Learning to program with F#

Jon Sporring

Department of Computer Science,
University of Copenhagen

October 6, 2017

Contents

9 | Organising code in libraries and application programs

In this chapter, we will focus on a number of ways to make code available as *library* functions in F#. By library we mean a collection of types, values and functions that an application program can use. A library does not perform calculations on its own. · library

F# includes several programming structures to organize code in libraries: Modules, namespaces and classes. In this chapter, we will describe modules and namespaces. Classes will be described in detail in Chapter 20.

9.1 Modules

An F# *module*, not to be confused with a Common Language Infrastructure module (see Appendix D), is a programming structure used to organise type declarations, values, functions, etc. · module

Every implementation and script file in F# implicitly defines a module, and the module name is given by the filename. Thus, creating a script file `Meta.fsx` as shown in Listing 9.1.[1]

Listing 9.1 Meta.fsx:
A script file defining the `apply` function.

```
1 type floatFunction = float -> float -> float
2 let apply (f : floatFunction) (x : float) (y : float) : float = f x y
```

we've implicitly defined a module of name `Meta`. Another script file may now use this function, which is accessed using the "`.`" notation, i.e., `Meta.apply` will refer to this function in other programs. An application program could be as shown in Listing 9.3.

Listing 9.2 MetaApp.fsx:
Defining a script calling the module.

```
1 let add : Meta.floatFunction = fun x y -> x + y
2 let result = Meta.apply add 3.0 4.0
3 printfn "3.0 + 4.0 = %A" result
```

In the above, we have explicitly used the module's type definition for illustration purposes. A shorter

[1] Jon: **Type definitions have not been introduced at this point!**

and possibly simpler program would have been to define `add` as `let add x y = x + y`, since F#'s typesystem will infer the implied type. However, **explicit definitions of types is recommended for readability.** Hence, an alternative to the above's use of lambda functions is, `let add (x: float) (y: float) : float = x + y`. To compile the module and the application program, we write as demonstrated in Listing 9.3. Advice

Listing 9.3: Compiling both the module and the application code. Note that fileorder matters, when compiling several files.

```
$ fsharpc --nologo Meta.fsx MetaApp.fsx && mono MetaApp.exe
3.0 + 4.0 = 7.0
```

Notice, since the F# compiler reads through the files once, the order of the filenames in the compile command is very important. Hence, the script containing the module and function definitions must be to the left of the script containing their use. Notice also that if not otherwise specified, then the F# compiler produces an `.exe` file derived from the last filename in the list of filenames.

We may also explicitly define the module name using the *`module`* using the syntax, ·module

Listing 9.4 Outer module.

```
module <ident>
<script>
```

Here, the identifier `<ident>` is a name not necessarily related to the filename, and the script `<script>` is expression. An example is given in Listing 9.20.

Listing 9.5 MetaExplicit.fsx:
Explicit definition of the outermost module.

```
module Meta
type floatFunction = float -> float -> float
let apply (f : floatFunction) (x : float) (y : float) : float = f x y
```

Since we have created a new file, where the module `Meta` is explicitly defined, we can use the same application program. This is demonstrated in Listing 9.6.

Listing 9.6: Changing the module definition to explicit naming has no effect on the application nor the compile command.

```
$ fsharpc --nologo MetaExplicit.fsx MetaApp.fsx && mono MetaApp.exe
3.0 + 4.0 = 7.0
```

Notice that, since `MetaExplicit.fsx` explicitly defines the module name, `apply` is not available to an application program as `MetaExplicit.apply`. **It is recommended that module names are defined explicitly, since filenames may change due to external conditions.** I.e., filenames are typically set from the perspective of the filesystem. The user may choose to change names to suit a filesystem structure, or different platforms may impose different file naming convention. Thus, direct linking of filenames with the internal workings of a program is a needless complication of structure. Advice

The definitions inside a module may be accessed directly from an application program, omitting the

"."-notation, by use of the *open* keyword, · open

Listing 9.7 Open module.

```
1 open <ident>
```

I.e., we can modify `MetaApp.fsx` as shown in Listing 9.9.

Listing 9.8 MetaAppWOpen.fsx: Avoiding the "."-notation by the open keyword.

```
1 open Meta
2 let add : floatFunction = fun x y -> x + y
3 let result = apply add 3.0 4.0
4 printfn "3.0 + 4.0 = %A" result
```

In this case, the namespace of our previously defined module is included into the scope of the application functions, and its types, values, functions, etc. can be used directly, as shown in Listing 9.9.

Listing 9.9: How the application program opens the module has no effect on the module code nor compile command.

```
1 $ fsharpc --nologo MetaExplicit.fsx MetaAppWOpen.fsx && mono
     MetaAppWOpen.exe
2 3.0 + 4.0 = 7.0
```

The open-keyword should be used sparingly, since including a library's definitions into the application scope can cause surprising naming conflicts, since the user of a library typically has no knowledge of the inner workings of the library. E.g., the user may accidentally use code defined in the library, but with different type and functionality than intended, which the type system will use to deduce types in the application program, and therefore will either give syntax or runtime errors that are difficult to understand. This problem is known as *namespace pollution*, and for clarity **it is recommended to use the open-keyword sparingly**. Notice that for historical reasons, the work namespace pollution is used to cover both pollution due to modules and namespaces. · namespace pollution Advice

Modules may also be nested, in which case the nested definitions must use the "="-sign and must be appropriately indented.

Listing 9.10 Nested modules.

```
1 module <ident> = <script>
```

In lightweight syntax, a newline may be entered before the script <script>, and the script must be indented. An example is shown in Listing 9.11.

Listing 9.11 nestedModules.fsx:
Modules may be nested.

```
module Utilities
let PI = 3.1415
module Meta =
  type floatFunction = float -> float -> float
  let apply (f : floatFunction) (x : float) (y : float) : float = f x y
module MathFcts =
  let add : Meta.floatFunction = fun x y -> x + y
```

In this case, `Meta` and `MathFcts` are defined at the same level and said to be siblings, while `Utilities` is defined at a higher level. In this relation, the former two are said to be the children of the latter. Note that the nesting respects the lexical scope rules, such that the constant `PI` is directly accessible in both modules `Meta` and `MathFcts`, as is the module `Meta` in `MathFcts` but not `MathFcts` in `Meta`. The "."-notation is reused to index deeper into the module hierarchy as the example in Listing 9.12 shows.

Listing 9.12 nestedModulesApp.fsx:
Applications using nested modules require additional usage of the "." notation to navigate the nesting tree.

```

let add : Utilities.Meta.floatFunction = fun x y -> x + y
let result = Utilities.Meta.apply Utilities.MathFcts.add 3.0 Utilities.PI
printfn "3.0 + 4.0 = %A" result
```

Modules can be recursive using the *rec*-keyword, meaning that in our example we can make the outer module recursive as demonstrated in Listing 9.13.[2] · rec

Listing 9.13 nestedRecModules.fsx:
Mutual dependence on nested modules requires the rec keyword in the module definition.

```
module rec Utilities
  module Meta =
    type floatFunction = float -> float -> float
    let apply (f : floatFunction) (x : float) (y : float) : float = f x y
  module MathFcts =
    let add : Meta.floatFunction = fun x y -> x + y
```

The consequence is that the modules `Meta` and `MathFcts` are accessible in both modules, but compilation will now give a warning, since soundness of the code will first be checked at runtime. In general it is advised to **avoid programming constructions, whose validity cannot be checked at compile-time.** Advice

9.2 Namespaces

An alternative to structure code in modules is use a *namespace*, which only can hold modules and type · namespace

[2] Jon: **Dependence on version 4.1 and higher.**

declarations and only works in compiled mode. Namespaces are defined as explicitly defined outer modules using the *namespace* keyword, · namespace

Listing 9.14 Namespace.

```
namespace <ident>
<script>
```

An example is given in Listing 9.15.

Listing 9.15 namespace.fsx:
Defining a namespace is similar to explicitly named modules.

```
namespace Utilities
type floatFunction = float -> float -> float
module Meta =
  let apply (f : floatFunction) (x : float) (y : float) : float = f x y
```

Notice that when organising code in a namespace, the first line of the file, other than comments and compiler directives, must be the one starting with namespace.

As for modules, the content of a namespace is accessed using the "." notation, as demonstrated in Listing 9.17.

Listing 9.16 namespaceApp.fsx:
The "."-notation lets the application program accessing functions and types in a namespace.

```
let add : Utilities.floatFunction = fun x y -> x + y
let result = Utilities.Meta.apply add 3.0 4.0
printfn "3.0 + 4.0 = %A" result
```

Likewise, compilation is performed identically, see Listing 9.17.

Listing 9.17: Compilation of files including namespace definitions uses the same procedure as modules.

```
$ fsharpc --nologo namespace.fsx namespaceApp.fsx && mono
   namespaceApp.exe
3.0 + 4.0 = 7.0
```

Hence, from an application point of view, it is not immediately possible to see, that `Utilities` is defined as a namespace and not a module. However, in contrast to modules, namespaces may span several files. E.g., we may add a third file extending the `Utilities` namespace with the `MathFcts` module as demonstrated in Listing 9.18.

Listing 9.18 namespaceExtension.fsx: Namespaces may span several files. Here is shown an extra file, which extends the `Utilities` namespace.

```
namespace Utilities
module MathFcts =
  let add : floatFunction = fun x y -> x + y
```

To compile we now need to include all three files in the right order. Likewise, compilation is performed identically, see Listing 9.19.

Listing 9.19: Compilation of namespaces defined in several files requires careful consideration of order, since the compiler reads once and only once through the files in the order they are given.

```
$ fsharpc --nologo namespace.fsx namespaceExtension.fsx namespaceApp.fsx
   && mono namespaceApp.exe
3.0 + 4.0 = 7.0
```

The order matters since `namespaceExtension.fsx` relies on the definition of `floatFunction` in the file `namespace.fsx`. You can use extensions to extend existing namespaces included with the F# compiler.[3][4]

Namespaces may also be nested. In contrast to modules, nesting is defined using the "`.`" notation. That is, to create a child namespace `more` of `Utilities` we must use initially write `namespace Utilities.more`. Indentation is ignored in the `namespace` line, thus left-most indentation is almost always used. Namespaces follow lexical scope rules, and identically to modules, namespaces containing mutually dependent children can be declared using the `rec` keyword, e.g., `namespace rec Utilities`.

9.3 Compiled libraries

Libraries may be distributed in compiled form as `.dll` files. This saves the user for having to recompile a possibly large library every time library functions needs to be compiled with an application program. In order to produce a library file from `MetaExplicitModuleDefinition.fsx` and then compile an application program, we first use the compiler's `-a` option to produce the `.dll`. A demonstration is given in Listing 9.20.

Listing 9.20: A stand-alone `.dll` file is created and used with special compile commands.

```
$ fsharpc --nologo -a MetaExplicit.fsx
```

This produces the file `MetaExplicit.dll`, which may be linked to an application using the `-r` option during compilation, see Listing 9.21.[5]

[3] Jon: **Something about intrinsic and optional extension** `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/type-extensions`.

[4] Jon: **Perhaps something about the global namespace** `global`.

[5] Jon: **This is the MacOS option standard, Windows is slightly different.**

Listing 9.21: The library is linked to an application during compilation to produce runnable code.

```
$ fsharpc --nologo -r MetaExplicit.dll MetaApp.fsx && mono MetaApp.exe
3.0 + 4.0 = 7.0
```

A library can be the result of compiling a number of files into a single `.dll` file. `.dll`-files may be loaded dynamically in script files (`.fsx`-files) using the *`#r` directive* as illustrated in Listing 9.23.

· `#r` directive

Listing 9.22 MetaHashApp.fsx:
The `.dll` file may be loaded dynamically in `.fsx` script files and in interactive mode. Nevertheless, this usage is not recommended.

```
#r "MetaExplicit.dll"
let add : Meta.floatFunction = fun x y -> x + y
let result = Meta.apply add 3.0 4.0
printfn "3.0 + 4.0 = %A" result
```

We may now omit the explicit mentioning of the library when compiling as shown in Listing 9.23.

Listing 9.23: When using the `#r` directive, then the `.dll` file need not be explicitly included in the list of files to be compiled.

```
$ fsharpc --nologo MetaHashApp.fsx && mono MetaHashApp.exe
3.0 + 4.0 = 7.0
```

The `#r` directive is also used to include a library in interactive mode. However, for code to be compiled, the use of the `#r` directive requires that the filesystem path to the library is coded inside the script. As for module names, direct linking of filenames with the internal workings of a program is a needless complication of structure, and **it is recommended not to rely on the use of the `#r` directive.**

Advice

In the above, we have compiled *script files* into libraries. However, F# has reserved the `.fs` filename suffix for library files, and such files are called *implementation files*. In contrast to script files, implementation files do not support the `#r` directive. When compiling a list of implementation and script files, all but the last file must explicitly define a module or a namespace.

· script files
· implementation files

Both script and implementation files may be augmented with *signature files*. A signature file contains no implementation but only type definitions. Signature files offers three distinct features:

· signature files

1. Signature files can be used as part of the documentation of code, since type information is of paramount importance for an application programmer to use a library.
2. Signature files may be written before the implementation file. This allows for a higher-level programming design that focusses on *which* functions should be included and *how* they can be composed.
3. Signature files allow for access control. Most importantly, if a type definition is not available in the signature file, then it is not available to the application program. Such definitions are private and can only be used internally in the library code. More fine grained control is available relating to classes, and will be discussed in Chapter 20.

Signature files can be generated automatically using the `--sig:<filename>` compiler directive. To

demonstrate this feature, we will first move the definition of `add` to the implementation file, see Listing 9.28.

Listing 9.24 MetaWAdd.fs: An implementation file including the `add` function.

```
module Meta
type floatFunction = float -> float -> float
let apply (f : floatFunction) (x : float) (y : float) : float = f x y
let add (x : float) (y : float) : float = x + y
```

A signature file may be automatically generated as shown in Listing 9.25.

Listing 9.25: Automatic generation of a signature file at compile time.

```
$ fsharpc --nologo --sig:MetaWAdd.fsi MetaWAdd.fs

MetaWAdd.fs(4,48): warning FS0988: Main module of program is empty:
   nothing will happen when it is run
```

The warning can safely be ignored, since it is at this point not our intention to produce runnable code. The above has generated the signature file in Listing 9.28.

Listing 9.26 MetaWAdd.fsi: An automatically generated signature file from `MetaWAdd.fs`.

```
module Meta
type floatFunction = float -> float -> float
val apply : f:floatFunction -> x:float -> y:float -> float
val add : x:float -> y:float -> float
```

We can generate a library using the automatically generated signature file using `fsharpc -a MetaWAdd.fsi MetaWAdd.fs`, which is identical to compiling the `.dll` file without the signature file. However, if we remove, e.g., the type definition for `add` in the signature file, then this function becomes private to the module, and cannot be accessed outside. Hence, using the signature file in Listing 9.30 and recompiling the `.dll` as Listing 9.28 generates no error.

Listing 9.27 MetaWAddRemoved.fsi: Removing the type defintion for `add` from `MetaWAdd.fsi`.

```
module Meta
type floatFunction = float -> float -> float
val apply : f:floatFunction -> x:float -> y:float -> float
```

Listing 9.28: Automatic generation of a signature file at compile time.

```
$ fsharpc --nologo -a MetaWAddRemoved.fsi MetaWAdd.fs
```

But, when using the newly created `MetaWAdd.dll` with a modified version of Listing 9.3, which does

not itself supply a definition of `add` as shown in Listing 9.29, we get a syntax error, since `add` now is inaccessible to the application program. This is demonstrated in Listing 9.30.

Listing 9.29 MetaWOAddApp.fsx:
A version of Listing 9.3 without a definition of `add`.

```
let result = Meta.apply add 3.0 4.0
printfn "3.0 + 4.0 = %A" result
```

Listing 9.30: Automatic generation of a signature file at compile time.

```
$ fsharpc --nologo -r MetaWAdd.dll MetaWOAddApp.fsx

MetaWOAddApp.fsx(1,25): error FS0039: The value or constructor 'add' is
   not defined.
```

13 | Recursion

Recursion is a central concept in F# and used to control flow in loops without the `for` and `while` constructions. Figure 13.1 illustrates the concept of an infinite loop with recursion.

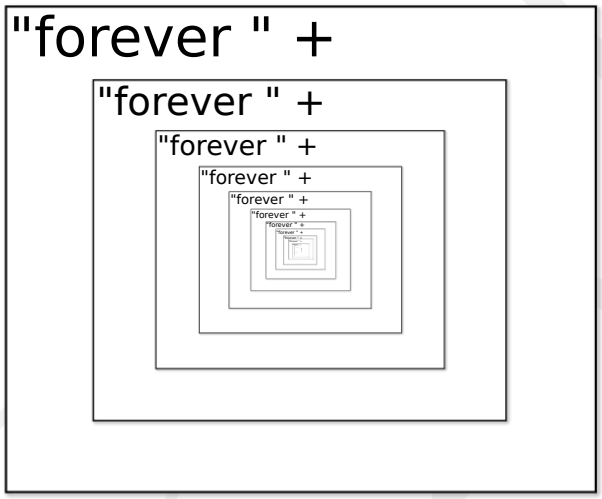


Figure 13.1: An infinitely long string of "forever forever forever...", conceptually calculated by `let rec forever () = "fsharp " + (forever ())`.

13.1 Recursive functions

A *recursive function* is a function, which calls itself, and the syntax for defining recursive functions is an extension of that for regular functions:

· recursive function

Listing 13.1 Syntax for defining one or more mutually dependent recursive functions.

```
let rec <ident> = <expr> {and <ident> = <expr>} [in] <expr>
```

From a compiler point of view, the *rec* is necessary, since the function is used before the compiler has completed its analysis. If two functions are mutually recursive, then they must be defined jointly using the *and* keyword.

· rec

· and

An example of a recursive function that counts from 1 to 10 similarly to Listing 8.5 is given in Listing 13.2.

Listing 13.2 countRecursive.fsx: Counting to 10 using recursion.

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

prt 1 10
```

```
$ fsharpc --nologo countRecursive.fsx && mono countRecursive.exe
1 2 3 4 5 6 7 8 9 10
```

Here the `prt` function calls itself repeatedly, such that the first call is `prt 1 10`, which calls `prt 2 10`, and so on until the last call `prt 11 10`. Each time `prt` is called, new bindings named `a` and `b` are made to new values. This is illustrated in Figure 13.2. The old values are no longer accessible as indicated by subscript in the figure. E.g., in $\texttt{prt}_3$ the scope has access to $\texttt{a}_3$ but not $\texttt{a}_2$ and $\texttt{a}_1$. Thus, in this program, process is similar to a `for` loop, where the counter is `a` and in each loop its value is reduced.

The structure of the function is typical for recursive functions. They very often follow the following pattern.

Listing 13.3 Recursive functions consists of a stopping criterium, a stopping expression, and a recursive step.

```
let rec f a =
  if <stopping condition>
  then <stopping step>
  else <recursion step>
```

The *match – with* are also very common conditional structures. In Listing 13.2 `a > b` is the *stopping condition*, `printfn "\n"` is *stopping step*, and `printfn "%d " a; prt (a + 1) b` is the *recursion step*.

- match
- with
- stopping condition
- stopping step
- recursion step

13.2 The call stack and tail recursion

Fibonacci's sequence of numbers is a recursive sequence of numbers with relations to the Golden ratio and structures in biology. Fibonacci's sequence is the sequence of numbers $1, 1, 2, 3, 5, 8, 13, \ldots$. The sequence starts with $1, 1$ and the next number is recursively given as the sum of the two previous. A direct implementation of this is given in Listing 8.7.

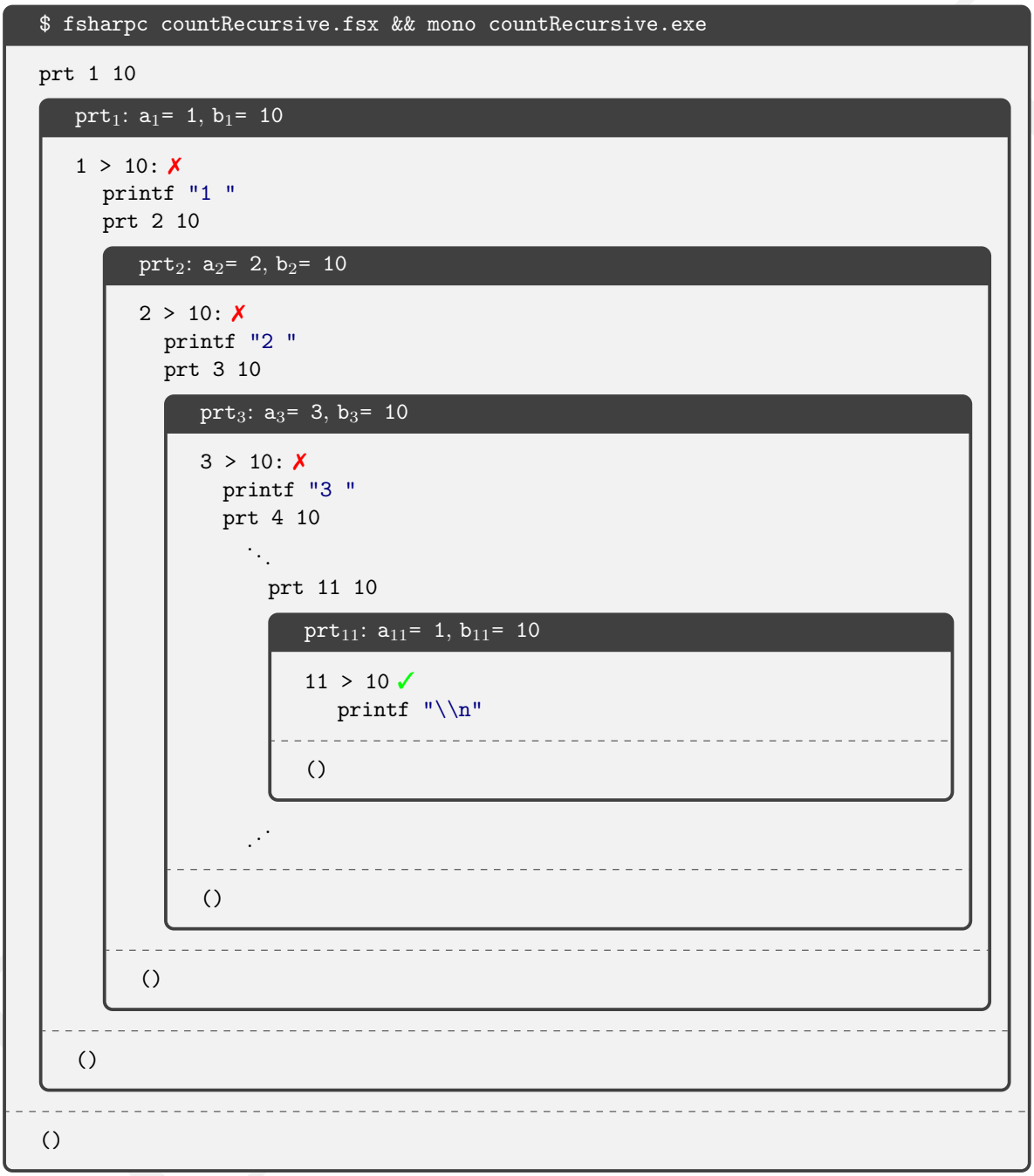


Figure 13.2: Illustration of the recursion used to write the sequence "1 2 3 ... 10" in line 8 in Listing 13.2. Each frame corresponds to a call to `prt`, where new values overshadow old. All return `unit`.

Listing 13.4 fibRecursive.fsx:
The n'th Fibonacci number using recursive.

```
let rec fib n =
  if n < 1 then
    0
  elif n = 1 then
    1
  else
    fib (n - 1) + fib (n - 2)

for i = 0 to 10 do
  printfn "fib(%d) = %d" i (fib i)
```

```
$ fsharpc --nologo fibRecursive.fsx && mono fibRecursive.exe
fib(0) = 0
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
```

Here we extended the sequence to $0, 1, 1, 2, 3, 5, \ldots$ and starting sequence $0, 1$ allowing us to define all $\text{fib}(n) = 0$, $n < 1$. Thus, our function is defined for all integers, and the irrelevant negative arguments fails gracefully by returning 0. This is a general advice: **make functions that fails gracefully.** Advice

A visualization of the calls and the scopes created by `fibRecursive` is shown in Figure 13.3. The figure illustrates that each recursive step results in two calls to the function, thus creating two new scopes. And it gets worse. Figure 13.4 illustrates the tree of calls for `fib 5`. Thus a call to the function `fib` generates a tree of calls that is five levels deep and has `fib(5)` number of nodes. In general for the program in Listing 13.4, a call to `fib(n)` produces a tree with `fib(n)` $\leq c\alpha^n$ calls to the function for some positive constant c and $\alpha \geq \frac{1+\sqrt{5}}{2} \sim 1.6$[1]. Each call takes time and requires memory, and we have thus created a slow and somewhat memory intensive function. This is a hugely ineffective implementation of calculating entries into Fibonacci's sequence, since many of the calls are identical. E.g., in Figure 13.4 `fib 1` is called five times. Before we examine a faster algorithm, we first need to discuss how F# executes function calls.

When a function is called, then memory is dynamically allocated internally for the function on what is known as the *call stack*. Stacks are used for many things in programming, but typically the call stack is considered special, since it is almost always implicitly part of any program execution. Hence, it is often just referred to as *The Stack*. When a function is called, a new *stack frame* is stacked (pushed) on the call stack including its arguments, local storage such as mutable values, and where execution should return to, when the function is finished. When the function finishes, the stack frame is unstacked (popped) and in its st the return value of the function is stacked. This return value is then unstacked and used by the caller. After unstacking the return value, the call stack is identical to its state prior to the call. Figure 13.5 shows snapshots of the call stack, when calling `fib 5` in Listing 13.4. The call first stacks a frame onto the call stack with everything needed to execute the function body plus a reference to where tl turn to, when the execution is finished. Then the body

· call stack
· The Stack
· stack frame

[1] Jon: `https://math.stackexchange.com/questions/674533/prove-upper-bound-big-o-for-fibonaccis-sequence`

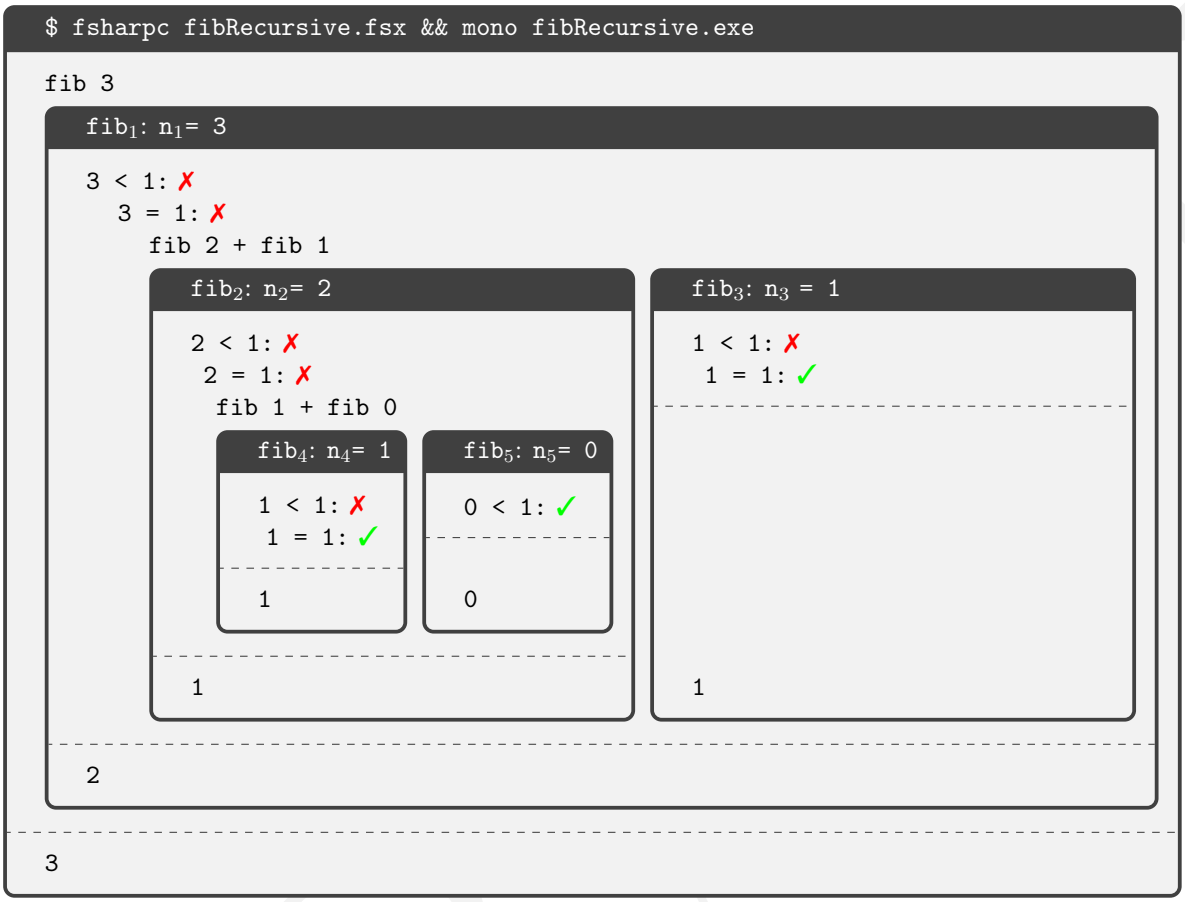
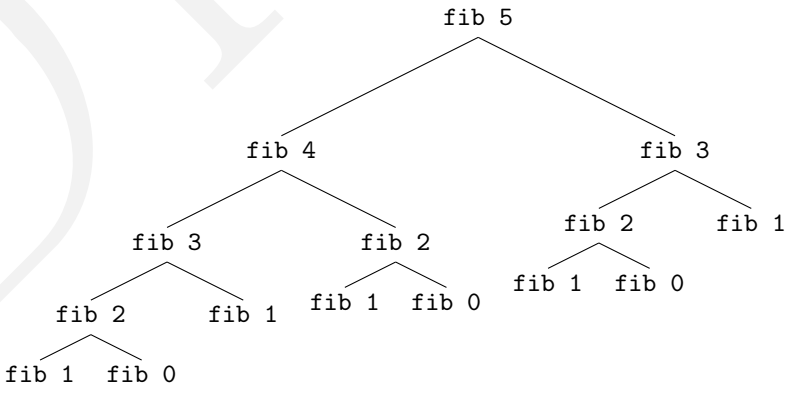


Figure 13.3: Illustration of the recursion used to write the sequence "1 2 3 ... 10" in line 8 in Listing 13.2. Each frame corresponds to a call to `fib`, where new values overshadow old.



Ialt 15 kald til fib ()

Figure 13.4: The function calls involved in calling `fib 5`.

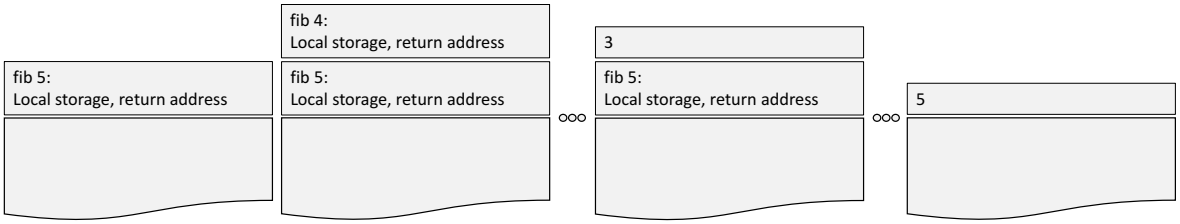


Figure 13.5: A call to `fib 5` in Listing 13.4 starts a sequence of function calls and stack frames on the call stack.

of `fib` is executed, which includes calling `fib 4` and `fib 3` in turn. The call to `fib 4` stacks a frame onto the call stack, and its body is executed. Once execution is returned from the call to `fib 4`, the result of the function is on top of the stack. It is unstacked, saved and the call to `fib 3` is treated equally. When the end of `fib 5` is reached, its frame is unstacked, and its result is stacked. In this way, the call stack is returned to its original state except for the result of the function, and execution is returned to the point right after the original call to `fib 5`. Thus, for Listing 13.4 $\mathcal{O}\left(\alpha^n\right)$, $\alpha = \frac{1+\sqrt{5}}{2}$ stacking operations are performed for a call to `fib n`. The $\mathcal{O}\left(f(n)\right)$ is the *Landau symbol* used to denote the order of a function, such that if $g(n) = \mathcal{O}(f(n))$ then there exists two real numbers $M > 0$ and a n_0 such that for all $n \geq n_0$, $|g(n)| \leq M|f(n)|$.[2] As indicated by the tree in Figure 13.4, the call tree is maximally n high, which corresponds to a maximum of n additional stack frames as compared to the starting point.

· Landau symbol

The implementation of Fibonacci's sequence in Listing 13.4 can be improved to run faster and use less memory. One such algorithm is given in Listing 13.5

Listing 13.5 fibRecursiveAlt.fsx:
A fast, recursive implementation of Fibonacci's numbers. Compare with Listing 13.4.

```
let fib n =
  let rec fibPair n pair =
    if n < 2 then pair
    else fibPair (n - 1) (snd pair, fst pair + snd pair)
  if n < 1 then 0
  elif n = 1 then 1
  else fibPair n (0, 1) |> snd

printfn "fib(10) = %d" (fib 10)
```

```
$ fsharpc --nologo fibRecursiveAlt.fsx && mono fibRecursiveAlt.exe
fib(10) = 55
```

Calculating the 45th Fibonacci number on a Macbook Pro, with a 2.9 Ghz Intel Core i5 using Listing 13.4 takes about 11.2s, while using Listing 13.5 is about 224 times faster and only takes 0.050s. The reason is that `fib` in Listing 13.5 calculates every number in the sequence once and only once by processing the list recursively, while maintaining the previous two values needed to calculate the next in the sequence. I.e., the function `helper` transforms the pair `(a,b)` to `(b,a+b)` such that, e.g., the 4th and 5th pair `(3,5)` is transformed into the 5th and the 6th pair `(5,8)` in the sequence. What complicates the algorithm is that besides the transformation, we must keep track of when to stop, which here is done using a counter variable, that is recursively reduced by 1 until our stopping criterium.

Listing 13.5 also uses much less memory than Listing 13.4, since its recursive call is the last expression

[2] Jon: **Introduction of Landau notation needs to be moved earlier, since it used in Collections chapter.**

in the function, and since the return value of two recursive calls to `helper` is the same as the return value of the last. In fact, the return value of any number of recursive calls to `helper` is the return value of the last. This structure is called *tail-recursion*. Compilers can easily optimize the call stack usage for tail recursion, since when in this example `helper` calls itself, then its frame is no longer needed, and may be replaced by the new `helper` with the slight modification, that the return point should be to `fib` and not the end of the previous `helper`. Once the recursion reaches the stopping criteria, then instead of popping a long list of calls of `helper` frames, then there is only one, and the return value is equal to the return value of the last call and the return point is to `fib`. Thus, many stack frames in tail recursion is replaced by one. Hence, **prefer tail-recursion whenever possible.**

· tail-recursion

Advice

13.3 Mutual recursive functions

Functions that recursively call each other are called *mutually recursive* functions. F# offers the *let – rec – and* notation for co-defining mutually recursive functions. As an example, consider the function `even : int -> bool`, which returns true if its argument is even and false otherwise, and the opposite function `odd : int -> bool`. A mutually recursive implementation of these functions can be developed from the following relations: `even 0 = true`, `odd 0 = false`, and for $n > 0$, `even n = odd (n-1)`, which implies that for $n > 0$, `odd n = even (n-1)`:

· mutually recursive
· `let`
· `rec`
· `and`

Listing 13.6 mutuallyRecursive.fsx:
Using mutual recursion to implement even and odd functions.

```
let rec even x =
  if x = 0 then true
  else odd (x - 1)
and odd x =
  if x = 0 then false
  else even (x - 1);;

let w = 5;
printfn "%*s %*s %*s" w "i" w "even" w "odd"
for i = 1 to w do
  printfn "%*d %*b %*b" w i w (even i) w (odd i)
```

```
$ fsharpc --nologo mutuallyRecursive.fsx && mono mutuallyRecursive.exe
    i  even   odd
    1 false  true
    2  true false
    3 false  true
    4  true false
    5 false  true
```

Notice that in the lightweight notation the `and` must be on the same indentation level as the original `let`.

Without the `and` keyword, F# will issue a compile error at the definition of `even`. However, it is possible to implement mutual recursion by using functions as an argument, e.g.,

Listing 13.7 mutuallyRecursiveAlt.fsx:
Mutual recursion without the and keyword needs a helper function.

```
let rec evenHelper (notEven: int -> bool) x =
  if x = 0 then true
  else notEven (x - 1)

let rec odd x =
  if x = 0 then false
  else evenHelper odd (x - 1);;

let even x = evenHelper odd x

let w = 5;
printfn "%*s %*s %*s" w "i" w "Even" w "Odd"
for i = 1 to w do
  printfn "%*d %*b %*b" w i w (even i) w (odd i)
```

```
$ fsharpc --nologo mutuallyRecursiveAlt.fsx
$ mono mutuallyRecursiveAlt.exe
    i  Even   Odd
    1 false  true
    2  true false
    3 false  true
    4  true false
    5 false  true
```

But, Listing 13.6 is clearly to be preferred over Listing 13.7.

In the above we used the `even` and `odd` function problems to demonstrate mutual recursion. There is, of course, a much simpler solution, which does not use recursion at all:

Listing 13.8 parity.fsx:
A better way to test for parity without recursion.

```
let even x = (x % 2 = 0)
let odd x = not (even x)
```

which is to be preferred anytime as the solution to the problem. [3]

[3] Jon: **Here it would be nice to have an intermezzo, giving examples of how to write a recursive program by thinking the problem has been solved.**

Bibliography

[1] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345—-363, 1936.

[2] Ole-Johan Dahl and Kristen Nygaard. SIMULA a language for programming and description of discrete event systems. introduction and user's manual. Technical report, Norwegian Computing Center, 1967.

[3] European Computer Manufacturers Association (ECMA). Standard ecma-335, common language infrastructure (cli). `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[4] International Organization for Standardization. Iso/iec 23271:2012, common language infrastructure (cli). `https://www.iso.org/standard/58046.html`.

[5] Object Management Group. Uml version 2.0. `http://www.omg.org/spec/UML/2.0/`.

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

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

[8] 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/`.

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

[10] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1936.

Index