Chapter 7

In-code documentation

Documentation is a very important part of writing programs, since it is most unlikely, that you will be writing really obvious code. And what seems obvious at the point of writing may be mystifying months later to the author and to others. The documentation serves several purposes:

1. Communicate what the code should be doing
2. Highlight big insights essential for the code
3. Highlight possible conflicts and/or areas, where the code could be changed later

The essential point is that coding is a journey in problem solving, and proper documentation is an aid in understanding the solution and the journey that lead to it. Documentation is most often a mixture between in-code documentation and accompanying documents. Here, we will focus on in-code documentation, but arguably this does cause problems in multi-language environments and run the risk of bloating code.

F# has the following simplified syntax for in-code documentation,

Listing 7.1: Comments.

```
1 blockComment = "(*" {codePoint} "*)";
2 lineComment = "//" {codePoint - newline} newline;
```

That is, text framed as a `blockComment` is still parsed by F# as keywords and basic types implying that `(* a comment (* in a comment *) *)` and `(* "*)" *)` are valid comments, while `(* " *)` is invalid.[1]

The F# compiler has an option for generating *Extensible Markup Language* (*XML*) files from scripts using the C# documentation comments tags[2]. The XML documentation starts with a triple-slash `///`, i.e., a lineComment and a slash, which serves as comments for the code construct, that follows immediately after. XML consists of tags which always appears in pairs, e.g., the tag "tag" would look like `<tag> ... </tag>`. The F# accept any tags, but recommends those listed in Table 7.1. If no tags are used, then it is automatically assumed to be a `<summary>`. An example of a documented script is,

- Extensible Markup Language
- XML

[1] Jon: **lstlisting colors is bad.**

[2] For specification of C# documentations comments see ECMA-334 3rd Edition, Annex E, Section 2: `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`

Tag	Description
<c>	Set text in a code-font.
<code>	Set one or more lines in code-font.
<example>	Set as an example.
<exception>	Describe the exceptions a function can throw.
<list>	Create a list or table.
<para>	Set text as a paragraph.
<param>	Describe a parameter for a function or constructor.
<paramref>	Identify that a word is a parameter name.
<permission>	Document the accessibility of a member.
<remarks>	Further describe a function.
<returns>	Describe the return value of a function.
<see>	Set as link to other functions.
<seealso>	Generate a See Also entry.
<summary>	Main description of a function or value.
<typeparam>	Describe a type parameter for a generic type or method.
<typeparamref>	Identify that a word is a type parameter name.
<value>	Describe a value.

Table 7.1: Recommended XML tags for documentation comments, from ECMA-334 3rd Edition, Annex E, Section 2.

Listing 7.2, commentExample.fsx: Code with XML comments.

```
/// The discriminant of a quadratic equation with parameters a, b, and c
let discriminant a b c = b ** 2.0 - 4.0 * a * c

/// <summary>Find x when 0 = ax^2+bx+c.</summary>
/// <remarks>Negative discriminant are not checked.</remarks>
/// <example>
///   The following code:
///   <code>
///     let a = 1.0
///     let b = 0.0
///     let c = -1.0
///     let xp = (solution a b c +1.0)
///     printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
///   </code>
///   prints <c>0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7</c> to the console.
/// </example>
/// <param name="a">Quadratic coefficient.</param>
/// <param name="b">Linear coefficient.</param>
/// <param name="c">Constant coefficient.</param>
/// <param name="sgn">+1 or -1 determines the solution.</param>
/// <returns>The solution to x.</returns>
let solution a b c sgn =
  let d = discriminant a b c
  (-b + sgn * sqrt d) / (2.0 * a)

let a = 1.0
let b = 0.0
let c = -1.0
let xp = (solution a b c +1.0)
printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
```

```
0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 1.0
```

Mono's `fsharpc` command may be used to extract the comments into an XML file,

Listing 7.3, Converting in-code comments to XML.

```
$ fsharpc --doc:commentExample.xml commentExample.fsx
F# Compiler for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
```

This results in an XML file with the following content,

Listing 7.4, An XML file generated by `fsharpc`.

```
<?xml version="1.0" encoding="utf-8"?>
<doc>
<assembly><name>commentExample</name></assembly>
<members>
<member name="M:CommentExample.solution(System.Double,System.Double,
   System.Double,System.Double)">
 <summary>Find x when 0 = ax^2+bx+c.</summary>
 <remarks>Negative discriminant are not checked.</remarks>
 <example>
   The following code:
   <code>
     let a = 1.0
     let b = 0.0
     let c = -1.0
     let xp = (solution a b c +1.0)
     printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
   </code>
   prints <c>0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7</c> to the console.
 </example>
 <param name="a">Quadratic coefficient.</param>
 <param name="b">Linear coefficient.</param>
 <param name="c">Constant coefficient.</param>
 <param name="sgn">+1 or -1 determines the solution.</param>
 <returns>The solution to x.</returns>
</member>
<member name="M:CommentExample.discriminant(System.Double,System.Double,
   System.Double)">
<summary>
 The discriminant of a quadratic equation with parameters a, b, and c
</summary>
</member>
</members>
</doc>
```

The extracted XML is written in C# type by convention, since F# is part of the Mono and .Net framework that may be used by any of the languages using Assemblies. Besides the XML inserted in the script, the XML has added `<?xml ...>` header, `<doc>`, `<assembly>`, `<members>`, and `<member>` tags. The header and the `<doc>` tag are standards for XML. The extracted XML is geared towards documenting big libraries of codes and thus highlights the structured programming organization, see ??, and `<assembly>`, `<members>`, and `<member>` are indications for where the functions belong in the hierarchy. As an example, the prefix "`M:CommentExample.`" means that it is a method in the namespace CommentExample, which in this case is the name of the file. Further, the function type `val solution : a:float -> b:float -> c:float -> sgn:float -> float` is in the XML documentation `M:CommentExample.solution(System.Double,System.Double,System.Double,System.Double)`, which is the C# equivalent.

An accompanying program in the Mono suite is `mdoc`, whose primary use is to perform a syntax analysis of an assembly and generate a scaffold XML structure for an accompanying document. With the `-i` flag, it is further possible to include the in-code comments as initial descriptions in the XML. The XML may be updated gracefully by `mdoc` as the code develops, without destroying manually entered documentation in the accompanying documentation. Finally, the XML may be exported to HTML

The primary use of the `mdoc` command is to analyze compiled code and generate an empty XML structure with placeholders to describe functions, values, and variables. This structure can then be updated and edited as the program develops. The edited XML files can then be exported to *Hyper Text Markup Language* (*HTML*) files, which can be viewed in any browser. Using the console, all of this is accomplished by executing the commands in Listing 7.5.

- Hyper Text Markup Language
- HTML

Listing 7.5, Converting an XML file to HTML.

```
1  $ mdoc update -o commentExample -i commentExample.xml commentExample.exe
2  New Type: CommentExample
3  Member Added: public static double determinant (double a, double b,
       double c);
4  Member Added: public static double solution (double a, double b, double c
       , double sgn);
5  Member Added: public static double a { get; }
6  Member Added: public static double b { get; }
7  Member Added: public static double c { get; }
8  Member Added: public static double xp { get; }
9  Namespace Directory Created:
10 New Namespace File:
11 Members Added: 6, Members Deleted: 0
12 $ mdoc export-html -out commentExampleHTML commentExample
13 .CommentExample
```

The primary use of the `mdoc` command is to analyze compiled code and generate an empty XML structure with placeholders to describe functions, values, and variables. This structure can then be updated and edited as the program develops. The edited XML files can then be exported to HTML files, which can be viewed in any browser, an example of which is shown in Figure 7.1. A full description of how to use `mdoc` is found here[3].

[3] http://www.mono-project.com/docs/tools+libraries/tools/monodoc/generating-documentation/

solution Method

Find x when 0 = ax^2+bx+c.

Syntax

```
[Microsoft.FSharp.Core.CompilationArgumentCounts(Mono.Cecil.CustomAttributeArgument[])]
public static double solution (double a, double b, double c, double sgn)
```

Parameters

a
: Quadratic coefficient.

b
: Linear coefficient.

c
: Constant coefficient.

sgn
: +1 or -1 determines the solution.

Returns

The solution to x.

Remarks

Negative discriminant are not checked.

Example

The following code:

Example

```
let a = 1.0
let b = 0.0
let c = -1.0
let xp = (solution a b c +1.0)
printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
```

prints `0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7` to the console.

Requirements

Namespace:
Assembly: commentExample (in commentExample.dll)
Assembly Versions: 0.0.0.0

Figure 7.1: Part of the HTML documentation as produce by `mdoc` and viewed in a browser.

Chapter 8

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,

Listing 8.1: Expressions for controlling the flow of execution.

```
1  expr = ...
2    | "if" expr "then" expr {"elif" expr "then" expr} ["else" expr] (*
     conditional*)
3    | "while" expr "do" expr ["done"] (*while loop*)
4    | "for" ident "=" expr "to" expr "do" expr ["done"] (*simple for loop*)
5    | "let" functionDefn "in" expr (*binding a function or operator*)
6    | "let" "rec" functionDefn {"and" functionDefn} "in" expr (*recursive fcts
     *)
```

8.1 For and while loops

Many programming constructs need to be repeated, and F# contains many structures for repetition such as the "`for`" and "`while`" loops, which have the syntax,

Listing 8.2: `for`- and `while`-loops.

```
1  expr = ...
2    | "while" expr "do" expr ["done"] (*while loop*)
3    | "for" ident "=" expr "to" expr "do" expr ["done"] (*simple for loop*)
```

As an example, consider counting from 1 to 10 with a "*for*"-loop,

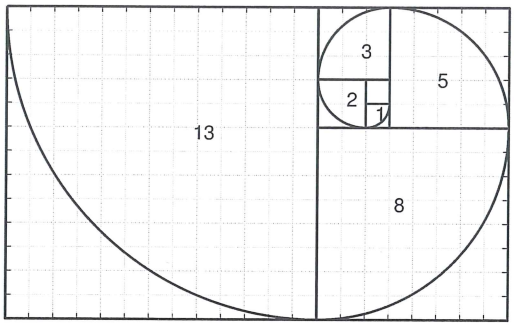

Figure 8.1: The Fibonacci spiral is an approximation of the golden spiral. Each square has side lengths of successive Fibonacci numbers, and the curve in each square is the circular arc with radius of the square it is drawn in.

Listing 8.3: Counting from 1 to 10 using a "for"-loop.

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

val it : unit = ()
```

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. Using lightweight syntax the block following the "*do*" keyword up to and including the "*done*" keyword may be replaced by a newline and indentation, e.g.,

- "do"
- "done"

Listing 8.4, countLightweight.fsx:
Counting from 1 to 10 using a "for"-loop, see Listing 8.3.

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

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

A more complicated example is,

Problem 8.1:

Write a program that calculates the n'th Fibonacci number.

The Fibonacci numbers is the series of numbers $1, 1, 2, 3, 5, 8, 13 \ldots$, where the $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$, and they are related to Golden spirals shown in Figure 8.1. We could solve this problem with a "`for`"-loop as follows,

Listing 8.5, fibFor.fsx:
The n'th Fibonacci number is the sum of the previous 2.

```
let fib n =
  let mutable prev = 1
  let mutable current = 1
  let mutable next = 0
  for i = 3 to n do
    next <- current + prev
    prev <- current
    current <- next
  next

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
```

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). For the fib(4) we only need fib(3) and fib(2), hence we may disregard fib(1). Thus we realize, that we can cyclicly update the previous, current, and next values by shifting values until we have reached the desired fib(n).

The *"while"*-loop is simpler than the "for"-loop and does not contain a builtin counter structure. Hence, if we are to repeat the count-to-10 program from Listing 8.3 example, it would look somewhat like,

Listing 8.6, countWhile.fsx:
Count to 10 with a counter variable.

```
let mutable i = 1 in while i <= 10 do printf "%d " i; i <- i + 1 done;
printf "\n"
```

```
$ fsharpc --nologo countWhile.fsx && mono countWhile.exe
./fsxeval2out: line 19: fsharpc: command not found
```

or equivalently using the lightweight syntax,

Listing 8.7, countWhileLightweight.fsx:
Count to 10 with a counter variable using lightweight syntax, see Listing 8.6.

```
let mutable i = 1
while i <= 10 do
  printf "%d " i
  i <- i + 1
printf "\n"
```

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

In this case, the "for"-loop is to be preferred, since more lines of code typically means more chances of making a mistake. But the "while"-loop allows for other logical structures. E.g., let's find the biggest Fibonacci number less than 100,

Listing 8.8, fibWhile.fsx:
Search for the largest Fibonacci number less than a specified number.

```
let largestFibLeq n =
  let mutable prev = 1
  let mutable current = 1
  let mutable next = 0
  while next <= n do
    next <- prev + current
    prev <- current
    current <- next
  prev

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

```
largestFibLeq(1) = 1
largestFibLeq(2) = 1
largestFibLeq(3) = 3
largestFibLeq(4) = 3
largestFibLeq(5) = 5
largestFibLeq(6) = 5
largestFibLeq(7) = 5
largestFibLeq(8) = 8
largestFibLeq(9) = 8
largestFibLeq(10) = 8
```

Thus, "while"-loops are most often used when the number of iteration cannot easily be decided when entering the loop.

Both "for"- and "while"-loops are often associated with variables, i.e., values that change while looping. If one mistakenly used values and rebinding, then the result would in most cases be of little use, e.g.,

Humm — the "let" expression introduces a local binding... There is no such thing as a "re-binding"!!!

Listing 8.9, forScopeError.fsx:
Lexical scope error. While rebinding is valid F# syntax, has little effect due to lexical scope.

```
let a = 1
for i = 1 to 10 do
  let a = a + 1
  printf "(%d, %d) " i a
printf "\n"
```

```
$ fsharpc --nologo forScopeError.fsx && mono forScopeError.exe
./fsxeval2out: line 19: fsharpc: command not found
```

I.e., the "let" expression rebinds a every iteration of the loop, but the value on the right-hand-side is taken lexically from above, where a has the value 1, so every time the result is the value 2.

8.2 Conditional expressions

Consider the task,*

Problem 8.2:

Write a function that, given n, writes the sentence, "I have n apple(s)", where the plural 's' is added appropriately.

For this task we need to test the value of n, and one option is to use conditional expressions. Conditional expression has the syntax, The grammar for conditional expressions is,

rephrase

Listing 8.10: Conditional expressions.

```
expr = ...
  | "if" expr "then" expr {"elif" expr "then" expr} ["else" expr] (*
   conditional*)
```

and an example using conditional expressions to solve the above problem is,

* I don't like these trailing commas. Consider rewriting the sentence to "Consider the task in Problem 8.2.

Listing 8.11, conditionalLightweight.fsx:
Using conditional expression to generate different strings.

```
let applesIHave n =
  if n < -1 then
    "I owe " + (string -n) + " apples"
  elif n < 0 then
    "I owe " + (string -n) + " apple"
  elif n < 1 then
    "I have no apples"
  elif n < 2 then
    "I have 1 apple"
  else
    "I have " + (string n) + " apples"

printfn "%A" (applesIHave -3)
printfn "%A" (applesIHave -1)
printfn "%A" (applesIHave 0)
printfn "%A" (applesIHave 1)
printfn "%A" (applesIHave 2)
printfn "%A" (applesIHave 10)
```

```
"I owe 3 apples"
"I owe 1 apple"
"I have no apples"
"I have 1 apple"
"I have 2 apples"
"I have 10 apples"
```

The `expr` following *"`if`"* and *"`elif`"* are *conditions*, i.e., expressions that evaluate to a boolean value. The `expr` following *"`then`"* and *"`else`"* are called *branches*, and all branches must have identical type, such that regardless which branch is chosen, then the type of the result of the conditional expression is the same. The result of the conditional expression is the first branch, for which its condition was true.

- "`if`"
- "`elif`"
- conditions
- "`then`"
- "`else`"
- branches

The sentence structure and its variants gives rise to a more compact solution, since the language to be returned to the user is a variant of "I have/or no/number apple(s)", i.e., under certain conditions should the sentence use "have" and "owe" etc.. So we could instead make decisions on each of these sentence parts and then build the final sentence from its parts. This is accomplished in the following example:

Listing 8.12, conditionalLightweightAlt.fsx:
Using sentence parts to construct the final sentence.

```
let applesIHave n =
  let haveOrOwe = if n < 0 then "owe" else "have"
  let pluralS = if (n = 0) || (abs n) > 1 then "s" else ""
  let number = if n = 0 then "no" else (string (abs n))

  "I " + haveOrOwe + " " + number + " apple" + pluralS

printfn "%A" (applesIHave -3)
printfn "%A" (applesIHave -1)
printfn "%A" (applesIHave 0)
printfn "%A" (applesIHave 1)
printfn "%A" (applesIHave 2)
printfn "%A" (applesIHave 10)
```

```
"I owe 3 apples"
"I owe 1 apple"
"I have no apples"
"I have 1 apple"
"I have 2 apples"
"I have 10 apples"
```

While arguably shorter, this solution is also more dense, and for a small problem like this, it is most likely more difficult to debug and maintain.

Note that both "`elif`" and "`else`" branches are optional, which may cause problems. For example, both `let a = if true then 3` and `let a = if true then 3 elif false then 4` will be invalid, since F# is not smart enough to realize that the type of the expression is uniquely determined. Instead F# looks for the "`else`" to ensure all cases have been covered, and that `a` always will be given a unique value of the same type regardless of the branches taken in the conditional statement, hence, `let a = if true then 3 else 4` is the only valid expression of the 3. In practice, F# assumes that the omitted branches returns "`()`", and thus it is fine to say `let a = if true then ()` and `if true then printfn "hej"`. Nevertheless, it is good practice in F# always to include and "`else`" branch.

8.3 Programming intermezzo

Using loops and conditional expressions we are now able to solve the following problem :

Problem 8.3:

Given an integer on decimal form, write its equivalent value on binary form .

To solve this problem, consider odd numbers: They all have the property, that the least significant bit is 1, e.g., $1_2 = 1, 101_2 = 5$ in contrast to even numbers such as $110_2 = 6$. Division by 2 is equal to right-shifting by 1, e.g., $1_2/2 = 0.1_2 = 0.5, 101_2/2 = 10.1_2 = 2.5, 110_2/2 = 11_2 = 3$. Thus by integer division by 2 and checking the remainder, we may sequentially read off the least significant bit. This leads to the following algorithm: