Learning to program with F#

Jon Sporring

July 28, 2016

Contents

Chapter 7

Controlling program flow

Non-recursive functions encapsulates code and allows for some control of flow, that is, if there is a piece of code, which we need to to have executed many times, then we can encapsulate it in the body of a function, and then call the function several times. In this chapter, we will look at more general control of flow via loops, conditional execution, and recursion, and therefore we look at further extension of the `expr` rule,

```
pat = const | ...
guard = "when" expr
rule = pat [guard] -> expr
rules = "|" rule | "|" rule rules (* first "|'' is optional' *)
expr = ...
  | "for " pat " in " expr " do " expr [" done "] (* for expression *)
  | "for " var "=" expr " to " expr " do " expr [" done "] (* simple for
      expression *)
  | "while " expr " do " expr [" done "] (* while expression *)
  | "if " expr " then " expr {" elif " expr " then " expr} " else " expr (*
      conditional expression *)
  | "match " expr " with " rules (* match expression *)
  | "function " rules (* matching function expression *)
  | "let " rec function-or-value-defns (* recursive definition *)
  | ...
```

7.1 For and while loops

Many programming constructs need to be repeated. The most basic example is counting, e.g., from 1 to 10 with a *for*-loop, · for

```
> for i = 1 to 10 do
-   printf "%d " i
- printfn "";;
1 2 3 4 5 6 7 8 9 10

val it : unit = ()
```

Listing 7.1: fsharpi, Counting from 1 to 10 using a `for`-loop.

As this interactive script demonstrates, the identifier `i` takes all the values between 1 and 10, but in spite of its changing state, it is not mutable. Note also that the return value of the `for` expression is `()` like the `printf` functions. The `for` and `while` loops follow the syntax,

```
pat = const | ...
expr = ...
```

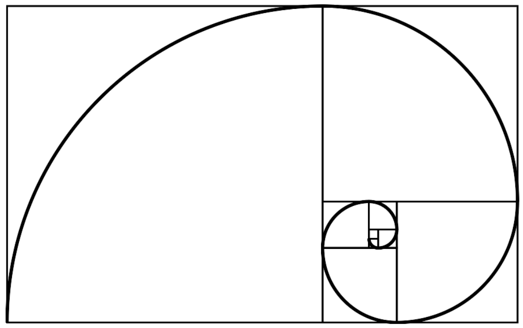

Figure 7.1: The fibonacci spiral is an approximation of the golden spiral. Each square has side lengths of successive fibonacci numbers, and the curve in each square is the circular arc with radius of the square it is drawn in. Figure by Dicklyon `https://commons.wikimedia.org/w/index.php?curid=3730979`

```
  | "for " pat " in " expr " do " expr [" done "] (* for expression *)
  | "for " var "=" expr " to " expr " do " expr [" done "] (* simple for
     expression *)
  | "while " expr " do " expr [" done "] (* while expression *)
 | ...
```

Using lightweight syntax the script block between the `do` and `done` keywords may be replaced by a newline and indentation, e.g.,

```
for i = 1 to 10 do
  printf "%d " i
printfn ""
```

```
1 2 3 4 5 6 7 8 9 10
```

Listing 7.2: countLightweight.fsx - Counting from 1 to 10 using a `for`-loop.

A more complicated example is,

> Write a program that prints the n'th fibonacci number.

The fibonnacci numbers is the series of numbers $1, 1, 2, 3, 5, 8, 13 \ldots$, where the $\mathrm{fib}(n) = \mathrm{fib}(n-1) + \mathrm{fib}(n-2)$, and they are related to Golden spirals shown in Figure 7.1. We could solve this problem with a `for`-loop as follows,

```
let fib n =
  let mutable a = 1
  let mutable b = 1
  let mutable f = 0
  for i = 3 to n do
    f <- a + b
    a <- b
    b <- f
  f

printfn "fib(1) = 1"
printfn "fib(2) = 1"
for i = 3 to 10 do
  printfn "fib(%d) = %d" i (fib i)
```

```
fib(1) = 1
fib(2) = 1
```

```
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
```

Listing 7.3: fibFor.fsx - The n'th fibonacci number as the sum of the previous 2 numbers, which are sequentially updated from 3 to n.

The basic idea of the solution is that if we are given the $(n-1)$'th and $(n-2)$'th numbers, then the n'th number is trivial to compute. And assume that fib(1) and fib(2) are given, then it is trivial to calculate the fib(3). Now we have the first 3 numbers, so we disregard fib(1) and calculate fib(4) from fib(2) and fib(3), and this process continues until we have reached the desired fib(n)
For the alternative for-loop, consider the problem,

> Write a program that identifies prime factors of a given integer n.

Prime numbers are integers divisible only be 1 and themselves with zero remainder. Let's assume that we already have identified a list of primes from 2 to n, then we could write a program that checks the remainder as follows,

```
let primeFactorCheck n =
  printfn "%d %% i = 0?" n
  for i in [2; 3; 5; 7; 11; 13; 17] do
    printfn "i = %d? %b" i (n%i = 0)
  ()

primeFactorCheck 10
```

```
10 % i = 0?
i = 2? true
i = 3? false
i = 5? true
i = 7? false
i = 11? false
i = 13? false
i = 17? false
```

Listing 7.4: primeCheck.fsx - Checking whether a given number has remainder zero after division by some low prime numbers.

In this example, the variable i runs through the elements of a list, which will be discussed in further detail in Chapter 8.
****
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,

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

```
1 2 3 4 5
```

Listing 7.5: flowWhile.fsx -

where the same result by recursion as

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

```
1 2 3 4 5
```

Listing 7.6: flowWhileRecursion.fsx -

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

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

```
1 2 3 4 5
```

Listing 7.7: flowFor.fsx -

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,

```
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
```

```
Course: Introduction to programming, Grade: 95
Course: Linear algebra, Grade: 80
Course: User Interaction, Grade: 85
Average grade: 86.6667
```

Listing 7.8: flowForListsIndex.fsx -

However, an elegant alternative is available as

```
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
```

```
Course: Introduction to programming, Grade: 95
Course: Linear algebra, Grade: 80
Course: User Interaction, Grade: 85
Average grade: 86.6667
```

Listing 7.9: flowForLists.fsx -

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

```
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
```

```
Course: Introduction to programming, Grade: 95
Course: Linear algebra, Grade: 80
Course: User Interaction, Grade: 85
Average grade: 86.6667
```

Listing 7.10: flowForListsRecursive.fsx -

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

7.2 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.,

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

```
3
```

Listing 7.11: flowIfThen.fsx -

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`

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

```
3
3
```

Listing 7.12: flowIfThenElse.fsx -

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

```
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)
```

```
1
2
0
```

Listing 7.13: flowIfThenElseNested.fsx -

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,

```
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)
```

```
1
2
```

```
0
```

Listing 7.14: flowIfThenElseNestedShort.fsx -

7.3 Pattern matching

Often functions are needed, that performs different calculations based on the input values. E.g., counting items in the english language requires various forms depending on the number, so we would say “I have 1 apple” and “I have 2 apples”. For this we may use the `match-with` programming construct, and a function that given a number returns a string on propper form could look like,

```
let applesIHave n =
  match n with
    | 0 -> "I have no apples"
    | 1 -> "I have 1 apple"
    | _ -> "I have " + (string n) + " apples"

printfn "%A" (applesIHave 0)
printfn "%A" (applesIHave 1)
printfn "%A" (applesIHave 2)
printfn "%A" (applesIHave 10)
```

```
"I have no apples"
"I have 1 apple"
"I have 2 apples"
"I have 10 apples"
```

Listing 7.15: matchWith.fsx - Using the `match-with` programming construct to vary calculation based on the input value.

This is an example of controlling programming flow, which will be discussed in more depth in Chapter 7.

```
expr = ... | "match " expr " with " rules
rule = pat [guard] -> expr
guard = "when" expr
pat = const | ...
```

Functions may be declared using pattern matching, which is a flexible method for declaring output depending on conditions on the input value. The most common pattern matching method is by use of the `match with` syntax,

```
let rec factorial n =
  match n with
  | 0 -> 1
  | 1 -> 1
  | _ -> n * (factorial (n - 1))

printfn "%d" (factorial 5)
```

```
120
```

Listing 7.16: functionDeclarationMatchWith.fsx -

A short-hand only for functions of 1 parameter is the `function` syntax,

```
let rec factorial = function
  | 0 -> 1
  | 1 -> 1
  | n -> n * (factorial (n - 1))
```

```
printfn "%d" (factorial 5)
```

```
120
```

Listing 7.17: functionDeclarationFunction.fsx -

Note that the name given in the match, here `n`, is not used in the first line, and is arbitrary at the line of pattern matchin, and may even be different on each line. For these reasons is this syntax discouraged.

7.4 Recursive functions

[1]

...

[1] **Recursive functions here.**

Bibliography

[1] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[2] Programming Research Group. Specifications for the ibm mathematical formula translating system, fortran. Technical report, Applied Science Division, International Business Machines Corporation, 1954.

[3] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[4] X3: ASA Sectional Committee on Computers and Information Processing. American standard code for information interchange. Technical Report ASA X3.4-1963, American Standards Association (ASA), 1963. `http://worldpowersystems.com/projects/codes/X3.4-1963/`.

[5] George Pólya. *How to solve it*. Princeton University Press, 1945.

Index