last element.

There are quite a number of built-in procedures for all arrays some of which we summarize in Table 1. Thus, the `arrayReassign.fsx` program can be written using arrays as,

append	Creates an array that contains the elements of one array followed by the elements of another array.
average	Returns the average of the elements in an array.
blit	Reads a range of elements from one array and writes them into another.
choose	Applies a supplied function to each element of an array. Returns an array that contains the results x for each element for which the function returns Some(x).
collect	Applies the supplied function to each element of an array, concatenates the results, and returns the combined array.
concat	Creates an array that contains the elements of each of the supplied sequence of arrays.
copy	Creates an array that contains the elements of the supplied array.
create	Creates an array whose elements are all initially the supplied value.
empty	Returns an empty array of the given type.
exists	Tests whether any element of an array satisfies the supplied predicate.
fill	Fills a range of elements of an array with the supplied value.
filter	Returns a collection that contains only the elements of the supplied array for which the supplied condition returns true.
find	Returns the first element for which the supplied function returns true. Raises System.Collections.Generic.KeyNotFoundException if no such element exists.
findIndex	Returns the index of the first element in an array that satisfies the supplied condition. Raises System.Collections.Generic.KeyNotFoundException if none of the elements satisfy the condition.
fold	Applies a function to each element of an array, threading an accumulator argument through the computation. If the input function is f and the array elements are i0...iN, this function computes f (...(f s i0)...) iN.
foldBack	Applies a function to each element of an array, threading an accumulator argument through the computation. If the input function is f and the array elements are i0...iN, this function computes f i0 (...(f iN s)).
forall	Tests whether all elements of an array satisfy the supplied condition.
isEmpty	Tests whether an array has any elements.
iter	Applies the supplied function to each element of an array.
length	Returns the length of an array. The System.Array.Length property does the same thing.
map	Creates an array whose elements are the results of applying the supplied function to each of the elements of a supplied array.
max	Returns the largest of all elements of an array. Operators.max is used to compare the elements.
min	Returns the smallest of all elements of an array. Operators.min is used to compare the elements.
ofList	Creates an array from the supplied list.
ofSeq	Creates an array from the supplied enumerable object.
partition	Splits an array into two arrays, one containing the elements for which the supplied condition returns true, and the other containing those for which it returns false.
rev	Reverses the order of the elements in a supplied array.
sort	Sorts the elements of an array and returns a new array. Operators.compare is used to compare the elements.
sub	Creates an array that contains the sup¡plied subrange, which is specified by starting index and length.
sum	Returns the sum of the elements in the array.
toList	Converts the supplied array to a list.
toSeq	Views the supplied array as a sequence.
unzip	Splits an array of tuple pairs into a tuple of two arrays.
zip	Combines two arrays into an array of tuples that have two elements. The two arrays must have equal lengths; otherwise, System.ArgumentException is raised.

Table 1: Some built-in procedures in the Array module for arrays (from `https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/fsharp-core-library-reference`)

arrayReassignModule.fsx

```
let A = [| 1 .. 5 |]

let printArray (a : int array) =
  Array.iter (fun x -> printf "%d " x) a
  printf "\n"

let square a = a * a

printArray A
let B = Array.map square A
printArray A
printArray B
```

```
> fsharpi arrayReassignModule.fsx
1 2 3 4 5
1 2 3 4 5
1 4 9 16 25
```

and the `flowForListsIndex.fsx` program can be written using arrays as,

flowForListsIndexModule.fsx

```
let courseGrades =
    ["Introduction to programming", 95;
    "Linear algebra", 80;
    "User Interaction", 85;]

let A = Array.ofList courseGrades
let printCourseNGrade (title, grade) =
  printfn "Course: %s, Grade: %d" title grade
Array.iter printCourseNGrade A
let (titles,grades) = Array.unzip A
let avg =  (float (Array.sum grades)) / (float grades.Length)
printfn "Average grade: %g" avg
```

```
> fsharpi flowForListsIndexModule.fsx
Course: Introduction to programming, Grade: 95
Course: Linear algebra, Grade: 80
Course: User Interaction, Grade: 85
Average grade: 86.6667
```

Both cases avoid the use of variables and side-effects which is a big advantage for code safety.

Higher dimensional arrays can be created as arrays of arrays (of arrays ...). These are known as *jagged arrays*, since there is no inherent control of that all sub-arrays are of similar size. E.g., the following is a jagged array of increasing width,

arrayJagged.fsx

```
let A = [| for n in 1..3 do yield [| 1 .. n |] |]

let printArrayOfArrays (a : int array array) =
  for i = 0 to a.Length - 1 do
    for j = 0 to a.[i].Length - 1 do
      printf "%d " a.[i].[j]
    printf "\n"

printArrayOfArrays A
```

```
> fsharpi arrayJagged.fsx
1
1 2
1 2 3
```

Indexing arrays of arrays is done sequentially, in the sense that in the above example, the number of outer arrays is `a.Length`, `a.[i]` is the i'th array, the length of the i'th array is `a.[i].Length`, and the j'th element of the i'th array is thus `a.[i].[j]`. Often 2 dimensional square arrays are used, which can be implemented as a jagged array as,

arrayJaggedSquare.fsx

```
let pownArray  (a : int array) p =
  for i = 0 to a.Length - 1 do
    a.[i] <- pown a.[i] p
  a

let A = [| for n in 1..3 do yield (pownArray [| 1 .. 4 |] n ) |]

let printArrayOfArrays (a : int array array) =
  for i = 0 to a.Length - 1 do
    for j = 0 to a.[i].Length - 1 do
      printf "%2d " a.[i].[j]
    printf "\n"

printArrayOfArrays A
```

```
> fsharpi arrayJaggedSquare.fsx
 1  2  3  4
 1  4  9 16
 1  8 27 64
```

In fact, square arrays of dimensions 2 to 4 are so common that fsharp has built-in modules for their support. In the following describe Array2D. The workings of Array3D and Array4D are very similar. An example of creating the same 2 dimensional array as above but as an `Array2D` is,

blit	Reads a range of elements from one array and writes them into another.
copy	Creates an array that contains the elements of the supplied array.
create	Creates an array whose elements are all initially the supplied value.
iter	Applies the supplied function to each element of an array.
length1	Returns the length of an array in the first dimension.
length2	Returns the length of an array in the second dimension.
map	Creates an array whose elements are the results of applying the supplied function to each of the elements of a supplied array.

Table 2: Some built-in procedures in the Array2D module for arrays (from `https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/fsharp-core-library-reference`)

array2D.fsx

```
let A = Array2D.create 3 4 0
for i = 0 to (Array2D.length1 A) - 1 do
  for j = 0 to (Array2D.length2 A) - 1 do
    A.[i,j] <- pown (j + 1) (i + 1)

let printArray2D (a : int [,]) =
  for i = 0 to (Array2D.length1 a) - 1 do
    for j = 0 to (Array2D.length2 a) - 1 do
      printf "%2d " a.[i, j]
    printf "\n"

printArray2D A
```

```
> fsharpi array2D.fsx
 1   2   3   4
 1   4   9  16
 1   8  27  64
```

Notice that the indexing uses a slightly different notation '`[,]`' and the length functions are also slightly different. The statement `A.Length` would return the total number of elements in the array, in this case 12.

There are a bit few built-in procedures for 2 dimensional array types, some of which are summarized in Table 2

6 Mutable Collections

`List`, `LinkedList`, `Stack`, `Queue`, `HashSet`, and `Dictionary` from `System.Collections.Generic`

7 Basic I/O

Reading and writing to files and the console window.

8 Exception Handling

Exception handling allows programmers to catch and handle errors whenever an application enters an invalid state.