9 Organising code in libraries and application programs

In this chapter, we will focus on a number of ways to make the code available as *library* functions in F#. A library is 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 organize 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
```

Here 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 type system 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 file order 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.

```
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.

```
open Meta
let add : floatFunction = fun x y -> x + y
let result = apply add 3.0 4.0
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.

```
$ fsharpc --nologo MetaExplicit.fsx MetaAppWOpen.fsx && mono
   MetaAppWOpen.exe
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 pollution both 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.

```
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 to use a *namespace*, which only can hold modules and · namespace

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

type 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 organizing 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, the 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.

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

To compile we now need to include all three files in the right order. Likewise, the 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.

```
1 $ fsharpc --nologo namespace.fsx namespaceExtension.fsx namespaceApp.fsx
     && mono namespaceApp.exe
2 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 from 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.

```
1 $ 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 the 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 offer 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 at this point it is 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.
```

10 | Testing programs

A software bug is an error in a computer program that causes it to produce an incorrect result or behave in an unintended manner. The term bug was used by Thomas Edison in 1878[1], but made popular in computer science by Grace Hopper, who found a moth interfering with the electronic circuits of the Harward Mark II electromechanical computer and coined the term *bug* for errors in computer programs. The original bug is shown in Figure 10.1. Software is everywhere, and errors therein have a huge economic impact on our society and can threaten lives[2].

· bug

The ISO/IEC organizations have developed standards for software testing[3]. To illustrate basic concepts of software quality, consider a hypothetical route planning system. Essential factors of its quality are,

· functionality

Functionality: Does the software compile and run without internal errors. Does it solve the problem, it was intended to solve? E.g., does the route planning software find a suitable route from point a to b?

· reliability

Reliability: Does the software work reliably over time? E.g., does the route planning software work in case of internet dropouts?

· usability

Usability: Is the software easy and intuitive to use by humans? E.g., is it easy to enter addresses and alternative routes in the software's interface?

· efficiency

[1] https://en.wikipedia.org/wiki/Software_bug, possibly http://edison.rutgers.edu/NamesSearch/DocImage.php3?DocId=LB003487

[2] https://en.wikipedia.org/wiki/List_of_software_bugs

[3] ISO/IEC 9126, International standard for the evaluation of software quality, December 19, 1991, later replaced by ISO/IEC 25010:2011

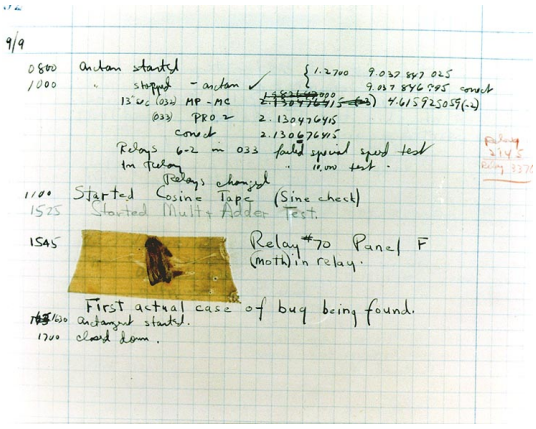

Figure 10.1: The first computer bug caught by Grace Hopper, U.S. Naval Historical Center Online Library Photograph NH 96566-KN.

Efficiency: How many computer and human resources does the software require? E.g., does it take milliseconds or hours to find a requested route? Can the software run on a mobile platform with limited computer speed and memory?

· maintainability

Maintainability: In case of the discovery of new bugs, is it easy to test and correct the software? Is it easy to extend the software with new functionality? E.g., is it easy to update the map with updated roadmaps and new information? Can the system be improved to work both for car drivers and bicyclists?

· portability

Portability: Is it easy to port the software to new systems such as new server architecture and screen sizes? E.g., if the routing software originally was written for IOS devices, will it be easy to port to Android systems?

The above-mentioned concepts are ordered based on the requirements of the system. Functionality and reliability are perhaps the most important concepts, since if the software does not solve the specified problem, then the software design process has failed. However, many times the problem definition will evolve along with the software development process. But as a bare minimum, the software should run without internal errors and not crash under a well-defined set of circumstances. Furthermore, it is often the case that software designed for the general public requires a lot of attention to the usability of the software since in many cases non-experts are expected to be able to use the software with little or no prior training. On the other hand, software used internally in companies will be used by a small number of people, who become experts in using the software, and it is often less important that the software is easy to understand by non-experts. An example is text processing software like Microsoft Word versus Gnu Emacs and LaTeX. Word is designed to be used by non-experts for small documents such as letters and notes and relies heavily on interfacing with the system using click-interaction. On the other hand, Emacs and LaTeX are for experts for longer and professionally typeset documents and relies heavily on keyboard shortcuts and text-codes for typesetting document entities.

· software testing
· debugging

The purpose of *software testing* is to find bugs. When errors are found, then we engage in *debugging*, which is the process of diagnosing and correcting bugs. Once we have a failed software test, i.e., one that does not find any bugs, then we have strengthened our belief in the software, but it is important to note that software testing and debugging rarely removes all bugs, and with each correction or change of software, there is a fair chance of introducing new bugs. It is not exceptional that the software testing the software is as large as the software itself.

· white-box testing
· black-box testing
· unit testing

In this chapter, we will focus on two approaches to software testing, which emphasizes functionality: *white-box* and *black-box testing*. An important concept in this context is *unit testing*, where the program is considered in smaller pieces, called units, and for which accompanying programs for testing can be made, which tests these units automatically. Black-box testing considers the problem formulation and the program interface, and can typically be written early in the software design phase. In contrast, white-box testing considers the program text, and thus requires the program to be available. Thus, there is a tendency for black-box test programs to be more stable, while white-box testing typically is developed incrementally alongside the software development.

To illustrate software testing we'll start with a problem:

Problem 10.1

Given any date in the Gregorian calendar, calculate the day of the week.

Facts about dates in the Gregorian calendar are:

- combinations of dates and weekdays repeat themselves every 400 years;
- the typical length of the months January, February, ... follow the knuckle rule, i.e., January belongs to the index knuckle, February to the space between the index and the middle finger,

and August restarts or starts on the other hand. All knuckle months have 31 days, all spacing months have 30 days except February, which has 29 days on leap years and 28 days all other years.

- A leap year is a multiple of 4, except if it is also a multiple of 100 but not of 400.

Many solutions to the problem have been discovered, and here we will base our program on Gauss' method, which is based on integer division and calculates the weekday of the 1st of January of a given year. For any other date, we will count our way through the weeks from the previous 1st of January. The algorithm relies on an enumeration of weekdays starting with Sunday = 0, Monday = 1, ..., and Saturday = 6. Our proposed solution is shown in Listing 10.1.[4]

Listing 10.1 date2Day.fsx:
A function that can calculate day-of-week from any date in the Gregorian calendar.

```
let januaryFirstDay (y : int) =
  let a = (y - 1) % 4
  let b = (y - 1) % 100
  let c = (y - 1) % 400
  (1 + 5 * a + 4 * b + 6 * c) % 7

let rec sum (lst : int list) j =
  if 0 <= j && j < lst.Length then
    lst.[0] + sum lst.[1..] (j - 1)
  else
    0

let date2Day d m y =
  let dayPrefix =
    ["Sun"; "Mon"; "Tues"; "Wednes"; "Thurs"; "Fri"; "Satur"]
  let feb = if (y % 4 = 0) && ((y % 100 <> 0) || (y % 400 = 0)) then 29
   else 28
  let daysInMonth = [31; feb; 31; 30; 31; 30; 31; 31; 30; 31; 30; 31]
  let dayOne = januaryFirstDay y
  let daysSince = (sum daysInMonth (m - 2)) + d - 1
  let weekday = (dayOne + daysSince) % 7;
  dayPrefix.[weekday] + "day"
```

10.1 White-box testing

White-box testing considers the text of a program. The degree to which the text of the program is covered in the test is called *coverage*. Since our program is small, we do have the opportunity to ensure that all functions are called at least once, which is called *function coverage*, we will also be able to test every branching in the program, which is called *branching coverage*, an in this case that implies *statement coverage*. The procedure is as follows:

- white-box testing
- coverage
- function coverage
- branching coverage
- statement coverage

1. Decide which are the units to test: The program, shown in Listing 10.1, has 3 functions, and we will consider these each as a unit, but we might as well just have chosen `date2Day` as a single unit. The important part is that the union of units must cover the whole program text, and since `date2Day` calls both `januaryFirstDay` and `sum`, designing test cases for the two later is superfluous. However, we may have to do this anyway, when debugging, and we may choose at

[4] Jon: **This example relies on lists, which has not been introduced yet.**

a later point to use these functions separately, and in both cases, we will be able to reuse the testing of the smaller units.

2. Identify branching points: The function `januaryFirstDay` has no branching function, `sum` has one, and depending on the input values two paths through the code may be used, and `date2Day` has one, where the number of days in February is decided. Note that in order to test this, our test-date must be March 1 or later. In this example, there are only examples of `if`-branch points, but they may as well be loops and pattern matching expressions. In the following code, the branch points have been given a comment and a number, as shown in Listing 10.2.

Listing 10.2 date2DayAnnotated.fsx:
In white-box testing, the branch points are identified.

```
// Unit: januaryFirstDay
let januaryFirstDay (y : int) =
  let a = (y - 1) % 4
  let b = (y - 1) % 100
  let c = (y - 1) % 400
  (1 + 5 * a + 4 * b + 6 * c) % 7

// Unit: sum
let rec sum (lst : int list) j =
  (* WB: 1 *)
  if 0 <= j && j < lst.Length then
    lst.[0] + sum lst.[1..] (j - 1)
  else
    0

// Unit: date2Day
let date2Day d m y =
  let dayPrefix =
    ["Sun"; "Mon"; "Tues"; "Wednes"; "Thurs"; "Fri"; "Satur"]
  (* WB: 1 *)
  let feb = if (y % 4 = 0) && ((y % 100 <> 0) || (y % 400 = 0)) then
   29 else 28
  let daysInMonth = [31; feb; 31; 30; 31; 30; 31; 31; 30; 31; 30; 31]
  let dayOne = januaryFirstDay y
  let daysSince = (sum daysInMonth (m - 2)) + d - 1
  let weekday = (dayOne + daysSince) % 7;
  dayPrefix.[weekday] + "day"
```

3. For each unit, produce an input set that tests each branch: In our example, the branch points depend on a Boolean expression, and for good measure, we are going to test each term that can lead to branching. Thus,

Unit	Branch	Condition	Input	Expected output
`januaryFirstDay`	0	-	2016	5
`sum`	1	`0 <= j && j < lst.Length`		
	1a	`true && true`	`[1; 2; 3] 1`	3
	1b	`false && true`	`[1; 2; 3] -1`	0
	1c	`true && false`	`[1; 2; 3] 10`	0
	1d	`false && false`	-	-

`date2Day`	1	`(y % 4 = 0)`
		`&& ((y % 100 <> 0)`
		`|| (y % 400 = 0))`		

	-	`true && (true || true)`	-	-
	1a	`true && (true || false)`	8 9 2016	Thursday
	1b	`true && (false || true)`	8 9 2000	Friday
	1c	`true && (false || false)`	8 9 2100	Wednesday
	-	`false && (true || true)`	-	-
	1d	`false && (true || false)`	8 9 2015	Tuesday
	-	`false && (false || true)`	-	-
	-	`false && (false || false)`	-	-

The impossible cases have been intentionally blank, e.g., it is not possible for $j < 0$ and $j > n$ for some positive value n.

4. Write a program that tests all these cases and checks the output, e.g.,

Listing 10.3 date2DayWhiteTest.fsx:
The tests identified by white-box analysis. The program from Listing 10.2 has been omitted for brevity.

```
printfn "White-box testing of date2Day.fsx"
printfn "  Unit: januaryFirstDay"
printfn "    Branch: 0 - %b" (januaryFirstDay 2016 = 5)

printfn "  Unit: sum"
printfn "    Branch: 1a - %b" (sum [1; 2; 3] 1 = 3)
printfn "    Branch: 1b - %b" (sum [1; 2; 3] -1 = 0)
printfn "    Branch: 1c - %b" (sum [1; 2; 3] 10 = 0)

printfn "  Unit: date2Day"
printfn "    Branch: 1a - %b" (date2Day 8 9 2016 = "Thursday")
printfn "    Branch: 1b - %b" (date2Day 8 9 2000 = "Friday")
printfn "    Branch: 1c - %b" (date2Day 8 9 2100 = "Wednesday")
printfn "    Branch: 1d - %b" (date2Day 8 9 2015 = "Tuesday")
```

```
$ fsharpc --nologo date2DayWhiteTest.fsx && mono
   date2DayWhiteTest.exe
White-box testing of date2Day.fsx
  Unit: januaryFirstDay
    Branch: 0 - true
  Unit: sum
    Branch: 1a - true
    Branch: 1b - true
    Branch: 1c - true
  Unit: date2Day
    Branch: 1a - true
    Branch: 1b - true
    Branch: 1c - true
    Branch: 1d - true
```

Notice that the output of the tests is organized such that they are enumerated per unit, hence we can rearrange as we like and still uniquely refer to a unit's test. Also, the output of the test program produces a list of tests that should return true or success or a similar positively loaded word, but without further or only little detail, such that we at a glance can identify any test that produced unexpected results.

After the white-box testing has failed to find errors in the program, we have some confidence in the program, since we have run every line at least once. It is, however, in no way a guarantee that the program is error free, which is why white-box testing is often accompanied with black-box testing to be described next.

10.2 Black-box testing

In black-box testing, the program is considered a black box, and no knowledge is required about how a particular problem is solved, in fact, it is often useful not to have that knowledge at all. It is rarely possible to test all input to a program, so in black-box testing, the solution is tested for typical and extreme cases based on knowledge of the problem. The procedure is as follows:

Decide on the interface to use: It is useful to have an agreement with the software developers about what interface is to be used, e.g., in our case, the software developer has made a function `date2Day d m y` where `d`, `m`, and `y` are integers specifying the day, month, and year.

Make an overall description of the tests to be performed and their purpose:

1 a consecutive week, to ensure that all weekdays are properly returned

2 two set of consecutive days across boundaries that may cause problems: across a new year, across a regular month boundary.

3 a set of consecutive days across February-March boundaries for a leap and non-leap year

4 four dates after February in a non-multiple-of-100 leap year and in a non-leap year, a multiple-of-100-but-not-of-400 non-leap year, and a multiple-of-400 leap year.

Given no information about the program's text, there are other dates that one could consider as likely candidates for errors, but the above is judged to be a fair coverage.

Choose a specific set of input and expected output relations on the tabular form:

Test number	Input	Expected output
1a	1 1 2016	Friday
1b	2 1 2016	Saturday
1c	3 1 2016	Sunday
1d	4 1 2016	Monday
1e	5 1 2016	Tuesday
1f	6 1 2016	Wednesday
1g	7 1 2016	Thursday
2a	31 12 2014	Wednesday
2b	1 1 2015	Thursday
2c	30 9 2017	Saturday
2d	1 10 2017	Sunday
3a	28 2 2016	Sunday
3b	29 2 2016	Monday
3c	1 3 2016	Tuesday
3d	28 2 2017	Tuesday
3e	1 3 2017	Wednesday
4a	1 3 2015	Sunday
4b	1 3 2012	Thursday
4c	1 3 2000	Wednesday
4d	1 3 2100	Monday

Write a program executing the tests as shown in Listing 10.4 and 10.5.

Listing 10.4 date2DayBlackTest.fsx:
The tests identified by black-box analysis. The program from Listing 10.2 has been omitted for brevity.

```
28 let testCases = [
29   ("A complete week",
30    [(1, 1, 2016, "Friday");
31     (2, 1, 2016, "Saturday");
32     (3, 1, 2016, "Sunday");
33     (4, 1, 2016, "Monday");
34     (5, 1, 2016, "Tuesday");
35     (6, 1, 2016, "Wednesday");
36     (7, 1, 2016, "Thursday");]);
37   ("Across boundaries",
38    [(31, 12, 2014, "Wednesday");
39     (1, 1, 2015, "Thursday");
40     (30, 9, 2017, "Saturday");
41     (1, 10, 2017, "Sunday")]);
42   ("Across feburary boundary",
43    [(28, 2, 2016, "Sunday");
44     (29, 2, 2016, "Monday");
45     (1, 3, 2016, "Tuesday");
46     (28, 2, 2017, "Tuesday");
47     (1, 3, 2017, "Wednesday")]);
48   ("Leap years",
49    [(1, 3, 2015, "Sunday");
50     (1, 3, 2012, "Thursday");
51     (1, 3, 2000, "Wednesday");
52     (1, 3, 2100, "Monday")]);
53   ]
54
55 printfn "Black-box testing of date2Day.fsx"
56 for i = 0 to testCases.Length - 1 do
57   let (setName, testSet) = testCases.[i]
58   printfn "  %d. %s" (i+1) setName
59   for j = 0 to testSet.Length - 1 do
60     let (d, m, y, expected) = testSet.[j]
61     let day = date2Day d m y
62     printfn "    test %d - %b" (j+1) (day = expected)
```

Listing 10.5: Output from Listing 10.4.

```
$ fsharpc --nologo date2DayBlackTest.fsx && mono
   date2DayBlackTest.exe
Black-box testing of date2Day.fsx
  1. A complete week
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
    test 5 - true
    test 6 - true
    test 7 - true
  2. Across boundaries
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
  3. Across feburary boundary
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
    test 5 - true
  4. Leap years
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
```

Notice how the program has been made such that it is almost a direct copy of the table, produced in the previous step.

A black-box test is a statement of what a solution should fulfill for a given problem. Hence, **it is a good idea to make a black-box test early in the software design phase, in order to clarify the requirements for the code to be developed, and take an outside view of the code prior to developing it.** Advice

After the black-box testing has failed to find errors in the program, we have some confidence in the program, since, from a user's perspective, the program produces sensible output in many cases. It is, however, in no way a guarantee that the program is error free.

10.3 Debugging by tracing

Once an error has been found by testing, then the *debugging* phase starts. The cause of a bug can either be that the chosen algorithm is the wrong one for the job, or the implementation of it has an error. In the debugging process, we have to keep an open mind, and not rely on assumptions, since assumptions tend to blind the reader of a text. A frequent source of errors is that the state of a program is different, than expected, e.g., because the calculation performed is different than intended, or that the return of a library function is different than expected. The most important tool for debugging is *simplification*. This is similar to white-box testing, but where the units tested are very small. E.g., the suspected piece of code could be broken down into smaller functions or code snippets, which is given well-defined input, and, e.g., use `printfn` statements to obtain the output of the code snippet. Another related technique is to use *mockup code*, which replaces parts of the code with code that produces safe and

· debugging

· mockup code

relevant results. If the bug is not obvious then more rigorous techniques must be used such as *tracing*. · tracing
Some development interfaces have built-in tracing system, e.g., `fsharpi` will print inferred types and some binding values. However, often a source of a bug is due to a misunderstanding of the flow of data through a program execution, and we will in the following introduce *hand tracing* a technique to · hand tracing
simulate the execution of a program by hand.

Consider the program in Listing 10.6. [5]

Listing 10.6 gcd.fsx:
gcd

```
1  let rec gcd a b =
2    if a < b then
3      gcd b a
4    elif b > 0 then
5      gcd b (a % b)
6    else
7      a
8
9  let a = 10
10 let b = 15
11 printfn "gcd %d %d = %d" a b (gcd a b)
```

```
1  $ fsharpc --nologo gcd.fsx && mono gcd.exe
2  gcd 10 15 = 5
```

The greatest common divisor of 2 integers. which includes a function for calculating the greatest common divisor of 2 integers, and calls this function with the numbers 10 and 15. Hand tracing this program means that we simulate its execution and as part of that keep track of the bindings, assignments and input and output of the program. To do this, we need to consider code snippet's *environment*. E.g., to hand trace the above program, we start by noting the outer environment, called · environment
E_0 for short. In line 1, then the `gcd` identifier is bound to a function, hence we write:

$$E_0 : \\ \text{gcd} \rightarrow \big((a, b), \text{gcd-body}, \varnothing\big)$$

Function bindings like this one is noted as a closure, which is the triplet $\big(\text{arguments}, \text{expression}, \text{environment}\big)$. The closure is everything needed for the expression to be calculated. Here we wrote gcd-body to denote everything after the equal sign in the function binding. Next, F# executes line 9 and 10, and we update our environment to reflect the bindings as,

$$E_0 : \\ \text{gcd} \rightarrow \big((a, b), \text{gcd-body}, \varnothing\big) \\ a \rightarrow 10 \\ b \rightarrow 15$$

[5] Jon: **This program uses recursion, which has not been introduced yet.**

In line 11 the function is evaluated. This initiates a new environment E_1, and we update our trace as,

$$
\begin{aligned}
&E_0: \\
&\quad \text{gcd} \to \big((a, b), \text{gcd-body}, \varnothing\big) \\
&\quad a \to 10 \\
&\quad b \to 15 \\
&\quad \text{line } 11\text{: gcd a b} \to ? \\
&E_1: \big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big)
\end{aligned}
$$

where the new environment is noted to have gotten its argument names `a` and `b` bound to the values `10` and `15` respectively, and where the return of the function to environment E_0 is yet unknown, so it is noted as a question mark. In line 2 the comparison `a < b` is checked, and since we are in environment E_1 then this is the same as checking `10 < 15`, which is true so the program executes line 3. Hence, we initiate a new environment E_2 and update our trace as,

$$
\begin{aligned}
&E_0: \\
&\quad \text{gcd} \to \big((a, b), \text{gcd-body}, \varnothing\big) \\
&\quad a \to 10 \\
&\quad b \to 15 \\
&\quad \text{line } 11\text{: gcd a b} \to ? \\
&E_1: \big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line } 3\text{: gcd b a} \to ? \\
&E_2: \big((a \to 15, b \to 10), \text{gcd-body}, \varnothing\big)
\end{aligned}
$$

where in the new environment `a` and `b` bound to the values `15` and `10` respectively. In E_2, `10 < 15` is false, so the program evaluates `b > 0`, which is true, hence line 5 is executed. This calls `gcd` once again, but with new arguments, and `a % b` is parenthesized, then it is evaluated before `gcd` is called.

Hence, we update our trace as,

$$
\begin{aligned}
&E_0: \\
&\quad \text{gcd} \to \big((a, b), \text{gcd-body}, \varnothing\big) \\
&\quad a \to 10 \\
&\quad b \to 15 \\
&\quad \text{line } 11\text{: gcd a b} \to ? \\
&E_1: \big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line } 3\text{: gcd b a} \to ? \\
&E_2: \big((a \to 15, b \to 10), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line } 5\text{: a \% b} \to 5 \\
&\quad \text{line } 5\text{: gcd b (a \% b)} \to ? \\
&E_3: \big((a \to 10, b \to 5), \text{gcd-body}, \varnothing\big)
\end{aligned}
$$

Again we fall through to line 5, evaluate the remainder operator and initiates a new environment,

E_0 :
gcd $\rightarrow \big((a, b), \text{gcd-body}, \varnothing\big)$
$a \rightarrow 10$
$b \rightarrow 15$
line 11: gcd a b $\rightarrow$?
E_1 : $\big((a \rightarrow 10, b \rightarrow 15), \text{gcd-body}, \varnothing\big)$
line 3: gcd b a $\rightarrow$?
E_2 : $\big((a \rightarrow 15, b \rightarrow 10), \text{gcd-body}, \varnothing\big)$
line 5: a % b $\rightarrow$ 5
line 5: gcd b (a % b) $\rightarrow$?
E_3 : $\big((a \rightarrow 10, b \rightarrow 5), \text{gcd-body}, \varnothing\big)$
line 5: a % b $\rightarrow$ 0
line 5: gcd b (a % b) $\rightarrow$?
E_4 : $\big((a \rightarrow 5, b \rightarrow 0), \text{gcd-body}, \varnothing\big)$

This time both `a < b` and `b > 0` are false, so we fall through to line 7, and `gcd` from E_4 returns its value of `a`, which is `5`, so we scratch E_4 and change the question mark in E_3 to `5:`

E_0 :
gcd $\rightarrow \big((a, b), \text{gcd-body}, \varnothing\big)$
$a \rightarrow 10$
$b \rightarrow 15$
line 11: gcd a b $\rightarrow$?
E_1 : $\big((a \rightarrow 10, b \rightarrow 15), \text{gcd-body}, \varnothing\big)$
line 3: gcd b a $\rightarrow$?
E_2 : $\big((a \rightarrow 15, b \rightarrow 10), \text{gcd-body}, \varnothing\big)$
line 5: a % b $\rightarrow$ 5
line 5: gcd b (a % b) $\rightarrow$?
E_3 : $\big((a \rightarrow 10, b \rightarrow 5), \text{gcd-body}, \varnothing\big)$
line 5: a % b $\rightarrow$ 0
line 5: gcd b (a % b) $\rightarrow$ ~~?~~ 5
~~E_4~~ : $\big((a \rightarrow 5, b \rightarrow 0), \text{gcd-body}, \varnothing\big)$

Now line 5 in E_3 is also a return point of `gcd`, hence we scratch E_3 and change the question mark in

E_2 to 5,

E_0 :
gcd $\to$ $\big((a, b), \text{gcd-body}, \varnothing\big)$
$a \to 10$
$b \to 15$
line 11: gcd a b $\to$?
E_1 : $\big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big)$
line 3: gcd b a $\to$?
E_2 : $\big((a \to 15, b \to 10), \text{gcd-body}, \varnothing\big)$
line 5: a % b $\to$ 5
line 5: gcd b (a % b) $\to$ 5
~~E_3~~ : $\big((a \to 10, b \to 5), \text{gcd-body}, \varnothing\big)$
line 5: a % b $\to$ 0
line 5: gcd b (a % b) $\to$ 5
~~E_4~~ : $\big((a \to 5, b \to 0), \text{gcd-body}, \varnothing\big)$

and likewise, for E_2 and E_1:

E_0 :
gcd $\to$ $\big((a, b), \text{gcd-body}, \varnothing\big)$
$a \to 10$
$b \to 15$
line 11: gcd a b $\to$ 5
~~E_1~~ : $\big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big)$
line 3: gcd b a $\to$ 5
~~E_2~~ : $\big((a \to 15, b \to 10), \text{gcd-body}, \varnothing\big)$
line 5: a % b $\to$ 5
line 5: gcd b (a % b) $\to$ 5
~~E_3~~ : $\big((a \to 10, b \to 5), \text{gcd-body}, \varnothing\big)$
line 5: a % b $\to$ 0
line 5: gcd b (a % b) $\to$ 5
~~E_4~~ : $\big((a \to 5, b \to 0), \text{gcd-body}, \varnothing\big)$

Now we are able to continue the program in environment E_0 with the `printfn` statement, and we

write:

E_0 :
gcd $\rightarrow \big((a, b), \text{gcd-body}, \varnothing\big)$
$a \rightarrow 10$
$b \rightarrow 15$
line 11: gcd a b $\rightarrow$ ↯ 5
line 11: stdout $\rightarrow$ "gcd 10 15 = 5"
~~E_1~~ :$\big((a \rightarrow 10, b \rightarrow 15), \text{gcd-body}, \varnothing\big)$
line 3: gcd b a $\rightarrow$ ↯ 5
~~E_2~~ :$\big((a \rightarrow 15, b \rightarrow 10), \text{gcd-body}, \varnothing\big)$
line 5: a % b $\rightarrow$ 5
line 5: gcd b (a % b) $\rightarrow$ ↯ 5
~~E_3~~ :$\big((a \rightarrow 10, b \rightarrow 5), \text{gcd-body}, \varnothing\big)$
line 5: a % b $\rightarrow$ 0
line 5: gcd b (a % b) $\rightarrow$ ↯ 5
~~E_4~~ :$\big((a \rightarrow 5, b \rightarrow 0), \text{gcd-body}, \varnothing\big)$

which completes the hand tracing of `gcd.fsx`.

F# uses the lexical scope, which implies that besides function arguments, we also at times need to consider the environment at the place of writing. Consider the program in Listing 10.7.

Listing 10.7 lexicalScopeTracing.fsx: lexicalScopeTracing

```
1  let testScope x =
2    let a = 3.0
3    let f z = a * z
4    let a = 4.0
5    f x
6  printfn "%A" (testScope 2.0)
```

```
1  $ fsharpc --nologo lexicalScopeTracing.fsx
2  $ mono lexicalScopeTracing.exe
3  6.0
```

Example of lexical scope and closure environment. To hand trace this, we start by creating the outer environment, define the closure for `testScope` and reach line 6.

E_0 :
testScope $\rightarrow \big(x, \text{testScope-body}, \varnothing\big)$
line 6: testScope 2.0 $\rightarrow$?

We create new environment for `testScope` and note the bindings,

$$
\begin{aligned}
&E_0: \\
&\quad \text{testScope} \rightarrow (x, \text{testScope-body}, \varnothing) \\
&\quad \text{line 6: testScope 2.0} \rightarrow ? \\
&E_1: (x \rightarrow 2.0, \text{testScope-body}, \varnothing) \\
&\quad a \rightarrow 3.0 \\
&\quad \text{f} \rightarrow (z, \text{a * z}, (a \rightarrow 3.0, x \rightarrow 2.0)) \\
&\quad a \rightarrow 4.0
\end{aligned}
$$

Since we are working with lexical scope, then `a` is noted twice, and its interpretation is by lexical order. Hence, the environment for the closure of `f` is everything above in E_1, so we add $a \rightarrow 3.0$ and $x \rightarrow 2.0$. In line 5 `f` is called, so we create an environment based on its closure,

$$
\begin{aligned}
&E_0: \\
&\quad \text{testScope} \rightarrow (x, \text{testScope-body}, \varnothing) \\
&\quad \text{line 6: testScope 2.0} \rightarrow ? \\
&E_1: (x \rightarrow 2.0, \text{testScope-body}, \varnothing) \\
&\quad a \rightarrow 3.0 \\
&\quad \text{f} \rightarrow (z, \text{a * z}, (a \rightarrow 3.0, x \rightarrow 2.0)) \\
&\quad a \rightarrow 4.0 \\
&\quad \text{line 5: f x} \rightarrow ? \\
&E_2: (z \rightarrow 2.0, \text{a * z}, (a \rightarrow 3.0, x \rightarrow 2.0)
\end{aligned}
$$

The expression in the environment E_2 evaluates to `6.0`, and unraveling the scopes we get,

$$
\begin{aligned}
&E_0: \\
&\quad \text{testScope} \rightarrow (x, \text{testScope-body}, \varnothing) \\
&\quad \text{line 6: testScope 2.0} \rightarrow \not{?}\ 6.0 \\
&\quad \text{line 6: stdout} \rightarrow \text{"6.0"} \\
&\not{E_1}: (x \rightarrow 2.0, \text{testScope-body}, \varnothing) \\
&\quad a \rightarrow 3.0 \\
&\quad \text{f} \rightarrow (z, \text{a * z}, (a \rightarrow 3.0, x \rightarrow 2.0)) \\
&\quad a \rightarrow 4.0 \\
&\quad \text{line 5: f x} \rightarrow \not{?}\ 6.0 \\
&\not{E_2}: (z \rightarrow 2.0, \text{a * z}, (a \rightarrow 3.0, x \rightarrow 2.0)
\end{aligned}
$$

For mutable bindings, i.e., variables, the scope is dynamic. For this, we need the concept of storage. Consider the program in Listing 10.8.

Listing 10.8 dynamicScopeTracing.fsx: dynamicScopeTracing

```
1  let testScope x =
2    let mutable a = 3.0
3    let f z = a * z
4    a <- 4.0
5    f x
6  printfn "%A" (testScope 2.0)
```

```
1  $ fsharpc --nologo dynamicScopeTracing.fsx
2  $ mono dynamicScopeTracing.exe
3  8.0
```

Example of lexical scope and closure environment. We add a storage area to our hand tracing, e.g., line 6,

$$
\begin{aligned}
&\text{Store}: \\
&\quad E_0: \\
&\qquad \text{testScope} \rightarrow (x, \text{testScope-body}, \varnothing) \\
&\qquad \text{line 6: testScope } 2.0 \rightarrow ?
\end{aligned}
$$

So when we generate environment E_1, the mutable binding is to a storage location,

$$
\begin{aligned}
&\text{Store}: \\
&\qquad \alpha_1 \rightarrow 3.0 \\
&\quad E_0: \\
&\qquad \text{testScope} \rightarrow (x, \text{testScope-body}, \varnothing) \\
&\qquad \text{line 6: testScope } 2.0 \rightarrow ? \\
&\quad E_1: (x \rightarrow 2.0, \text{testScope-body}, \varnothing) \\
&\qquad a \rightarrow \alpha_1
\end{aligned}
$$

which is assigned the value `3.0` at the definition of `a`. Now the definition of `f` is using the storage location

$$
\begin{aligned}
&\text{Store}: \\
&\qquad \alpha_1 \rightarrow 3.0 \\
&\quad E_0: \\
&\qquad \text{testScope} \rightarrow (x, \text{testScope-body}, \varnothing) \\
&\qquad \text{line 6: testScope } 2.0 \rightarrow ? \\
&\quad E_1: (x \rightarrow 2.0, \text{testScope-body}, \varnothing) \\
&\qquad a \rightarrow \alpha_1 \\
&\qquad \text{f} \rightarrow (z, \text{a * z}, (a \rightarrow \alpha_1, x \rightarrow 2.0))
\end{aligned}
$$

and in line 4 it is the value in the storage, which is updated,

$$
\begin{aligned}
&\text{Store}: \\
&\qquad \alpha_1 \to \cancel{3.0}\ 4.0 \\
&E_0: \\
&\qquad \text{testScope} \to (x, \text{testScope-body}, \varnothing) \\
&\qquad \text{line 6: testScope } 2.0 \to ? \\
&E_1: (x \to 2.0, \text{testScope-body}, \varnothing) \\
&\qquad a \to \alpha_1 \\
&\qquad \text{f} \to (z, \text{a * z}, (a \to \alpha_1, x \to 2.0))
\end{aligned}
$$

Hence,

$$
\begin{aligned}
&\text{Store}: \\
&\qquad \alpha_1 \to \cancel{3.0}\ 4.0 \\
&E_0: \\
&\qquad \text{testScope} \to (x, \text{testScope-body}, \varnothing) \\
&\qquad \text{line 6: testScope } 2.0 \to ? \\
&E_1: (x \to 2.0, \text{testScope-body}, \varnothing) \\
&\qquad a \to \alpha_1 \\
&\qquad \text{f} \to (z, \text{a * z}, (a \to \alpha_1, x \to 2.0)) \\
&\qquad \text{line 5: f x} \to ? \\
&E_2: (z \to 2.0, \text{a * z}, (a \to \alpha_1, x \to 2.0)
\end{aligned}
$$

and the return value from `f` evaluated in environment E_2 now reads the value `4.0` for `a` and returns `8.0`. Hence,

$$
\begin{aligned}
&\text{Store}: \\
&\qquad \alpha_1 \to \cancel{3.0}\ 4.0 \\
&E_0: \\
&\qquad \text{testScope} \to (x, \text{testScope-body}, \varnothing) \\
&\qquad \text{line 6: testScope } 2.0 \to \cancel{?}\ 8.0 \\
&\qquad \text{line 6: stdout} \to \text{"8.0"} \\
&\cancel{E_1}: (x \to 2.0, \text{testScope-body}, \varnothing) \\
&\qquad a \to \alpha_1 \\
&\qquad \text{f} \to (z, \text{a * z}, (a \to \alpha_1, x \to 2.0)) \\
&\qquad \text{line 5: f x} \to \cancel{?}\ 8.0 \\
&\cancel{E_2}: (z \to 2.0, \text{a * z}, (a \to \alpha_1, x \to 2.0)
\end{aligned}
$$

By the above examples, it is seen that hand tracing can be used to in detail study the flow of data through a program. It may seem tedious in the beginning, but the care illustrated above is useful at the start to ensure rigor in the analysis. Most will find that once accustomed to the method, the analysis can be performed rigorously but with less paperwork, and in conjunction with strategically placed debugging `printfn` statements, it is a very valuable tool for debugging.