Chapter 1

Organising Code in Libraries and Application Programs

Abstract We have previously seen how code can be organized into functions to make programs easier to read, make code pieces reusable, and make programs easier to debug. Functions and values may further be grouped into libraries, and the `List` module is an example of such a library that you have already used. F# includes several programming structures to organize code in libraries: Modules, namespaces, and classes. In this chapter, we will focus on modules. Classes will be described in detail in **??**. Here you will learn how to:

- Use the `dotnet` command-line tool to create project files and how to compile these into an executable file.
- The difference between running interpreted and compiled programs.
- Make libraries using modules and write applications using such libraries.
- Specify the abstract structure of a module using signature files.
- Create an implementation of the Stack abstract datatype.
- Update an integer stack to a generic stack.

1.1 Dotnet projects: Libraries and applications

As our programs grow in size, it can be convenient to split the program over several files, e.g., by separating functionality into something general and specific for the problem being solved. An example of this is the `List` module which contains general functions on lists, and which you have used in your programs. In this chapter, we will write modules ourselves, also known as libraries, and the programs using these libraries, we will call applications. Using the `dotnet` command-line tool, we are able to create project files which have a `.fsproj` suffix, which include information about which source code and packages belongs together. The `dotnet` command-line tool can help structure the files on the filesystem by use of directories, but here we advocate for a simple, hand-held solution.

To make a light version of `dotnet` project with a library and an application file, start by creating the `dotnet` project template as follows:

Listing 1.1: Creating an initial library-application file setup.

```
$ dotnet new console -lang "F#" -o app
```

This creates the `app` directory with among other things a `Program.fs` file. The `Program.fs` is the default filenames for an application. Usually, the `.fs` suffix is reserved for libraries, so, rename `Program.fs` to `Program.fsx`. Then create a possibly empty library file `Library.fs` using a standard editor. You should now have a directory as shown in Figure 1.1. The `.fsproj` file is an XML-file which

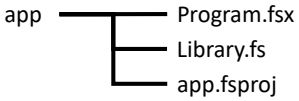

Fig. 1.1 A set of files for a light version of a dotnet project.

describes how dotnet should combine various files. The `dotnet` command-line tool can edit this file, but we might as well do this ourselves in a text editor: Open the file in your favorite text editor and in the `<ItemGroup>` change `Program.fs` to `Program.fsx` and add a line with the name of your library file `<Compile Include="Library.fs" />`. The resulting file should look like this:

Listing 1.2: The initial content of `app.fsproj`.

```
<Project Sdk="Microsoft.NET.Sdk">
 <PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net6.0</TargetFramework>
 </PropertyGroup>
 <ItemGroup>
  <Compile Include="Library.fs" />
  <Compile Include="Program.fsx" />
 </ItemGroup>
</Project>
```

If you decide to rename the application or library files, then you must update the project files accordingly.

If you wish to add references to packages such as the `DIKU.Canvas` package, this can be done as,

Listing 1.3: Creating an initial library-application file setup.

```
$ dotnet add app/app.fsproj package "DIKU.Canvas" --version
   1.0.1
```

or manually by editing the project file appropriately. When a package is included in the project file, then it does not need to be loaded in libraries and applications using the `#r` directive. This version will compile and run the library and the program, but will not build the library separately.

The order of the references to packages, libraries, and application files are important, since `dotnet` will read them from top to bottom, and only if, e.g., `Library.fs` is above `Program.fsx` will the library functions be available in the application.

As an example, change `Program.fs` to become what is shown in Listing 1.4, change

Listing 1.4 solution/app/Program.fs:
A simple application program.

```
open Library

printfn "%A" (greetings "Jon")
```

`Library.fs` to become what is shown in Listing 1.5, and run it in *compile mode* by

Listing 1.5 solution/library/Library.fs:
A simple library.

```
module Library

let greetings (str: string) : string = "Greetings " + str
```

changing to the `app` directory and using the `dotnet run` command as demonstrated in Listing 1.6.

Listing 1.6: Running an application setup with one or more project files.

```
$ cd solution/app
$ dotnet run
"Greetings Jon"
```

Assuming that `Program.fs` was renamed to `Program.fsx` and `app.fsproj` was edited appropriately, `dotnet run` is almost the same as

Listing 1.7: Running an application setup with one or more project files.

```
$ dotnet fsi ../library/Library.fs Program.fsx
"Greetings Jon"
```

However, `dotnet fsi` *interprets* the library and application into executable code everytime it is called, while `dotnet run` only *compiles* the program once. On my laptop, the time these different steps take depends on what else is running on the computer, but typical timings are

Command	Time
`dotnet fsi ../library/Library.fs Program.fsx`	1.2s
`dotnet run` (first time)	4.0s
`dotnet run`	1.0s

The example application, we are studying here, is tiny, but even in this case, the repeated translation by `dotnet fsi` is a 16% overhead when compared to an already compiled program, and you should expect this overhead to be larger for larger programs. However, for tiny programs, the cost of the initial compilation is 400% and not worth the effort from a time perspective.

1.2 Libraries and applications

A library in F# is expressed as a *module*, which is a programming structure used to organize type declarations, values, functions, etc. The libraries should have the suffix `.fs`, and here will call them *implementation files* in contrast to the signature files to be discussed below, which we will call *signature files*.

A module is typically a file where the module name is declared in the first lines using the `module` with the following syntax,

Listing 1.8: Outer module.

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

Here, the identifier `<ident>` is a name not necessarily related to the filename, and the script `<script>` is an expression.

Consider the example from **??** in which functions are defined for solving the values of x where $f(x) = 0$ for a quadratic equation. In the following, we will split this into a library of functions and an application program. For this, we set up a project system of files as described in Section 1.1, where `Program.fs` has been replaced by `Program.fsx` and the `app.fsproj` has been edited appropriately. The content of `Library.fs` has been changed to become what is shown in Listing 1.9, and

Listing 1.9 solve/library/Library.fs:
A library for solving quadratic equations.

```
module Solve

let discriminant a b c = b ** 2.0 - 4.0 * a * c

let solveQuadraticEquation a b c =
  let d = discriminant a b c
  ((-b + sqrt d) / (2.0 * a),
   (-b - sqrt d) / (2.0 * a))
```

`Program.fsx` has been changed to what is shown in Listing 1.10.

Listing 1.10 solve/app/Program.fsx:
An application using the `Solve` module.

```
open Solve

let p1 = solveQuadraticEquation 1.0 0.3 -1.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p1
let p2 = solveQuadraticEquation 1.0 0.0 0.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p2
let p3 = solveQuadraticEquation 1.0 0.0 1.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p3
```

1.3 Specifying a Module's Interface with a Signature File

As the 8-step guide suggests, the design of programs is helped by first considering what the program's function is to do before actually implementing them. This also holds for libraries, and signature files can aid this process.

A *signature files* is a file accompanying *implementation files* and have the suffix `.fsi`. A signature file contains almost no implementation, but only type definitions. Signature files offer three distinct features:

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 focuses 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 related to classes is available and will be discussed in **??**.

These features help the programmer structure the process of programming and protect the user of a library from irrelevant data and functions. A signature file contains the type definitions and the types of values and functions to be exposed to the user of the library. For example, for the library in Listing 1.9, we can define a signature files which makes the `solveQuadraticEquation` function but not the `discriminant` function available to the user of the library as demonstrated in Listing 1.11. To compile the application using the signature file, we must add the

Listing 1.11 solve/library/Library.fsi: A signature file for Listing 1.9.

```
1 module Solve
2
3 val solveQuadraticEquation: a: float -> b: float -> c: float
     -> float*float
```

created file, e.g., `Library.fsi`, to the project file as, e.g., shown in Listing 1.12.

Listing 1.12: The `library.fsproj` with a signature file added.

```
<Project Sdk="Microsoft.NET.Sdk">
 <PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net6.0</TargetFramework>
 </PropertyGroup>
 <ItemGroup>
  <PackageReference Include="DIKU.Canvas" Version="1.0.1" />
  <Compile Include="Library.fsi" />
  <Compile Include="Library.fs" />
  <Compile Include="Program.fsx" />
 </ItemGroup>
</Project>
```

In the context of the 8-step guide, it is useful to write the signature file before the implementation file, and that the signature file contains the documentation for the functions available in the application.

For technical reasons in the `dotnet` framework, exposed type abbreviations must be given both in the signature file and the implementation file. The implication is that the signature at times must also define implementation details, see e.g., the example below, and thus becomes less abstract the desirable in general.

1.4 Programming Intermezzo: Postfix Arithmetic with a Stack

To this point, we have performed simple arithmetic using *infix* notation, meaning that expressions like $(4+6*3)/2-8$ are evaluated using the precedence and association rules of the operators as

$$\begin{aligned}(4+6*3)/2-8 &\rightsquigarrow (4+18)/2-8 && (1.1)\\ &\rightsquigarrow 22/2-8 && (1.2)\\ &\rightsquigarrow 11-8 && (1.3)\\ &\rightsquigarrow 3 && (1.4)\end{aligned}$$

However, there is an equally valid notation, *postfix*, in which the same expression is written as 4 6 3 $*$ + 2 / 8 −. Here, the rule is to read from left to right, and whenever there are two values and an operator, a b op, replaced this with the value a op b and repeat until only one value remains, which is the result of the calculation. Hence,

$$4\,6\,3\,*\,+\,2\,/\,8\,- \rightsquigarrow 4\,18\,+\,2\,/\,8\,- \tag{1.5}$$
$$\rightsquigarrow 22\,2\,/\,8\,- \tag{1.6}$$
$$\rightsquigarrow 11\,8\,- \tag{1.7}$$
$$\rightsquigarrow 3 \tag{1.8}$$

This was implemented on a series of calculators released by Hewlett-Packard in the 1960-1980'ies, and one of the arguments for this notation was, that the expressions could be evaluated by a stack with only 3 levels. In the following, we will look at stacks as an abstract datatype and build a library for stacks and an arithmetic solver for such simple expressions using this stack.

A *stack* is an abstract datatype, meaning that it is defined by its concepts, not its implementation. The concept of a stack is like a stack of plates in a cafeteria, they are placed in a physical stack, and you can take the top plate and place a plate on the top, but you cannot access a plate in the middle of a stack. Stacks typically come with the following functions:

create: Create an empty stack.

pop: Return the top element and the resulting stack.

push: Put an element on a stack and return the resulting stack.

isEmpty: Check whether the stack is empty.

Following the 8-step guide **??**, the above directly suggests names and includes brief descriptions (Steps 1 and 2). Step 3 suggests that we write a simple test, and since we are fond of piping, our test program is shown in Listing 1.13. We expect this

Listing 1.13 postfixTest.fsx:
A simple program using a yet to be written library.

```
1 open Stack
2
3 create () |> push 1.0 |> push 2.0 |> pop |> printfn "%A"
```

to print the result of the last `pop` call, which should include information about the element `2`.

In the functional programming paradigm, our stack is a constant, implying that every time we pop and push, we create new stacks. Thus, for step 4 in the 8-step guide, we must accept that all but `isEmpty` returns a new stack, and all but `create` must take a stack as input. Thus we arrive at a signature file for the stack-library given in Listing 1.14. A limitation to F#'s modules is that the type specifications need explicit declaration. We would have liked to write `type stack` and functions of some variable type `'e`, since the stack concept is independent of the type of elements

Listing 1.14 postfixLibrary.fsi:
A signature file for the stack library

```
module Stack

type stack = float list // a stack of elements

// create an empty stack
val create: unit -> stack
// return the top element and the resulting stack
val pop: stack -> float * stack
// put an element on a stack and return the resulting stack
val push: float -> stack -> stack
// check whether the stack is empty
val isEmpty: stack -> bool
```

it contains. However, this is not possible, and thus, we here specialize to float stacks. For similar reasons, we are forced to specify details about the implementation of our type. Our idea is that stacks can be implemented as lists since lists are well suited to work with the first elements.

Implementing a stack using lists is simple, since lists already contains the properties `Head`, `Tail`, and `IsEmpty`, which closely mimics the needed operations for a stack. Thus we arrive at Listing 1.15. And now we can run our test code as shown in

Listing 1.15 postfixLibrary.fs:
An implementation of a stack module.

```
module Stack

type stack = float list
let create () : stack = []
let pop (stck: stack) : float*stack =  (stck.Head, stck.Tail)
let push (elm: float) (stck: stack): stack = elm::stck
let isEmpty (stck: stack): bool = stck.IsEmpty
```

Listing 1.16. As expected, the top element and the resulting stack is `(2,[1])`.

Listing 1.16: Running the test program

```
$ dotnet fsi postfixLibrary.fsi postfixLibrary.fs
   postfixTest.fsx
(2.0, [1.0])
```

To implement simple postfix algebra, we will use discriminated unions. That is, we define a type,

```
type element = Value of int | Multiply | Plus | Minus | Divide
```

This allows us to make lists of tokens such as,

```
[Value 4; Value 6; Value 3; Multiply; Plus]
```

for the expression 3 4 2 / + which is equivalent to $3+4/2$ in infix notation. The next step is to understand how to use a stack to evaluate such expressions. The idea is to process the list of tokens from its head and push values to a stack. When the head of the tokens list is an operator, say 'op' then the two top elements from the stack are popped, say a and b, the mathematical expression $c = a$ op b is evaluated and c is pushed to the stack. For our example, the evolution of the stack will be:

Unused tokens	Evaluation stack
4 6 3 ∗ + 2 / 8 −	[]
6 3 ∗ + 2 / 8 −	[4]
3 ∗ + 2 / 8 −	[6; 4]
∗ + 2 / 8 −	[3; 6; 4]
+ 2 / 8 −	[18; 4]
2 / 8 −	[22]
/ 8 −	[2; 22]
8 −	[11]
−	[8; 11]
	[3]

As demonstrated, the result of the expression is the last element on the stack, once the list of tokens is empty. An F# implementaiton is given in Listing 1.17.

1.5 Generic Modules

The stack is an example of an abstract datatype, and in the previous section, we implemented a stack for floats, however, stacks can be of many other types, and although we could make a stack module for each type, it would greatly improve the usefulness of our library, if we could make a stack module, which is generic, i.e., where the user can decide when writing applications, which type of values to stack. Luckily, this is supported in F#.

In **??** we discussed the usefulness of the variable type such as `'a`, which makes functions and types generic, that is, the same definition can be used for any type. To make a generic module, it is often useful first to make a non-generic version, such as our float stack, since it is often easier to spot errors in concrete program versions. The float stack already works as desired, so we will now modify the module to be a stack for a variable type. In our example, we must update both the type abbreviation and function types in the signature file. The result is shown in Listing 1.18. The next step is to update the implementation file. During such transformations, it is not

Listing 1.17 postfixApp.fsx:
An application for evaluating lists of tokens on postfix form using a stack.

```
open Stack

type element = Value of float | Mul | Add | Sub | Div

let tokens = [Value 4.0; Value 6.0; Value 3.0; Mul; Add;
   Value 2.0; Div; Value 8.0; Sub]

let rec eval (tkns: element list) (stck: stack): stack =
  match tkns with
    [] -> stck
    | elm::rst ->
      match elm with
        Value v ->
          push v stck |> eval rst
        | Mul ->
          let (a, stck1) = pop stck
          let (b, stck2) = pop stck1
          push (b*a) stck2 |> eval rst
        | Add ->
          let (a, stck1) = pop stck
          let (b, stck2) = pop stck1
          push (b+a) stck2 |> eval rst
        | Sub ->
          let (a, stck1) = pop stck
          let (b, stck2) = pop stck1
          push (b-a) stck2 |> eval rst
        | Div ->
          let (a, stck1) = pop stck
          let (b, stck2) = pop stck1
          push (b/a) stck2 |> eval rst

printfn "%A = %A" tokens (eval tokens (create ()))
```

```
$ dotnet fsi postfixLibrary.fsi postfixLibrary.fs
   postfixApp.fsx
[Value 4.0; Value 6.0; Value 3.0; Mul; Add; Value 2.0; Div;
   Value 8.0; Sub] = [3.0]
```

uncommon to realize that restrictions must be put on the type, which is possible but which we will not consider further in this book. Since the stack does not rely on any properties of the stack elements, there is no challenge in modifying the implementation file as shown in Listing 1.19. Finally, we can make an application using stacks of various kinds, as shown in Listing 1.20. In the program, we see that F# can infer that a stack, on which floats are pushed, must be of `stack<float>` type, and when characters are pushed, then the stack must be of type `stack<char>`. Thus, we have arrived at a stack for any type, whose interface is given by the signature file, and almost all of its implementation is hidden in the implementation file.

Listing 1.18 postfixLibraryGeneric.fsi:
A signature file for the generic stack library

```
module Stack

type stack<'a> = 'a list // a stack of elements

// create an empty stack
val create: unit -> stack<'a>
// return the top element and the resulting stack
val pop: stack<'a> -> 'a * stack<'a>
// put an element on a stack and return the resulting stack
val push: 'a -> stack<'a> -> stack<'a>
// check whether the stack is empty
val isEmpty: stack<'a> -> bool
```

Listing 1.19 postfixLibraryGeneric.fs:
An implementation of a generic stack module.

```
module Stack

type stack<'a> = 'a list
let create () : stack<'a> = []
let pop (stck: stack<'a>) : 'a*stack<'a> =  (stck.Head,
   stck.Tail)
let push (elm: 'a) (stck: stack<'a>): stack<'a> = elm::stck
let isEmpty (stck: stack<'a>): bool = stck.IsEmpty
```

Listing 1.20 postfixTestGeneric.fsx:
Running the test program

```
open Stack

create () |> push 1.0 |> push 2.0 |> pop |> printfn "%A"
create () |> push 'a' |> push 'b' |> pop |> printfn "%A"
```

```
$ dotnet fsi postfixLibraryGeneric.fsi
   postfixLibraryGeneric.fs postfixTestGeneric.fsx
(2.0, [1.0])
('b', ['a'])
```

1.6 Key Concepts and Terms in This Chapter

In this chapter, we have looked at how to build libraries and applications. Key concepts have been

- How to build organise the compilation of libraries and applications using `dotnet` **project files**.

- How to design libraries using **modules** and **signature files**.
- How to make **library implementation files**.
- How to implement the **abstract datatype Stack** both as a float stack and as a **generic module**.